iAnywhere®
A SYBASE COMPANY

MobiLink Administration Guide

Part number: DC38132-01-0902-01
Last modified: October 2004

Contents

About This Manual

Subject This manual describes MobiLink, a session-based relational-database synchronization system. MobiLink technology allows two-way replication and is well suited to mobile computing environments.

Audience This manual is for users of Adaptive Server Anywhere and other relational database systems who wish to add synchronization or replication to their information systems.

Before you begin ☞ For a comparison of MobiLink with other synchronization and replication technologies, see “ Introducing Replication Technologies” [*Introducing SQL Anywhere Studio,* page 21].

SQL Anywhere Studio documentation

This book is part of the SQL Anywhere documentation set. This section describes the books in the documentation set and how you can use them.

The SQL Anywhere Studio documentation

The SQL Anywhere Studio documentation is available in a variety of forms: in an online form that combines all books in one large help file; as separate PDF files for each book; and as printed books that you can purchase. The documentation consists of the following books:

- **Introducing SQL Anywhere Studio** This book provides an overview of the SQL Anywhere Studio database management and synchronization technologies. It includes tutorials to introduce you to each of the pieces that make up SQL Anywhere Studio.
- **What's New in SQL Anywhere Studio** This book is for users of previous versions of the software. It lists new features in this and previous releases of the product and describes upgrade procedures.
- **Adaptive Server Anywhere Database Administration Guide** This book covers material related to running, managing, and configuring databases and database servers.
- **Adaptive Server Anywhere SQL User's Guide** This book describes how to design and create databases; how to import, export, and modify data; how to retrieve data; and how to build stored procedures and triggers.
- **Adaptive Server Anywhere SQL Reference Manual** This book provides a complete reference for the SQL language used by Adaptive Server Anywhere. It also describes the Adaptive Server Anywhere system tables and procedures.
- **Adaptive Server Anywhere Programming Guide** This book describes how to build and deploy database applications using the C, C++, and Java programming languages. Users of tools such as Visual Basic and PowerBuilder can use the programming interfaces provided by those tools. It also describes the Adaptive Server Anywhere ADO.NET data provider.
- **Adaptive Server Anywhere SNMP Extension Agent User's Guide** This book describes how to configure the Adaptive Server Anywhere SNMP Extension Agent for use with SNMP management applications to manage Adaptive Server Anywhere databases.
- **Adaptive Server Anywhere Error Messages** This book provides a complete listing of Adaptive Server Anywhere error messages together with diagnostic information.

- **SQL Anywhere Studio Security Guide** This book provides information about security features in Adaptive Server Anywhere databases. Adaptive Server Anywhere 7.0 was awarded a TCSEC (Trusted Computer System Evaluation Criteria) C2 security rating from the U.S. Government. This book may be of interest to those who wish to run the current version of Adaptive Server Anywhere in a manner equivalent to the C2-certified environment.

- **MobiLink Administration Guide** This book describes how to use the MobiLink data synchronization system for mobile computing, which enables sharing of data between a single Oracle, Sybase, Microsoft or IBM database and many Adaptive Server Anywhere or UltraLite databases.

- **MobiLink Clients** This book describes how to set up and synchronize Adaptive Server Anywhere and UltraLite remote databases.

- **MobiLink Server-Initiated Synchronization User's Guide** This book describes MobiLink server-initiated synchronization, a feature of MobiLink that allows you to initiate synchronization from the consolidated database.

- **MobiLink Tutorials** This book provides several tutorials that walk you through how to set up and run MobiLink applications.

- **QAnywhere User's Guide** This manual describes MobiLink QAnywhere, a messaging platform that enables the development and deployment of messaging applications for mobile and wireless clients, as well as traditional desktop and laptop clients.

- **iAnywhere Solutions ODBC Drivers** This book describes how to set up ODBC drivers to access consolidated databases other than Adaptive Server Anywhere from the MobiLink synchronization server and from Adaptive Server Anywhere remote data access.

- **SQL Remote User's Guide** This book describes all aspects of the SQL Remote data replication system for mobile computing, which enables sharing of data between a single Adaptive Server Anywhere or Adaptive Server Enterprise database and many Adaptive Server Anywhere databases using an indirect link such as e-mail or file transfer.

- **SQL Anywhere Studio Help** This book includes the context-sensitive help for Sybase Central, Interactive SQL, and other graphical tools. It is not included in the printed documentation set.

- **UltraLite Database User's Guide** This book is intended for all UltraLite developers. It introduces the UltraLite database system and provides information common to all UltraLite programming interfaces.

- **UltraLite Interface Guides** A separate book is provided for each UltraLite programming interface. Some of these interfaces are provided as UltraLite components for rapid application development, and others are provided as static interfaces for C, C++, and Java development.

In addition to this documentation set, PowerDesigner and InfoMaker include their own online documentation.

Documentation formats

SQL Anywhere Studio provides documentation in the following formats:

- **Online documentation** The online documentation contains the complete SQL Anywhere Studio documentation, including both the books and the context-sensitive help for SQL Anywhere tools. The online documentation is updated with each maintenance release of the product, and is the most complete and up-to-date source of documentation.

  To access the online documentation on Windows operating systems, choose Start ➤ Programs ➤ SQL Anywhere 9 ➤ Online Books. You can navigate the online documentation using the HTML Help table of contents, index, and search facility in the left pane, as well as using the links and menus in the right pane.

  To access the online documentation on UNIX operating systems, see the HTML documentation under your SQL Anywhere installation.

- **PDF books** The SQL Anywhere books are provided as a set of PDF files, viewable with Adobe Acrobat Reader.

  The PDF books are accessible from the online books, or from the Windows Start menu.

- **Printed books** The complete set of books is available from Sybase sales or from eShop, the Sybase online store, at *http://eshop.sybase.com/eshop/documentation*.

Documentation conventions

This section lists the typographic and graphical conventions used in this documentation.

Syntax conventions

The following conventions are used in the SQL syntax descriptions:

- **Keywords** All SQL keywords appear in upper case, like the words ALTER TABLE in the following example:

  ALTER TABLE [*owner*.]*table-name*

- **Placeholders** Items that must be replaced with appropriate identifiers or expressions are shown like the words *owner* and *table-name* in the following example:

  ALTER TABLE [*owner*.]*table-name*

- **Repeating items** Lists of repeating items are shown with an element of the list followed by an ellipsis (three dots), like *column-constraint* in the following example:

  ADD *column-definition* [*column-constraint*, . . .]

  One or more list elements are allowed. In this example, if more than one is specified, they must be separated by commas.

- **Optional portions** Optional portions of a statement are enclosed by square brackets.

  RELEASE SAVEPOINT [*savepoint-name*]

  These square brackets indicate that the *savepoint-name* is optional. The square brackets should not be typed.

- **Options** When none or only one of a list of items can be chosen, vertical bars separate the items and the list is enclosed in square brackets.

  [**ASC** | **DESC**]

  For example, you can choose one of ASC, DESC, or neither. The square brackets should not be typed.

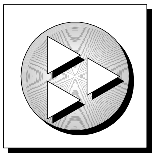
- **Alternatives** When precisely one of the options must be chosen, the alternatives are enclosed in curly braces and a bar is used to separate the options.

  [**QUOTES** { **ON** | **OFF** }]

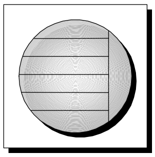
  If the QUOTES option is used, one of ON or OFF must be provided. The brackets and braces should not be typed.

Graphic icons The following icons are used in this documentation.

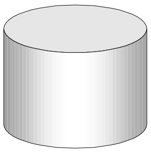
- A client application.



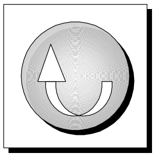
- A database server, such as Sybase Adaptive Server Anywhere.



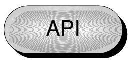
- A database. In some high-level diagrams, the icon may be used to represent both the database and the database server that manages it.



- Replication or synchronization middleware. These assist in sharing data among databases. Examples are the MobiLink Synchronization Server and the SQL Remote Message Agent.



- A programming interface.



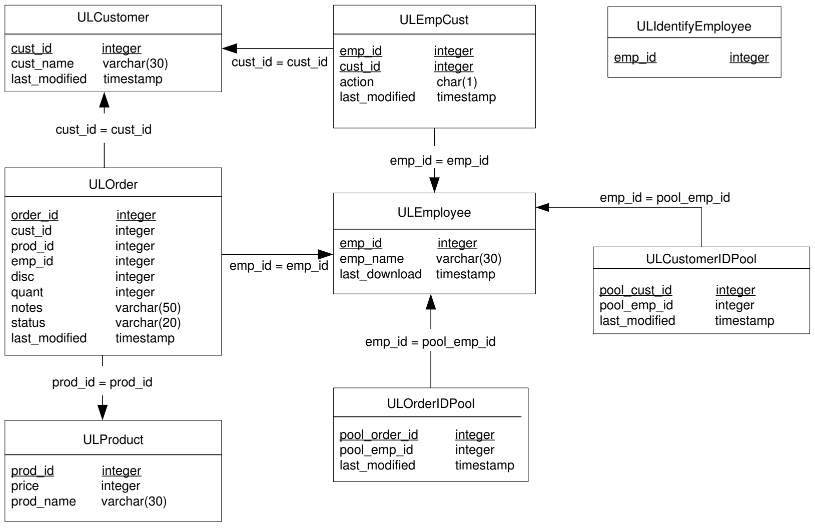
The CustDB sample database

Many of the examples in the MobiLink and UltraLite documentation use the UltraLite sample database.

The reference database for the UltraLite sample database is held in a file named *custdb.db*, and is located in the *Samples\UltraLite\CustDB* subdirectory of your SQL Anywhere directory. A complete application built on this database is also supplied.

The sample database is a sales-status database for a hardware supplier. It holds customer, product, and sales force information for the supplier.

The following diagram shows the tables in the CustDB database and how they are related to each other.



Finding out more and providing feedback

Finding out more

Additional information and resources, including a code exchange, are available at the iAnywhere Developer Network at *http://www.ianywhere.com/developer/*.

If you have questions or need help, you can post messages to the iAnywhere Solutions newsgroups listed below.

When you write to one of these newsgroups, always provide detailed information about your problem, including the build number of your version of SQL Anywhere Studio. You can find this information by typing **dbeng9 -v** at a command prompt.

The newsgroups are located on the *forums.sybase.com* news server. The newsgroups include the following:

- sybase.public.sqlanywhere.general
- sybase.public.sqlanywhere.linux
- sybase.public.sqlanywhere.mobilink
- sybase.public.sqlanywhere.product_futures_discussion
- sybase.public.sqlanywhere.replication
- sybase.public.sqlanywhere.ultralite
- ianywhere.public.sqlanywhere.qanywhere

Newsgroup disclaimer

iAnywhere Solutions has no obligation to provide solutions, information or ideas on its newsgroups, nor is iAnywhere Solutions obliged to provide anything other than a systems operator to monitor the service and ensure its operation and availability.

iAnywhere Solutions Technical Advisors as well as other staff assist on the newsgroup service when they have time available. They offer their help on a volunteer basis and may not be available on a regular basis to provide solutions and information. Their ability to help is based on their workload.

Feedback

We would like to receive your opinions, suggestions, and feedback on this documentation.

You can e-mail comments and suggestions to the SQL Anywhere documentation team at iasdoc@ianywhere.com. Although we do not reply to e-mails sent to that address, we read all suggestions with interest.

In addition, you can provide feedback on the documentation and the software through the newsgroups listed above.

PART I

USING MOBILINK TECHNOLOGY

This part introduces MobiLink synchronization technology and describes how to use it to replicate data between two or more databases.

CHAPTER 1

Introducing MobiLink Synchronization

About this chapter

This chapter introduces you to MobiLink synchronization technology. It describes the purpose and characteristics of MobiLink.

☞ For information about MobiLink clients, see "Introducing MobiLink Clients" [*MobiLink Clients*, page 3].

☞ For a more detailed introduction to MobiLink technology, see "Synchronization Basics" on page 7.

Contents

The MobiLink synchronization process

MobiLink is a session-based synchronization system that allows two-way synchronization between a main database, called the consolidated database, and many remote databases. The consolidated database, which can be one of several ODBC-compliant databases, holds the master copy of all the data. Remote databases can be either Adaptive Server Anywhere or UltraLite databases.

Synchronization typically begins when a MobiLink remote site opens a connection to a MobiLink synchronization server. During synchronization, the MobiLink client at the remote site uploads database changes that were made to the remote database since the previous synchronization. On receiving this data, the MobiLink synchronization server updates the consolidated database, and then downloads changes on the consolidated database to the remote database.

MobiLink features

MobiLink synchronization is adaptable and flexible. Following are some of its key features:

- **Data coordination** MobiLink allows you to choose selected portions of the data for synchronization. MobiLink synchronization also allows you to resolve conflicts between changes made in different databases. The synchronization process is controlled by synchronization logic, which can be written as a SQL, Java, or .NET application. Each piece of logic is called a **script**. With scripts, for example, you can specify how uploaded data is applied to the consolidated, specify what gets downloaded, and handle different schema and names between the consolidated and remote databases.

- **Automation** MobiLink has a number of automated capabilities. The MobiLink synchronization server can be instructed to generate scripts suitable for snapshot synchronization, or instructed to generate example synchronization scripts. It can also automatically add users for authentication. Server-initiated synchronization allows you to push data updates to remote databases.

- **Monitoring and reporting** MobiLink provides two mechanisms for monitoring your synchronizations: the MobiLink Monitor, and statistical scripts. You can monitor scripts, schema contents, row-count values, script names, translated script contents, and row values.

- **Performance tuning** There are a number of mechanisms for tuning MobiLink performance. For example, you can adjust the degree of contention, upload cache size, number of database connections, number of worker threads, logging verbosity, or BLOB cache size.

- **Two-way synchronization** Changes to a database can be made at any location.
- **Upload-only or download-only synchronization** You can choose to perform only an upload or only a download.
- **File-based download** Downloads can be distributed as files, enabling offline distribution of synchronization changes. This allows you to create a file once and distribute it widely.
- **Server-initiated synchronization** You can initiate MobiLink synchronization from the consolidated database. This means you can push data updates to remote databases, as well as cause remote databases to upload data to the consolidated database.
- **Choice of communication streams** Synchronization can be carried out over TCP/IP, HTTP, or HTTPS. Palm devices can synchronize through HotSync. Windows CE devices can synchronize using ActiveSync.
- **Remote-initiated** Synchronization between a remote database and a consolidated database can be initiated at the remote database.
- **Session-based** All changes can be uploaded in a single transaction and downloaded in a single transaction. At the end of each successful synchronization, the consolidated and remote databases are consistent.
- **Transactional integrity** Either a whole transaction is synchronized, or none of it is synchronized. This ensures transactional integrity for each database.
- **Data consistency** MobiLink operates using a *loose consistency* policy. All changes are synchronized with each site over time in a consistent manner, but different sites may have different copies of data at any instant.
- **Wide variety of hardware and software platforms** A variety of widely-used database management systems can be used as a MobiLink consolidated database: Adaptive Server Anywhere, Adaptive Server Enterprise, Oracle, Microsoft SQL Server, or IBM DB2. Remote databases can be Adaptive Server Anywhere or UltraLite databases. The MobiLink synchronization server runs on Windows or UNIX platforms. Adaptive Server Anywhere runs on Windows, Windows CE, or UNIX machines. UltraLite runs on Palm, Windows CE, or Java-based devices.

- **Flexibility** The MobiLink synchronization server uses SQL, Java, or .NET scripts to control the upload and download of data. The scripts are executed according to an event model during each synchronization. Event-based scripting provides great flexibility in the design of the synchronization process, including such features as conflict resolution, error reporting, and user authentication.

- **Scalability and performance** MobiLink synchronization is scalable: a single server can handle thousands of simultaneous synchronizations, and multiple MobiLink servers can be run simultaneously using load balancing. The MobiLink synchronization server is multi-threaded and uses connection pooling with the consolidated database. MobiLink provides extensive monitoring and reporting facilities.

- **Easy to get started** Simple MobiLink installations can be constructed quickly. More complex refinements can be added incrementally for full-scale production work.

CHAPTER 2

Synchronization Basics

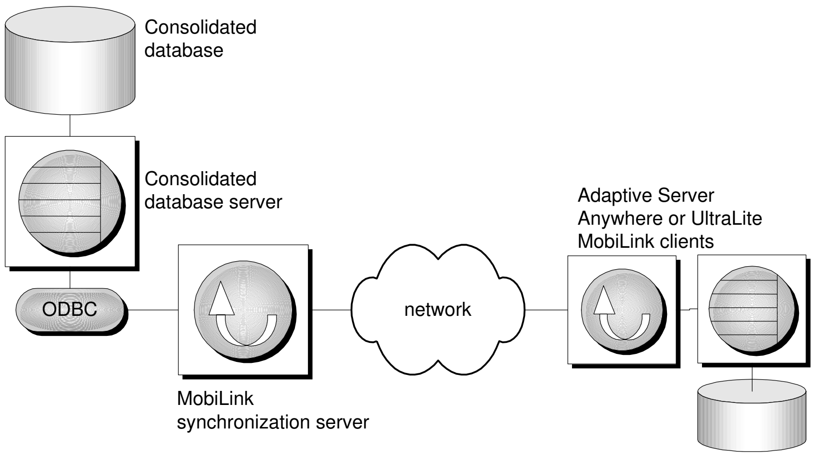
About this chapter

This chapter introduces the basic components of MobiLink technology and provides information about how to set up your synchronization system.

Contents

Parts of the synchronization system

The following diagram shows the major parts of the synchronization system.


Consolidated database
Consolidated database server
ODBC
MobiLink synchronization server
network
Adaptive Server Anywhere or UltraLite MobiLink clients

- **consolidated database** This database contains the central copy of all information in the synchronization system.

  ☞ For more information, see "Consolidated database" on page 10.

- **consolidated database server** The server, or DBMS, that manages the consolidated database. This server can be a Sybase product, such as Adaptive Server Anywhere or Adaptive Server Enterprise, or it may be a supported system made by another company.

  ☞ For more information, see "MobiLink Consolidated Databases" on page 31.

- **ODBC connection** All communication between the MobiLink synchronization server and the consolidated database occurs through an ODBC connection. ODBC allows the synchronization server to utilize a variety of consolidated database systems.

  ☞ For more information, see "iAnywhere Solutions ODBC Drivers" on page 547.

- **MobiLink synchronization server** This server manages the synchronization process and provides the interface between all MobiLink clients and the consolidated database server.

  ☞ For more information, see "The MobiLink synchronization server" on page 11.

♦ **Network** The connection between the MobiLink synchronization server, dbmlsrv9, and the MobiLink client, dbmlsync or UltraLite, can use a number of protocols.

☞ For more information about connecting to dbmlsync, see "-x option" [*MobiLink Clients*, page 151]. For information about connecting to UltraLite, see "Network protocol options for UltraLite synchronization clients" [*MobiLink Clients*, page 341].

♦ **MobiLink client** The client can be installed on a handheld device such as a Palm Pilot or PocketPC, a server or desktop computer, or an embedded device such as a cell phone or vending machine. Two types of clients are supported: UltraLite and Adaptive Server Anywhere databases. Either or both may be used in a single MobiLink installation.

☞ For more information, see "MobiLink clients" on page 14.

Consolidated database

Applications synchronize with a central, consolidated database. This database is the master repository of information in the synchronization system.

There are many ways to structure the relations between consolidated and remote databases. Following are two examples.

The schema of the remote databases can be a subset of the schema of the consolidated database. For example, a table called emp might be repeated among a number of different remote sites, and the consolidated database might use column data from emp.salary in a table called expense. In this instance, the schemas of the consolidated and remote databases are different, though data is shared.

The schema of the remote database can also be parallel in structure to the schema of the consolidated database. Here, the schema of the consolidated database is a reference for the remote database. In the consolidated database, you may already have tables that correspond to each of the remote tables. In this instance, the schemas in the consolidated and remote databases are virtually the same, and the data in the remote is only a subset of the data on the consolidated.

You write **synchronization scripts** for each table in the remote database and you save these scripts on the consolidated database. These scripts, from their central location on the consolidated database, direct the synchronization server in moving data between remote and consolidated databases. One script for a particular remote table tells the synchronization server where to store data uploaded from that remote table in the consolidated database. Another script tells the synchronization server which data to download to the same remote table.

☞ For more information about consolidated databases, see "MobiLink Consolidated Databases" on page 31.

The MobiLink synchronization server

All MobiLink clients synchronize through the MobiLink synchronization server. None connect directly to a database server. You must start the MobiLink synchronization server before asking a MobiLink client to synchronize.

Running the MobiLink synchronization server

The MobiLink synchronization server opens connections, via ODBC, with your consolidated database server. It then accepts connections from remote applications and controls the synchronization process.

❖ To start the MobiLink synchronization server

1. Run dbmlsrv9. Use the –c option to specify the ODBC connection parameters for your consolidated database.

☞ For information about connection parameters, see "-c option" on page 196.

You must specify connection parameters. Other options are available, but are optional. These options allow you to specify how the server works. For example, you can specify a maximum number of worker threads, cache size, and logging options.

☞ For more information about dbmlsrv9 options, see "MobiLink Synchronization Server Options" on page 189.

Note: The dbmlsrv9 options allow you to specify how the MobiLink synchronization server works. To control what the server does, you define scripts that are invoked at synchronization events.

☞ For more information, see "MobiLink events" on page 16.

Example

The following command starts the MobiLink synchronization server, identifying the ODBC data source *UltraLite 9.0 Sample* as the consolidated database. Enter the entire command on one line.

```
dbmlsrv9
  -c "dsn=UltraLite 9.0 Sample;uid=DBA;pwd=SQL"
  -zs MyServer
  -o mlsrv.log
  -vcr
  -x tcpip
```

In this example, the -zs option provides a server name. The –o option specifies that the log file should be named *mlsrv.log*. The contents of *mlsrv.log* are verbose because of the –vcr option. The –x option specifies that MobiLink clients will be permitted to connect via TCP/IP.

☞ You can also start the MobiLink synchronization server as a Windows service or UNIX daemon. For more information, see "Running MobiLink Outside the Current Session" on page 157.

Stopping the MobiLink synchronization server

The MobiLink synchronization server can be stopped from the computer where the server was started. You can stop the MobiLink server in the following ways:

- Click Shutdown on the MobiLink server window.
- Use Exit from the System tray context menu.
- Use the dbmlstop utility.

  ☞ For more information, see "MobiLink stop utility" on page 490.

Logging MobiLink synchronization server actions

Logging the actions that the server takes is particularly useful during the development process, and when troubleshooting. Verbose output is not recommended for normal operation of a production environment because it can slow performance.

Logging output to a file

Logging output is sent to the MobiLink synchronization server window. In addition, you can send the output to a log file using the -o option. The following command sends output to a log file named *mlsrv.log*.

```
dbmlsrv9 -o mlsrv.log -c ...
```

☞ For more information, see "-o option" on page 203.

Controlling the size of log files

You can control the size of log files, and specify what you want done when a file reaches its maximum size.

- With the -on option, you specify the size at which the log file is renamed with the extension .old, and a new file is started with the original name.
- With the -os option, you specify the size at which a new log file is started with a new name based on the date and a sequential number.
- With the -ot option, the contents of the log file are deleted before messages are sent to it.

☞ For more information, see

- “-on option” on page 204
- “-os option” on page 205
- “-ot option” on page 206

Controlling the amount of logging output

You can control the amount of output that is logged using the -v option.

☞ For more information, see “-v option” on page 211.

Controlling which errors are reported

☞ You can also control which warning messages are reported.

☞ For more information, see

- “-zw option” on page 222
- “-zwd option” on page 223
- “-zwe option” on page 223

MobiLink clients

Each remote database, together with its applications, is referred to as a **MobiLink client.** Two types of MobiLink client are supported:

♦ Adaptive Server Anywhere

♦ UltraLite

☞ For more information, see "Introducing MobiLink Clients" [*MobiLink Clients*, page 3].

The synchronization process

A **synchronization** is the process of bidirectional data exchange between the MobiLink client and synchronization server. During this process, the client must establish and maintain a connection to the synchronization server. If successful, the session leaves the remote and consolidated databases in a mutually consistent state.

The client normally initiates the synchronization process. It begins by establishing a connection to the MobiLink synchronization server.

The upload stream and the download stream

To upload rows, MobiLink clients prepare and send an **upload stream** that contains a list of all the rows that have been updated, inserted, or deleted on the MobiLink client since the last synchronization. Similarly, to download rows, the MobiLink synchronization server prepares and sends a **download stream** that contains a list of inserts, updates, and deletes.

1. **Upload stream** The MobiLink client automatically keeps track of which rows in the remote database have been inserted, updated, or deleted since the previous successful synchronization. Once the connection is established, the MobiLink client uploads a list of all these changes to the synchronization server.

   The upload stream consists of a set of new and old row values for each row modified in the remote database. If a row has been updated or deleted, the old values are those that were present immediately following the last successful synchronization. If a row has been inserted or updated, the new values are the current row values. No intermediate values are sent, even if the row was modified several times before arriving at its current state.

   The MobiLink synchronization server receives the upload stream and applies the changes to the consolidated database. It normally applies all the changes in a single transaction. When it has finished, the MobiLink synchronization server commits the transaction.

   Note
   MobiLink operates using the ODBC isolation level SQL_TXN_-READ_COMMITTED as the default isolation level for the consolidated database. MobiLink does so because repeatable reads are required for conflict detection purposes. If you have no conflict detection scripts or if you want to select an isolation level more suited to your needs, you can set this level in your begin_connection script.

2. **Download stream** The MobiLink synchronization server compiles a list of rows to be inserted, updated, or deleted on the MobiLink client, using synchronization logic that you create. It downloads these rows to

the MobiLink client. To compile this list, the MobiLink synchronization server opens a new transaction on the consolidated database.

The MobiLink client receives the download stream. It takes the arrival of this stream as confirmation that the consolidated database has successfully applied all uploaded changes. It will then ensure these changes are not sent to the consolidated database again.

Next, the MobiLink client automatically processes the download stream, deleting old rows, inserting new rows, and updating rows that have changed. It applies all these changes in a single transaction in the remote database. When finished, it commits the transaction.

3. **Optional download acknowledgement** The MobiLink client optionally sends a short confirmation message to the MobiLink synchronization server.

   The MobiLink synchronization server receives the confirmation message. This message tells the synchronization server that the client has received and processed all downloaded changes. In response, it commits the download transaction begun in step 2.

   ☞ For more information about the SendDownloadAck extended option, see "SendDownloadACK (sa) extended option" [*MobiLink Clients*, page 131] and "Send Download Acknowledgement synchronization parameter" [*MobiLink Clients*, page 331].

During MobiLink synchronization, there are few distinct exchanges of information. The client builds and uploads the entire upload stream. In response, the synchronization server builds and downloads the entire download stream. Limiting the chattiness of the protocol is especially important when communication is slower and has higher latency, as is the case when using telephone lines or public wireless networks.

MobiLink events

When the MobiLink client initiates a synchronization, a number of synchronization events occur. At the occurrence of an event, MobiLink looks for a script to match the synchronization event. This script contains your instructions outlining what you want done. The basic sequence is:

Event occurs ➤ *Script is invoked* (if it exists)

☞ For more information about synchronization events and scripts, see:

♦ "Synchronization Events" on page 319

♦ "Writing Synchronization Scripts" on page 227

MobiLink scripts

Whenever an event occurs, the MobiLink synchronization server executes the associated script if you have created one. If no script exists, the next event in the sequence occurs.

Following are some typical synchronization scripts for tables.

Event	Script
upload_insert	`INSERT INTO emp (emp_id,emp_name) VALUES (?,?)`
upload_delete	`DELETE FROM emp WHERE emp_id=?`
upload_update	`UPDATE emp SET emp_name=? WHERE emp_id=?`
upload_old_row_insert	`INSERT INTO old_emp (emp_id,emp_name) VALUES (?,?)`
upload_new_row_insert	`INSERT INTO new_emp (emp_id,emp_name) VALUES (?,?)`
upload_fetch	`SELECT id, name, size, quantity, unit_price FROM Product WHERE id=?`

The first event, upload_insert, triggers the running of the upload_insert script, which inserts the emp_id and emp_name into the emp table. In like fashion, the upload_delete and upload_update tables will perform similar functions for delete and update actions on the same emp table.

The download script uses a cursor. Following is an example of a download_cursor script:

```
SELECT emp_id, emp_name
   FROM emp
   WHERE emp_name = ?
```

The download_cursor acquires data from the emp table for a specified emp_name.

COMMIT or ROLLBACK statements within scripts alter the transactional nature of the synchronization steps. If you use them, you cannot guarantee the integrity of your data in the event of a failure. There should be no implicit or explicit commit or rollback in your synchronization scripts or the procedures or triggers that are called from your synchronization scripts.

Scripts can be written in SQL, Java, or .NET

You can write scripts using the native SQL dialect of your consolidated database, or using Java or .NET synchronization logic. Java and .NET synchronization logic allow you to write code, invoked by the MobiLink synchronization server, to connect to a database, manipulate variables, and create user-defined procedures that can work with MobiLink and any supported relational database. There is a MobiLink Java API and a MobiLink .NET API that have routines to suit the needs of synchronization.

☞ For more information about programming synchronization logic, see "Options for writing synchronization logic" on page 25.

☞ For information about DBMS-dependent scripting, such as scripting for Oracle, MS SQL Server, IBM DB2 or Adaptive Server Enterprise databases, see "DBMS-dependent synchronization scripts" on page 34.

Storing scripts

SQL scripts are stored in system tables in the consolidated database. For Java and .NET, pointers to other locations are stored in the consolidated database. You can add all kinds of scripts to a consolidated database in two ways:

- ♦ By using stored procedures that are installed along with the MobiLink system tables when you create a consolidated database.
- ♦ By using Sybase Central.

☞ For more information, see "Adding and deleting scripts in your consolidated database" on page 241.

Stored procedures

MobiLink stored procedures are used for programmatic conflict resolution, adding scripts, user authentication, and other customization procedures.

☞ For more information about using MobiLink stored procedures for customization, see "Stored Procedures" on page 479.

Other means to gain procedural control are commonly used with databases that don't have a defined procedural language. For example, with databases that do not permit user-defined procedures, such as IBM's DB2, Java procedures may be employed to act as MobiLink stored procedures.

☞ For more information about writing scripts using Java or .NET synchronization logic, see "Writing Synchronization Scripts in Java" on page 255 and "Writing Synchronization Scripts in .NET" on page 281.

For Adaptive Server Anywhere clients, you can use stored procedures called **client event hook procedures**, which are held on the remote database. A variety of event hook procedures are available for you to insert your own logic into the MobiLink synchronization process.

☞ For more information, see "Dbmlsync Client Event Hooks" [*MobiLink Clients*, page 175].

Transactions in the synchronization process

The MobiLink synchronization server incorporates changes uploaded from each MobiLink client into the consolidated database in one transaction. The MobiLink synchronization server commits these changes once it has completed inserting new rows, deleting old rows, making updates, and resolving any conflicts.

The MobiLink synchronization server prepares the download stream, including all deletes, inserts, and updates, using another transaction. Depending on the SendDownloadAck setting, it does not commit this transaction until it receives a positive confirmation from the MobiLink client. If the client confirms a successful download, the MobiLink synchronization server commits the download transaction. If the application encounters problems or cannot reply, the MobiLink synchronization server instead rolls back the download transaction.

Do not commit or roll back transactions within a script
COMMIT or ROLLBACK statements within scripts alter the transactional nature of the synchronization steps. If you use them, you cannot guarantee the integrity of your data in the event of a failure. There should be no implicit or explicit commit or rollback in your synchronization scripts or the procedures or triggers that are called from your synchronization scripts.

Tracking downloaded information

The primary role of the download transaction is to select rows in the consolidated database. If the download fails being sent to the remote, the remote will upload the same timestamp over again, and no data will be lost.

The MobiLink synchronization server uses two other transactions, one at the beginning of synchronization, and one at the end. These transactions allow you to record information regarding each synchronization and its duration. Thus, you can record statistics about attempted synchronizations, successful synchronizations, and the duration of synchronizations. Since data is committed at various points in the process, these transactions also let you commit data useful when analyzing failed synchronization attempts.

Similarly, the MobiLink client processes information in the download stream in *one transaction*. Rows are inserted, updated, and deleted to bring the remote database up to date with the consolidated data.

How synchronization failure is handled

MobiLink synchronization is fault tolerant. For example, if a communication link fails during synchronization, both the remote database and the consolidated database are left in a consistent state.

On the client, failure is indicated by a return code. For example, in an embedded SQL UltraLite application, the SQLCode is set to SQLE_COMMUNICATION_ERROR when ULSynchronize returns.

There are three cases that are handled in different ways:

- **Failure during upload** If the failure occurs while building or applying the upload stream, the remote database is left in exactly the same state as at the start of synchronization. At the server, any part of the upload stream that has been applied will be rolled back.

- **Failure between upload and download** If the failure occurs once the upload stream is complete, but before the MobiLink client receives the download stream, the client cannot be certain whether the uploaded changes were successfully applied to the consolidated database. The upload stream might be fully applied and committed, or the failure may have occurred before the server applied the entire upload stream. The MobiLink synchronization server automatically rolls back incomplete transactions in the consolidated database.

  The MobiLink client maintains a record of all uploaded changes in case they must be sent again. The next time the client synchronizes, it requests the state of the previous upload stream before building the new upload stream. If the previous upload was not committed, the new upload stream contains all changes from the previous upload stream.

- **Failure during download** If the failure occurs in the remote device while applying the download stream, any part of the download that has been applied is rolled back and the remote database is left in the same state as before the download. The MobiLink synchronization server automatically rolls back the download transaction in the consolidated database.

In all cases where failure may occur, no data is lost. The MobiLink server and the MobiLink client manage this for you. The developer/user need not worry about maintaining consistent data in their application.

How the upload stream is processed

When the MobiLink synchronization server receives an upload stream from a MobiLink client, the entire upload stream is stored until the synchronization is complete. This is done for three purposes.

- **Deadlock** When an upload stream is being applied to the consolidated database, it may encounter deadlock due to concurrency with other transactions. These transactions might be upload transactions from other MobiLink synchronization server database connections, or transactions from other applications using the consolidated database. When an upload transaction is deadlocked, it is rolled back and the MobiLink synchronization server automatically starts applying the upload stream from the beginning again.

  Performance tip
  It is important to write your synchronization scripts to avoid contention as much as possible. Contention has a significant impact on performance when multiple users are synchronizing simultaneously.

- **Filtering download rows** The most common technique for determining rows to download is to download rows that have been modified since the previous download. When synchronizing, the upload precedes the download. Any rows inserted or updated during the upload will be rows that have been modified since the previous download.

  It would be difficult to write a download_cursor script that omits from the download stream rows that were sent as part of the upload. For this reason, the MobiLink synchronization server automatically removes these rows from the download stream. When a row is being added to the download stream, the MobiLink synchronization server locates the row in the upload stream and omits the row from the download stream when it is found to be the same.

- **Processing deletes after inserts and updates** The upload stream is applied to the consolidated database in an order that avoids referential integrity violations. The upload stream is formatted so all operations (inserts, updates, and deletes) for a single table are grouped together. The tables in the upload stream are ordered based on foreign key relationships. All tables in the remote database that are referenced by another table in the remote database will be in the upload stream before the referencing table.

  For example, if table A and table C both have foreign keys that reference a primary key column in B, then table B rows are uploaded first.

When the upload stream is applied to the consolidated database, the inserts and updates are applied in the order they appear in the upload stream. When an inserted or updated row references a newly inserted row, this ensures the referenced row will be inserted before the referencing row. Deletes are applied in the opposite order after all inserts and updates have been applied. When a row being deleted references another row that is also being deleted, this order of operations ensures the referencing row is deleted before the referenced row is deleted.

Referential integrity and synchronization

All MobiLink clients enforce referential integrity when they incorporate the download stream into the remote database.

Rather than failing the download transaction, the MobiLink client automatically deletes all rows that violate referential integrity.

This feature affords you these key benefits.

- Protection from mistakes in your synchronization scripts. Given the flexibility of the scripts, it is possible to accidentally download rows that would break the integrity of the remote database. The MobiLink client automatically maintains referential integrity without requiring intervention.

- You can use this referential integrity mechanism to delete information from a remote database efficiently. By only sending a delete to a parent record, the MobiLink client will remove all the child records automatically for you. This can greatly reduce the amount of traffic MobiLink must send to the remote database.

MobiLink clients provide notification if they have to explicitly delete rows to maintain referential integrity.

- For Adaptive Server Anywhere clients, dbmlsync writes an entry in a log.

- For UltraLite clients, a warning SQLE_ROW_DELETED_TO_-MAINTAIN_REFERENTIAL_INTEGRITY is raised. This warning takes a parameter, which is the table name. The warning is raised on every row that is deleted to maintain referential integrity. Your application can ignore the warnings if you wish synchronization to just proceed. If you wish to explicitly handle the warnings, you can use the error callback function to trap them and, for example, count the number of rows deleted.

  If you wish to fail synchronization when the warning is raised, you must implement a synchronization observer and then signal the observer (perhaps through a global variable) from the error callback function to fail the synchronization on the next call to the observer.

Referential integrity checked at the end of the transaction

The MobiLink client incorporates changes from the download stream in a single transaction. To offer more flexibility, referential integrity checking occurs at the end of this transaction. Because checking is delayed, the database may temporarily pass through states where referential integrity is violated as rows are inserted, updated, and deleted, but the rows that violate referential integrity are automatically removed before the download is committed.

Errors are avoided

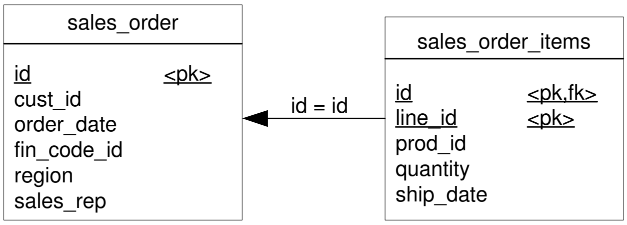
The MobiLink client resolves referential integrity violations automatically. This feature minimizes administration requirements. It also prevents an error in a synchronization script from disabling an MobiLink client. UltraLite clients raise a warning when a row is deleted to maintian referential integrity, which your application can handle or ignore.

An efficient way to delete rows

You can exploit the automatic referential integrity mechanism of MobiLink clients to delete large quantities of information in a very efficient manner. If your MobiLink client contains a primary row, and other rows that reference it, you can delete all the referencing rows simply by synchronizing a delete of the primary row.

Example

Suppose that an UltraLite sales application contains, among others, the following two tables. One table contains sales orders. Another table contains items that were sold in each order. They have the following relationship.



If you use the download_delete_cursor for the sales_order table to delete an order, the automatic referential integrity mechanism automatically deletes all rows in the sales_order_items table that point to the deleted sales order.

This arrangement has the following advantages.

- ♦ You do not require a sales_order_items script because rows from this table will be deleted automatically.
- ♦ The efficiency of synchronization is improved. You need not download rows to delete from the sales_order_item table. If each sales order contains many items, the performance improves because the download stream is now smaller. This technique is particularly valuable when using slow communication methods.

Upload-only and download-only synchronization

For both Adaptive Server Anywhere and UltraLite remote databases, you can choose to do a full synchronization, or you can perform only an upload or download.

Adaptive Server Anywhere remote databases

In Adaptive Server Anywhere remote databases, you perform upload-only synchronization using the dbmlsync option -ds or the extended option DownloadOnly.

☞ For more information, see "-ds option" [*MobiLink Clients,* page 104] and "DownloadOnly (ds) extended option" [*MobiLink Clients,* page 115].

In Adaptive Server Anywhere remote databases, you perform download-only synchronization using the dbmlsync option -uo or the extended option UploadOnly.

☞ For more information, see "-uo option" [*MobiLink Clients,* page 149] or "UploadOnly (uo) extended option" [*MobiLink Clients,* page 133].

UltraLite remote databases

In UltraLite remote databases, you perform download-only synchronization using the Download Only synchronization parameter.

☞ For more information, see "Download Only synchronization parameter" [*MobiLink Clients,* page 320].

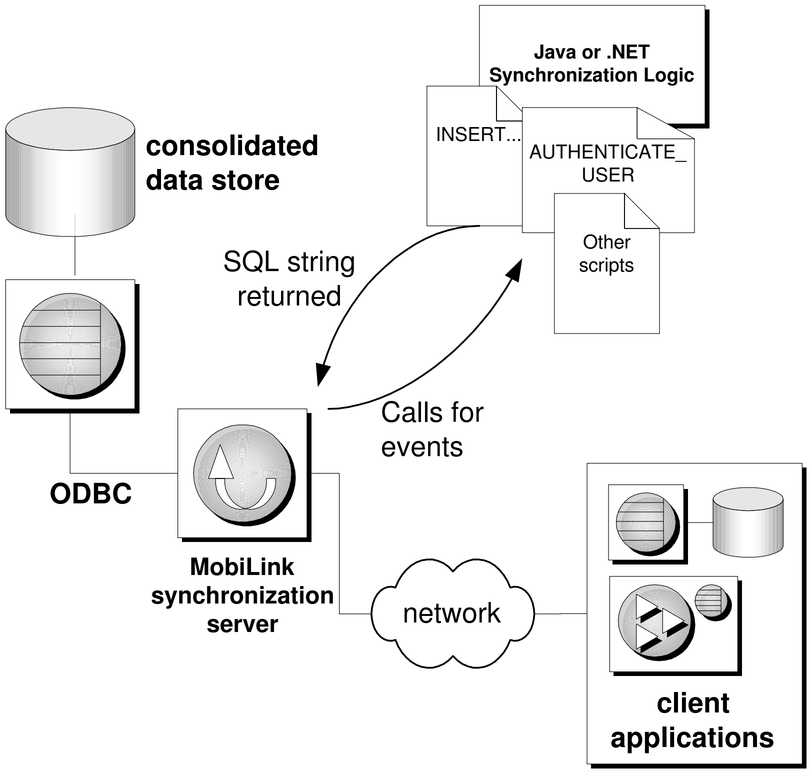
In UltraLite remote databases, you perform upload-only synchronization using the Upload Only synchronization parameter.

☞ For more information, see "Upload Only synchronization parameter" [*MobiLink Clients,* page 337].

Options for writing synchronization logic

MobiLink synchronization scripts can be written in SQL, in Java, or in .NET programming languages. Java or .NET are a good choice whenever your design is restricted by the limitations of the SQL language or by the capabilities of your database-management system, of if you want DBMS-independent synchronization logic.

Program synchronization logic can function just as SQL logic functions, as shown in the figure below. The MobiLink synchronization server can make calls to Java or .NET methods on the occurrence of MobiLink events just as it can access SQL scripts on the occurrence of MobiLink events. However, the upload and download streams are not directly accessible from Java or .NET synchronization logic, where a SQL string must be returned to MobiLink.



SQL synchronization logic is restricted to the procedural language capabilities of your consolidated database. SQL languages are unlikely to offer all the programming logic given by Java or .NET programming languages. You might want to use Java or .NET synchronization logic when your SQL logic is limited, when you need to perform operations across database platforms, and when you need portability across RDBMSs and operating systems. Following are some scenarios where you might want to consider writing scripts in Java or .NET.

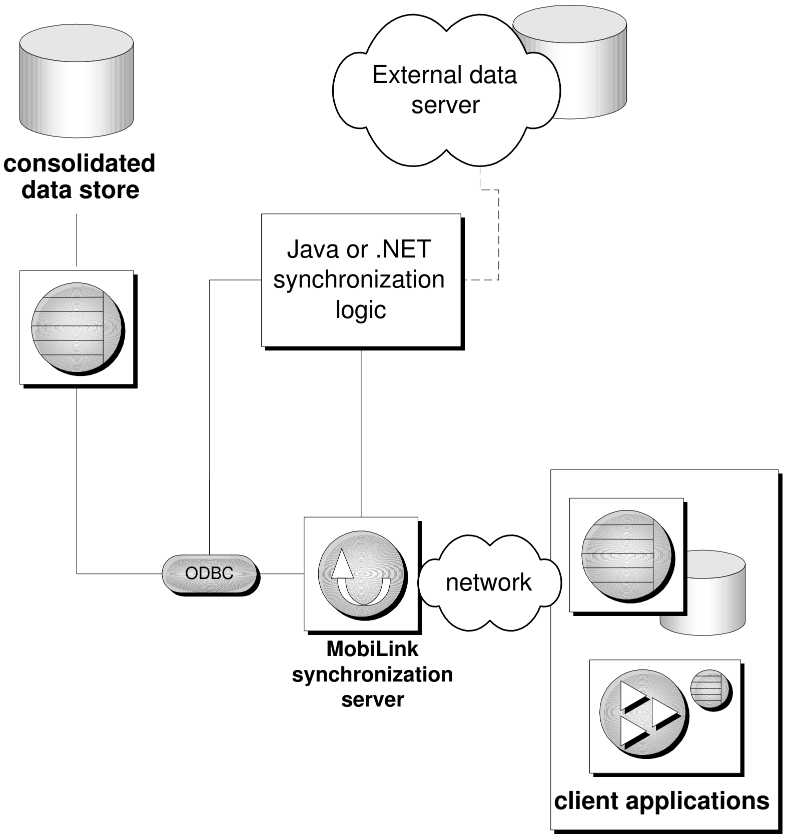
♦ A user authentication procedure can be written in Java or .NET that inserts the user ID of a MobiLink user into a table on the consolidated database for audit purposes.

♦ If your database lacks the ability to handle variables, you can create a variable in Java or .NET that persists throughout your connection or synchronization.

♦ If your database lacks the ability to make user-defined stored procedures, you can make a method in Java or .NET that can perform the needed functionality.

♦ If your program calls for contacting an external server midway through a synchronization event, you can use Java or .NET synchronization logic to perform actions triggered by synchronization events. Java and .NET synchronization logic can be shared across multiple connections.

♦ With Java and .NET synchronization logic, you can use MobiLink to access data from application servers, Web servers, and files. You can use JDBC or iAnywhere classes in your synchronization logic to access data in relational databases other than the consolidated database. For example, an external server can be used to validate a user ID and password. The figure below shows the links between Java or .NET synchronization logic and both a consolidated database and a second data server.



MobiLink APIs

With Java and .NET synchronization logic, you have access to a MobiLink API. The MobiLink APIs are sets of classes and interfaces for MobiLink synchronization. There are two MobiLink APIs: Java and .NET.

The MobiLink Java API offers you:

♦ Access to the existing ODBC connection as a JDBC connection.

♦ The ability to create new JDBC connections to perform commits or connects outside the current synchronization connection. For example, you can use this for error logging.

♦ The ability to write and debug Java code before it is executed by the MobiLink server. SQL development environments for many database management systems are relatively primitive compared to those available for Java applications.

- ♦ Code that runs inside the Java virtual machine and allows access to all Java libraries and Java Native Interface calls.

☞ For more information, see "MobiLink Java API Reference" on page 273.

The MobiLink .NET API offers you:

- ♦ Access to the existing ODBC connection using iAnywhere classes that call ODBC from .NET.
- ♦ Code that runs inside the .NET Common Language Runtime (CLR) and allows access to all .NET libraries and unmanaged calls.

☞ For more information, see "MobiLink .NET API Reference" on page 303.

Further reading

☞ For more information about your options for writing synchronization scripts, see

- ♦ "Writing Synchronization Scripts" on page 227
- ♦ "Synchronization Techniques" on page 45
- ♦ "Writing Synchronization Scripts in Java" on page 255
- ♦ "Writing Synchronization Scripts in .NET" on page 281

Security

There are several aspects to securing data throughout a widely distributed system such as a MobiLink installation:

- ♦ **Protecting data in the consolidated database** Data in the consolidated database can be protected using the DBMS user authentication system and other security features.

  ☞ For more information, see your DBMS documentation. If you are using an Adaptive Server Anywhere consolidated database, see "Keeping Your Data Secure" [*SQL Anywhere Studio Security Guide*, page 3].

- ♦ **Protecting data in the remote databases** If you are using Adaptive Server Anywhere remote databases, the data can be protected using Adaptive Server Anywhere security features. By default, these are designed to prevent unauthorized access through client/server communications, but not to be proof against a serious attempt to extract information directly from the database file.

  Files on the client are protected by the security features of the client operating system.

  ☞ If you are using an Adaptive Server Anywhere remote database, see "Keeping Your Data Secure" [*SQL Anywhere Studio Security Guide*, page 3].

- ♦ **Protecting data during synchronization** Communication from MobiLink clients to MobiLink synchronization servers can be protected by the MobiLink transport layer security features.

  ☞ For more information, see "MobiLink Transport-Layer Security" on page 165.

- ♦ **Protecting the synchronization system from unauthorized users** MobiLink synchronization can be secured by a password-based user authentication system. This mechanism prevents unauthorized users from synchronizing data.

  ☞ For more information, see "Authenticating MobiLink Users" [*MobiLink Clients*, page 9].

CHAPTER 3

MobiLink Consolidated Databases

About this chapter

This chapter describes how to set up and use your consolidated database.

Contents

Introduction

Your consolidated database can be one of the following ODBC-compliant databases: Adaptive Server Anywhere, Adaptive Server Enterprise, Oracle, IBM DB2, and Microsoft SQL Server. You can use synchronization scripts to exploit the features of your particular consolidated server.

Synchronization scripts are associated with the consolidated database. SQL scripts are stored in the consolidated database, and Java and .NET scripts are referenced.

☞ For information about setting up each type of database as a consolidated database, see "Setting up a consolidated database" on page 33.

☞ For information about writing synchronization scripts for particular consolidated databases, see "DBMS-dependent synchronization scripts" on page 34.

How remote tables relate to consolidated tables

Synchronization designs can specify mappings between tables and rows in the remote database with tables and rows in the consolidated database.

Arbitrary relationships permitted

Tables in a remote database need not be identical to those in the consolidated database. Synchronized data in one remote application table can be distributed between columns in different tables, and even between tables in different consolidated databases. You specify these relationships using synchronization scripts.

Direct relationships are simple

You can often simplify your design using a table structure in the remote database that is a subset of that in the consolidated database. Using this method, every table in the remote database exists in the consolidated database. Corresponding tables have the same structure and foreign key relationships as those in the consolidated database.

Tables in the consolidated database will frequently contain extra columns that are not synchronized. Indeed, extra columns can aid synchronization. For example, a timestamp column can identify new or updated rows in the consolidated database. In other cases, extra columns or tables in the consolidated database may hold information that is not required at remote sites.

Setting up a consolidated database

Setup scripts

To set up a database so that it can be used as a MobiLink consolidated database, you must run a **setup script** that installs MobiLink system tables and stored procedures. The exception is Adaptive Server Anywhere databases, which are preconfigured with the appropriate system tables and stored procedures. For instructions on how to run the setup scripts, see the sections below for each supported RDBMS.

☞ For more information about MobiLink system tables, see "MobiLink System Tables" on page 501.

☞ For more information about stored procedures, see "Stored Procedures" on page 479.

ODBC connection

In addition, the MobiLink synchronization server needs an ODBC connection to your consolidated database. You must configure the appropriate ODBC driver for your server and create an ODBC data source for the database on the computer where your MobiLink synchronization server is running.

☞ For a summary of supported ODBC drivers, see "iAnywhere Solutions ODBC Drivers" on page 547.

☞ For updated information and complete functional specifications, see *http://www.ianywhere.com/developer/technotes/odbc_mobilink.html*.

☞ For information about configuring ODBC drivers for MobiLink consolidated databases, see "Introduction to iAnywhere Solutions ODBC Drivers" [*ODBC Drivers for MobiLink and Remote Data Access*, page 1].

☞ For specific information about each type of consolidated database, see the appropriate section:

- ♦ "Adaptive Server Anywhere consolidated database" on page 36
- ♦ "Sybase Adaptive Server Enterprise consolidated database" on page 37
- ♦ "Oracle consolidated database" on page 39
- ♦ "IBM DB2 consolidated database" on page 40
- ♦ "Microsoft SQL Server consolidated database" on page 43

DBMS-dependent synchronization scripts

MobiLink uses synchronization scripts to provide flexibility in the rules you use to synchronize data. The scripts define:

- ♦ How data uploaded from the remote database is to be applied to the consolidated database.
- ♦ What data should be downloaded from the consolidated database.

☞ For more information about writing synchronization scripts, see "Writing Synchronization Scripts" on page 227.

☞ For a complete list of events you can write scripts for, see "Synchronization Events" on page 319.

Some aspects of scripts depend on the DBMS you are using. A number of factors determine the kind of scripting needed for your synchronization with your ODBC compliant database, and these factors include, but are not limited to:

- ♦ User defined procedures
- ♦ Session-wide connection variables

  Session-wide variables provide a useful means to store values persisting through a synchronization. For example, in the begin_synchronization script store the user name into a session-wide variable variable called UserName. In subsequent scripts, refer to UserName as many times as you want.

- ♦ Autoincrement methods

  If supported, you can use autoincrement to maintain unique primary key values in a MobiLink environment. For more information, see "Maintaining unique primary keys" on page 56.

☞ For specific information about each type of consolidated database, see the appropriate section:

- ♦ "Adaptive Server Anywhere consolidated database" on page 36
- ♦ "Sybase Adaptive Server Enterprise consolidated database" on page 37
- ♦ "Oracle consolidated database" on page 39
- ♦ "IBM DB2 consolidated database" on page 40
- ♦ "Microsoft SQL Server consolidated database" on page 43

.NET and Java synchronization scripts

One strategy for using MobiLink with supported databases is to write your table scripts and synchronization logic in the DBMS version of the SQL language. Another strategy for using MobiLink with any supported consolidated database uses Java or .NET synchronization logic. When you use Java or .NET synchronization logic you can hold session-wide variables and create user-defined procedures.

☞ For information about Java synchronization logic, see "Writing Java synchronization logic" on page 259.

☞ For information about .NET synchronization logic, see "Writing Synchronization Scripts in .NET" on page 281.

Invoking procedures from scripts

Some databases, such as Microsoft SQL Server, require that procedure calls with parameters be written using the ODBC syntax.

```
{ CALL procedure_name( ?, ?, ... ) }
```

You can pass return values by defining the parameters as OUT or INOUT in the procedure definition.

CHAR columns

In many other DBMSs, CHAR data types are fixed length and blank-padded to the full length of the string. In MobiLink remote databases (Adaptive Server Anywhere or UltraLite) CHAR is the same as VARCHAR: values are not blank-padded to a fixed width. The dbmlsrv9 -b command line option can be used to remove trailing blanks from strings during synchronization. This option is important for string comparisons used to detect conflicts.

☞ For more information, see "-b option" on page 195.

Data conversion

For information about the conversion of data that must take place when a MobiLink synchronization server communicates with a consolidated database that was not made with Adaptive Server Anywhere, see "DataType Conversions" on page 533.

Adaptive Server Anywhere consolidated database

Adaptive Server Anywhere databases are automatically configured so that they can be used as a MobiLink consolidated database without running a setup script.

A setup script is provided for Adaptive Server Anywhere databases in case you want to examine source code. For example, it includes source code for the ml_add_connection_script stored procedure. This setup script is called *syncasa.sql* and it is located in the *scripts* subdirectory of your SQL Anywhere installation.

Setting up the ODBC driver

You must set up an ODBC DSN for your Adaptive Server Anywhere consolidated database. The ODBC driver for Adaptive Server Anywhere is installed with SQL Anywhere Studio.

☞ For information about the Adaptive Server Anywhere ODBC driver, see "Working with ODBC data sources" [*ASA Database Administration Guide*, page 53].

Sybase Adaptive Server Enterprise consolidated database

To set up Adaptive Server Enterprise version 12.5 or later to work as a MobiLink consolidated database, run the *syncase125.sql* setup script, located in the *MobiLink\setup* subdirectory of your SQL Anywhere installation. For versions prior to 12.5, run *syncase.sql* from the same location.

To ensure that the MobiLink administrator has adequate permissions to change the MobiLink system tables (required for adding scripts and other things), the user who will need access to those tables should run the setup script.

ODBC driver

You must set up an ODBC DSN for your Adaptive Server Enterprise consolidated database. SQL Anywhere Studio includes an iAnywhere Solutions ODBC driver for Adaptive Server Enterprise. You must configure this driver to work with MobiLink.

☞ For more information, see "iAnywhere Solutions ODBC Driver for Sybase Adaptive Server Enterprise" [*ODBC Drivers for MobiLink and Remote Data Access*, page 11].

Adaptive Server Enterprise issues

- **Column sizes** To download BLOB data from an Adaptive Server Enterprise consolidated database, you need to set an ODBC driver connection option to allow column sizes greater than 4096 bytes. To do this, use the ODBC driver connection option called StaticCursorLongColBuffLen. For example,

  ```
  dbmlsrv9 -c "...;StaticCursorLongColBuffLen=number"
  ```

  where *number* is in bytes, and is larger than the largest expected BLOB.

  Note that using this option consumes significantly more disk space on the computer that runs the MobiLink synchronization server.

- **Numeric primary key values** In addition, the MobiLink synchronization server requires that primary key values of type numeric or decimal be explicitly converted to their types under Adaptive Server Enterprise.

  You must add an explicit conversion to the numeric parameters in the script as displayed in the following examples.

  ```
  SELECT ...
  WHERE numeric_col = CONVERT( NUMERIC, ? )
  ...
  ```

  The above statement explicitly converts the first parameter to type

NUMERIC.

```
SELECT ...
WHERE decimal_col = CONVERT( DECIMAL(10,8), ? )
...
```

The above statement explicitly converts the first parameter to type DECIMAL (10,8).

- **CHAR columns** In Adaptive Server Enterprise, CHAR data types are fixed length and blank-padded to the full length of the string. In MobiLink remote databases (Adaptive Server Anywhere or UltraLite) CHAR is the same as VARCHAR: values are not blank-padded to a fixed width. The dbmlsrv9 -b command line option can be used to remove trailing blanks from strings during synchronization. This option is important for string comparisons used to detect conflicts.

  ☞ For more information, see "-b option" on page 195.

- **Data type mapping** For details of how Adaptive Server Anywhere data types are mapped to Adaptive Server Enterprise data types, see "Adaptive Server Enterprise data mapping" on page 534.

Oracle consolidated database

To set up Oracle to work as a MobiLink consolidated database, run the *syncora.sql* setup script, located in the *MobiLink\setup* subdirectory of your SQL Anywhere installation.

To ensure that the MobiLink administrator has adequate permissions to change the MobiLink system tables (required for adding scripts and other things), the user who will need access to those tables should run the setup script.

ODBC driver

You must set up an ODBC DSN for your Oracle consolidated database. SQL Anywhere Studio includes an iAnywhere Solutions ODBC driver for Oracle. You must configure this driver to work with MobiLink.

☞ For more information, see "iAnywhere Solutions ODBC Driver for Oracle" [*ODBC Drivers for MobiLink and Remote Data Access*, page 31].

Oracle issues

- **Session-wide variables** Oracle does not provide session-wide variables. You can store session-wide information in variables within Oracle packages. Oracle packages allow variables to be created, modified and destroyed and these variables may last as long as the Oracle package is current.

- **Autoincrement methods** To maintain primary key uniqueness, you can use an Oracle sequence to generate a list of keys similar to that of an autoincrement field. The CustDB sample database provides coding examples, which can be found in *Samples\MobiLink\CustDB\custora.sql*. Unlike autoincrement, however, you must explicitly reference the sequence. Autoincrement inserts a column value automatically if the column is not referenced in an INSERT statement.

  ☞ For an example of using an Oracle sequence, see "Tutorial: Using MobiLink with an Oracle 8i Consolidated Database" [*MobiLink Tutorials*, page 39].

- **CHAR columns** In Oracle, CHAR data types are fixed length and blank-padded to the full length of the string. In MobiLink remote databases (Adaptive Server Anywhere or UltraLite) CHAR is the same as VARCHAR: values are not blank-padded to a fixed width. The dbmlsrv9 -b command line option can be used to remove trailing blanks from strings during synchronization. This option is important for string comparisons used to detect conflicts.

  ☞ For more information, see "-b option" on page 195.

- **Data type mapping** For details of how Adaptive Server Anywhere data types are mapped to Oracle data types, see "Oracle data mapping" on page 538.

IBM DB2 consolidated database

To set up IBM DB2 UDB to work as a MobiLink consolidated database, run the *syncdb2long.sql* setup script, located in the *MobiLink\setup* subdirectory of your SQL Anywhere installation. However, prior to running the script you must complete the following steps.

To install MobiLink system tables using the setup script, an IBM DB2 tablespace must use a minimum of 8 Kb pages. If a tablespace does not use 8 Kb pages, complete the following steps.

- ♦ Verify that at least one of your buffer pools has 8 Kb pages. If not, create a buffer pool with 8 Kb pages.
- ♦ Create a new tablespace and temporary tablespace that use the buffer pool with 8 Kb pages. For more information, consult your DB2 documentation.

To ensure that the MobiLink administrator has adequate permissions to change the MobiLink system tables (required for adding scripts and other things), the user who will need access to those tables should run the setup script.

The *syncdb2long.sql* script contains a default connection statement, `connect to DB2Database`. You should make a copy of the script and alter this line to be appropriate for your installation. The syntax of the line must be:

```
connect to DB2Database user userid using password ~
```

where *DB2Database*, *userid*, and *password* are names you provide.

The *syncdb2long.sql* script uses the tilde character (~) as a command delimiter. You can run the scripts as follows:

```
db2 -c -ec -td~ +s -v -f syncdb2long.sql
```

In order for DB2 to use the stored procedures defined in *syncdb2long.sql*, you must copy the *syncdb2long_version* Java and class files located in the *MobiLink\setup* subdirectory of your SQL Anywhere installation to the *FUNCTION* subdirectory of your DB2 installation.

ODBC driver

To set up an ODBC DSN for your DB2 consolidated database, use the iAnywhere Solutions DB2 Wire Protocol driver.

☞ For more information, see "iAnywhere Solutions ODBC Driver for DB2" [*ODBC Drivers for MobiLink and Remote Data Access*, page 51].

DB2 issues

- ♦ **Tablespace capacity** A tablespace and temporary tablespace of any DB2 database that you wish to use as a consolidated database must use 8 Kb pages.

  In addition, there are columns that require a LONG tablespace. If there is no default LONG tablespace, the creation statements for the tables containing these columns must be qualified appropriately, as in the following example.

  ```
  CREATE TABLE ... ( ... )
  IN tablespace
  LONG IN long-tablespace
  ```

  For an example using the sample application, see "The CustDB Sample Application" [*MobiLink Tutorials*, page 99].

- ♦ **Session-wide variables** Earlier versions of DB2 (prior to version 8) do not support session-wide variables. A convenient solution is to use a base table with columns for the MobiLink user name and other session data. The base table will have rows representing concurrent synchronizations.

- ♦ **User-defined procedures.** DB2 requires a C compiler to compile SQL procedures into an executable library (such as a DLL). The resulting DLL/shared library must be copied to a special directory on the server. Note that you can write stored procedures using several different languages, including C/C++ and Java, among others.

  ☞ For an example of Java as a procedural language for DB2, see the CustDB scripts in the files *Samples\MobiLink\CustDB\custdbq.sql* and *Samples\MobiLink\CustDB\custdbq.java.*

  ☞ For more information about Java and .NET synchronization scripts, see

  - "Options for writing synchronization logic" on page 25
  - "Writing Synchronization Scripts in Java" on page 255
  - "Writing Synchronization Scripts in .NET" on page 281

- ♦ **CHAR columns** In IBM DB2, CHAR data types are fixed length and blank-padded to the full length of the string. In MobiLink remote databases (Adaptive Server Anywhere or UltraLite) CHAR is the same as VARCHAR: values are not blank-padded to a fixed width. The dbmlsrv9 -b command line option can be used to remove trailing blanks from strings during synchronization. This option is important for string comparisons used to detect conflicts.

  ☞ For more information, see "-b option" on page 195.

- **Data type mapping** For details of how Adaptive Server Anywhere data types are mapped to DB2 data types, see "IBM DB2 data mapping" on page 536.

Microsoft SQL Server consolidated database

To set up Microsoft SQL Server to work as a MobiLink consolidated database, run the *syncmss.sql* setup script, located in the *MobiLink\setup* subdirectory of your SQL Anywhere installation.

To ensure that the MobiLink administrator has adequate permissions to change the MobiLink system tables (required for adding scripts and other things), the user who will need access to those tables should run the setup script.

ODBC driver

You must set up an ODBC DSN for your Microsoft SQL Server consolidated database. Unlike the other consolidated databases, iAnywhere Solutions does not provide an ODBC driver for Microsoft SQL Server. This is because the Microsoft SQL Server driver is freely available for download.

☞ For updated details, see *http://www.ianywhere.com/developer/technotes/odbc_mobilink.html*.

SQL Server issues

♦ **Procedure calls** Microsoft SQL Server requires that procedure calls with parameters be written using the ODBC syntax:

```
{ CALL procedure_name( ?, ?, ... ) }
```

♦ **CHAR columns** In Microsoft SQL Server, CHAR data types are fixed length and blank-padded to the full length of the string. In MobiLink remote databases (Adaptive Server Anywhere or UltraLite) CHAR is the same as VARCHAR: values are not blank-padded to a fixed width. The dbmlsrv9 -b command line option can be used to remove trailing blanks from strings during synchronization. This option is important for string comparisons used to detect conflicts.

☞ For more information, see "-b option" on page 195.

♦ **Data type mapping** For details of how Adaptive Server Anywhere data types are mapped to SQL Server data types, see "Microsoft SQL Server data mapping" on page 540.

CHAPTER 4

Synchronization Techniques

About this chapter

This chapter describes a variety of techniques that you can use to tackle common synchronization tasks encountered in MobiLink installations.

☞ There are sample applications that provide examples of the techniques that are described in this chapter. For more information, see "The Contact Sample Application" [*MobiLink Tutorials,* page 83], and "The CustDB Sample Application" [*MobiLink Tutorials,* page 99].

The techniques in this chapter are illustrated using SQL scripts. Many of the same techniques can be implemented in Java or .NET synchronization logic. For more information, see

- "Writing Synchronization Scripts in Java" on page 255
- "Writing Synchronization Scripts in .NET" on page 281

Contents

Introduction

The chapter “Writing Synchronization Scripts” on page 227 describes how to write simple synchronization scripts, store them in your database, and test that they are free of syntax errors.

Many useful synchronization features require not just one script, but a set of scripts working together. This chapter describes how to implement some common synchronization techniques. The examples describe SQL synchronization scripts. You can also use Java or .NET synchronization logic, although the upload and download events still require a knowledge of the SQL scripts.

Example

The timestamp-based synchronization of the Customer table used in the Contact sample application requires the following scripts:

- An **upload_insert** script to handle new rows added at remote databases at the consolidated database.
- An **upload_update** script to handle modifications made at remote databases at the consolidated database.
- An **upload_delete** script to handle rows deleted from remote databases at the consolidated database.
- A **download_cursor** script to download new and updated rows to remote databases.
- A **download_delete_cursor** script to download rows to be deleted from remote databases.

Development tips

Adding synchronization functionality to an application adds an added level of complexity to your application. The following tips may be useful.

- **Wait** If you try to add synchronization to a prototype application, it can be difficult to see which components are causing problems. This is particularly the case with UltraLite applications, where database and application are compiled together. When developing a prototype, temporarily hard code INSERT statements in your application to provide data for testing and demonstration purposes. Once your prototype is working correctly, enable synchronization and discard the temporary INSERT statements.
- **Go step-by-step** Start with straightforward synchronization techniques. Operations such as a simple upload or download require only one or two scripts. Once those are working correctly, you can introduce more advanced techniques, such as timestamps, primary key pools, and conflict resolution.

Following are some fundamental rules of MobiLink synchronization applications.

- Every table that is to be synchronized must have a primary key.
- Don’t update the values of primary keys.
- Primary keys must be unique across all synchronized databases.
- The MobiLink user name that identifies each remote database must be unique.

Timestamp-based synchronization

The timestamp method is the most useful general technique for efficient synchronization. The technique involves tracking the last time that each user synchronized, and using this information to control the rows downloaded to each remote database.

MobiLink maintains a timestamp value indicating when each MobiLink user last downloaded data. This value is called the **last download timestamp**. The last download timestamp is provided as a parameter to many events, and can be used in synchronization scripts.

If you are using an Adaptive Server Anywhere consolidated database and the column holding last modified information is of type DEFAULT TIMESTAMP, then the column should not be synchronized. If your remote databases require such a column, a different column name should be used. Otherwise, the default timestamp value may be overridden by the uploaded value, and will not contain the time that the row was last modified on the consolidated.

❖ To implement timestamp-based synchronization for a table

1. At the consolidated database, add a column that holds the most recent time the row was modified. The column is typically declared as follows:

DBMS	last modified column
Adaptive Server Anywhere	`timestamp DEFAULT timestamp`
Adaptive Server Enterprise	`datetime`
Microsoft SQL Server	`datetime`
Oracle	`date`
IBM DB2	`timestamp`

2. In scripts for the download_cursor and download_delete_cursor events, compare the first parameter to the value in the timestamp column.

Example

The following table declaration and scripts implement timestamp-based synchronization on the Customer table in the Contact sample:

- Table definition:

```
CREATE TABLE "DBA"."Customer"(
   "cust_id"  integer NOT NULL
                     DEFAULT GLOBAL AUTOINCREMENT,
   "name"  char(40) NOT NULL,
   "rep_id"  integer NOT NULL,
   "last_modified" timestamp NULL DEFAULT timestamp,
   "active"  bit NOT NULL,
   PRIMARY KEY ("cust_id") )
```

- download_delete_cursor script:

```
SELECT cust_id
FROM Customer JOIN SalesRep
ON Customer.rep_id = SalesRep.rep_id
WHERE Customer.last_modified >= ?
    AND ( SalesRep.ml_username != ?
          OR Customer.active = 0 )
```

- download_cursor script:

```
SELECT cust_id, Customer.name, Customer.rep_id
FROM Customer KEY JOIN SalesRep
WHERE Customer.last_modified >= ?
AND SalesRep.ml_username = ?
AND Customer.active = 1
```

☞ For more information, see "Synchronization logic source code" [*MobiLink Tutorials*, page 113], and "Synchronizing contacts in the Contact sample" [*MobiLink Tutorials*, page 93].

Snapshot synchronization

Timestamp-based synchronization is appropriate for most synchronizations. However, occasionally you may want to update a snapshot of your data.

Snapshot synchronization of a table is a complete download of all relevant rows in the table, even if they have been downloaded before. This is the simplest synchronization method, but can involve unnecessarily large data sets being exchanged, which can limit performance.

You can use snapshot synchronization for downloading all the rows of the table, or in conjunction with a partitioning of the rows as described in "Partitioning rows among remote databases" on page 52.

When to use snapshot synchronization

The snapshot method is typically most useful for tables that have both the following characteristics.

- **Relatively few rows** When there are few rows, the overhead associated with downloading all of them is small.
- **Rows that change frequently** When most rows in a table change frequently, there is little to be gained by explicitly excluding those that have not changed since the last synchronization.

A table that holds a list of exchange rates could be suited to this approach because there are relatively few currencies, but the rates of most change frequently. Depending on the nature of the business, a table that holds prices, a list of interest rates, or current news items could all be candidates.

❖ To implement snapshot-based synchronization

1. Leave the upload scripts undefined unless remote users update the values.
2. If the table may have rows deleted, write a download_delete_cursor script that deletes all the rows from the remote table, or at least all rows no longer required. Do not delete the rows from the consolidated database; rather, mark them for deletion. You must know the row values to delete them from the remote database.

   ☞ For more information, see "Writing download_delete_cursor scripts" on page 248.
3. Write a download_cursor script that selects all the rows you want to include in the remote table.

Deleting rows

Rather than deleting rows from the consolidated database, mark them for deletion. You must know the row values to delete them from the remote database. Select only unmarked rows in the download_cursor script and only marked rows in the download_delete_cursor script.

The download_delete_cursor script is executed before the download_cursor script. If a row is to be included in the download stream, you need not include a row with the same primary key in the delete list. When a downloaded row is received at the remote location, it replaces a preexisting row with the same primary key.

☞ For more information, see "Writing scripts to download rows" on page 246.

An alternative deletion technique

Rather than delete rows from the remote database using a download_cursor script, you can allow the remote application to delete the rows. For example, immediately following synchronization, you could allow the application to execute SQL statements that delete the unneeded rows.

Rows deleted by the application are ordinarily uploaded to the MobiLink synchronization server upon the next synchronization, but you can prevent this upload using the STOP SYNCHRONIZATION DELETE statement. For example,

```
STOP SYNCHRONIZATION DELETE;
DELETE FROM table-name
   WHERE expiry_date < CURRENT TIMESTAMP;
COMMIT;
START SYNCHRONIZATION DELETE;
```

☞ For more information about deleting rows, see "Writing download_delete_cursor scripts" on page 248.

Snapshot example

The ULProduct table in the sample application is maintained by snapshot synchronization. The table contains relatively few rows, and for this reason, there is little overhead in using snapshot synchronization.

1. There is no upload script. This reflects a business decision that products cannot be added at remote databases.

2. There is no download_delete_cursor, reflecting an assumption that products are not removed from the list.

3. The download_cursor script selects the product identifier, price, and name of every current product. If the product is pre-existing, the price in the remote table will be updated. If the product is new, a row will be inserted in the remote table.

   ```
   SELECT prod_id, price, prod_name
   FROM ULProduct
   ```

☞ For another example of snapshot synchronization in a table with very few rows, see "Synchronizing sales representatives in the Contact sample" [*MobiLink Tutorials*, page 91].

Partitioning rows among remote databases

Each MobiLink remote database can contain a different subset of the data in the consolidated database. Stated another way, you can write your scripts so that data is **partitioned** among remote databases.

The partitioning may be disjoint, or it may contain overlaps. For example, if each employee has their own set of customers, with no shared customers, the partitioning would be **disjoint**. If there are shared customers, who appear in more than one remote database, the partitioning contains **overlaps**.

Partitioning is implemented in the download_cursor and download_delete_cursor scripts for the table, which define the rows to be downloaded to the remote database. Each of these scripts takes a MobiLink user name as a parameter parameter. By defining your scripts using this parameter in the WHERE clause, each user gets the appropriate rows.

Disjoint partitioning

Partitioning is controlled by the download_cursor and download_delete_cursor scripts for each table involved in synchronization. These scripts take two parameters, a last download timestamp and the MobiLink user name supplied in the call to synchronize.

❖ To partition a table among remote databases

1. Include in the table definition a column containing the synchronization user name in the consolidated database. You need not download this column to remote databases.

2. Include a condition in the WHERE clause of the download_cursor and download_delete_cursor scripts requiring this column to match the script parameter.

   The script parameter is represented by a question mark in the script. The user name is the second parameter in the download_cursor script. For example, the following download_cursor script partitions the Contact table by employee ID.

   ```
   SELECT id, contact_name
   FROM Contact
   WHERE last_modified >= ?
   AND emp_id = ?
   ```

   ☞ For more information, see "download_cursor table event" on page 371, and "download_delete_cursor table event" on page 375.

Example

The primary key pool tables in the CustDB sample application are used to supply each remote database with its own set of primary key values. This technique is used to avoid duplicate primary keys, and is discussed in "Maintaining unique primary keys using key pools" on page 60.

A necessary feature of the method is that primary key-pool tables must be partitioned among remote databases in a disjoint fashion.

One key-pool table is ULCustomerIDPool, which holds primary key values for each user to use when they add customers. The table has three columns:

- **pool_cust_id** A primary key value for use in the ULCustomer table. This is the only column downloaded to the remote database.
- **pool_emp_id** The employee who owns this primary key.
- **last_modified** This table is maintained using the timestamp technique, based on the last_modified column.

  ☞ For information on timestamp synchronization, see "Timestamp-based synchronization" on page 48.

The download_cursor script for this table is as follows.

```
SELECT pool_cust_id
FROM ULCustomerIDPool
WHERE last_modified >= ?
   AND pool_emp_id = ?
```

When not using a variable, you should use a join or sub-selection that includes the **?** placeholder.

☞ For more information, see "Synchronizing customers in the Contact sample" [*MobiLink Tutorials,* page 91], and "Synchronizing contacts in the Contact sample" [*MobiLink Tutorials,* page 93].

Partitioning with overlaps

Some tables in your consolidated database may have rows that belong to many remote databases. Each remote database has a subset of the rows in the consolidated database and the subset overlaps with other remote databases. This is frequently the case with a customer table. In this case, there is a many-to-many relationship between the table and the remote databases and there will usually be a table to represent the relationship. The scripts for the download_cursor and download_delete_cursor events need to join the table being downloaded to the relationship table.

Example

The CustDB sample application uses this technique for the ULOrder table. The ULEmpCust table holds the many-to-many relationship information between ULCustomer and ULEmployee.

Each remote database receives only those rows from the ULOrder table for which the value of the emp_id column matches the MobiLink user name.

The Adaptive Server Anywhere version of the download_cursor script for ULOrder in the CustDB application is as follows:

```
SELECT o.order_id, o.cust_id, o.prod_id,
   o.emp_id, o.disc, o.quant, o.notes, o.status
FROM ULOrder o , ULEmpCust ec
WHERE o.cust_id = ec.cust_id
   AND ec.emp_id = ?
   AND ( o.last_modified >= ?
      OR ec.last_modified >= ?)
   AND    ( o.status IS NULL
      OR o.status != 'Approved' )
   AND ( ec.action IS NULL )
```

This script is fairly complex. It illustrates that the query defining a table in the remote database can include more than one table in the consolidated database. The script downloads all rows in ULOrder for which all of the following are true:

- the cust_id column in ULOrder matches the cust_id column in ULEmpCust
- the emp_id column in ULEmpCust matches the synchronization user name
- the last modification of either the order or the employee-customer relationship was later than the most recent synchronization time for this user
- the status is anything other than **Approved**

The action column on ULEmpCust is used to mark columns for delete. Its purpose is not relevant to the current topic.

The download_delete_cursor script is as follows.

```
SELECT o.order_id
FROM ULOrder o, ULEmpCust ec
WHERE o.cust_id = ec.cust_id
  AND ec.emp_id = ?
  AND ( o.last_modified >= ? OR
        c.last_modified >= ? )
  AND ( o.status IS NULL OR
        o.status != 'Approved' )
  AND ( ec.action IS NULL )
```

This script deletes all approved rows from the remote database.

Partitioning child tables

The example in "Partitioning with overlaps" on page 53 illustrates how to partition tables based on a criterion in some other table.

Some tables in your remote database may have disjoint subsets or overlapping subsets, but do not contain a column that determines the subset. These are child tables that usually have a foreign key (or a series of foreign keys) referencing another table. The referenced table has a column that determines the correct subset.

In this case, the download_cursor script and the download_delete_cursor script need to join the referenced tables and have a WHERE clause that restricts the rows to the correct subset.

☞ For an example, see "Synchronizing contacts in the Contact sample" [*MobiLink Tutorials*, page 93].

Maintaining unique primary keys

Every table that is to be synchronized must have a primary key, and the primary key must be unique across all synchronized databases. The values of primary keys should not be updated.

It is often convenient to use a single column as the primary key for tables. For example, each customer should be assigned a unique identification value. If all the sales representatives work in an environment where they can maintain a direct connection to the database, assigning these numbers is easily accomplished. Whenever a new customer is inserted into the customer table, automatically add a new primary key value that is greater than the last value.

In a disconnected environment, assigning unique values for primary keys when new rows are inserted is not as easy. When a sales representative adds a new customer, she is doing so to a remote copy of the Customer table. You must prevent other sales representatives, working on other copies of the Customer table, from using the same customer identification value.

This section describes the following ways to solve the problem of how to generate unique primary keys:

- ♦ Using Universally Unique IDs (UUIDs)
- ♦ Using global autoincrement values.
- ♦ Using primary key pools.

Maintaining unique primary keys using UUIDs

You can ensure that primary keys in Adaptive Server Anywhere databases are unique by using the newid( ) function to create universally unique values for your primary key. The resulting UUIDs can be converted to a string using the uuidtostr( ) function, and converted back to binary using the strtouuid( ) function.

UUIDs are unique across all computers. However, the values are completely random and so cannot be used to determine when a value was added, or the order of values. UUID values are also considerably larger than the values required by other methods (including global autoincrement), and require more table space in both the primary and foreign key tables. Indexes on tables using UUIDs are also less efficient.

☞ For more information, see

- ♦ "The NEWID default" [*ASA SQL User's Guide*, page 86]

- ♦ "NEWID function [Miscellaneous]" [*ASA SQL Reference*, page 185]
- ♦ "UNIQUEIDENTIFIER data type [Binary]" [*ASA SQL Reference*, page 75]

Example

The following CREATE TABLE statement creates a primary key that is universally unique:

```
CREATE TABLE customer (
   cust_key UNIQUEIDENTIFIER NOT NULL
            DEFAULT NEWID( ),
   rep_key VARCHAR(5),
   PRIMARY KEY(cust_key))
```

Maintaining unique primary keys using global autoincrement

In Adaptive Server Anywhere and UltraLite databases, you can set the default column value to be GLOBAL AUTOINCREMENT. You can use this default for any column in which you want to maintain unique values, but it is particularly useful for primary keys.

❖ To use global autoincrement columns

1. Declare the column as a global autoincrement column.

   When you specify default global autoincrement, the domain of values for that column is partitioned. Each partition contains the same number of values. For example, if you set the partition size for an integer column in a database to 1000, one partition extends from 1001 to 2000, the next from 2001 to 3000, and so on.

   ☞ See "Declaring default global autoincrement" on page 58.

2. Set the GLOBAL_DATABASE_ID value.

   Adaptive Server Anywhere supplies default values in a database only from the partition uniquely identified by that database's number. For example, if you assigned the database in the above example the identity number 10, the default values in that database would be chosen in the range 10001–11000. Another copy of the database, assigned the identification number 11, would supply default value for the same column in the range 11001–12000.

   ☞ See "Setting the GLOBAL_DATABASE_ID value" on page 58.

For more information

This section describes how to use global autoincrement columns in Adaptive Server Anywhere remote databases. For information on using global autoincrement columns in UltraLite databases, see "Declaring default global autoincrement columns" [*MobiLink Clients*, page 291].

☞ For information on how global autoincrement columns work in Adaptive Server Anywhere databases, see "How default values are chosen" on page 59.

Declaring default global autoincrement

You can set default values in your database by selecting the column properties in Sybase Central, or by including the DEFAULT GLOBAL AUTOINCREMENT phrase in a CREATE TABLE or ALTER TABLE statement.

Optionally, the partition size can be specified in parentheses immediately following the AUTOINCREMENT keyword. The partition size may be any positive integer, although the partition size is generally chosen so that the supply of numbers within any one partition will rarely, if ever, be exhausted.

For columns of type INT or UNSIGNED INT, the default partition size is 2^{16} = 65536; for columns of other types the default partition size is 2^{32} = 4294967296. Since these defaults may be inappropriate, especially if our column is not of type INT or BIGINT, it is best to specify the partition size explicitly.

For example, the following statement creates a simple table with two columns: an integer that holds a customer identification number and a character string that holds the customer's name.

```
CREATE TABLE customer (
   id   INT          DEFAULT GLOBAL AUTOINCREMENT (5000),
   name VARCHAR(128) NOT NULL,
   PRIMARY KEY (id)
)
```

In the above example, the chosen partition size is 5000.

☞ For more information on GLOBAL AUTOINCREMENT, see "CREATE TABLE statement" [*ASA SQL Reference*, page 407].

Setting the GLOBAL_DATABASE_ID value

When deploying an application, you must assign a different identification number to each database. You can accomplish the task of creating and distributing the identification numbers by a variety of means. One method is to place the values in a table and download the correct row to each database based on some other unique property, such as user name.

❖ To set the global database identification number

1. You set the identification number of a database by setting the value of the public option GLOBAL_DATABASE_ID. The identification number must be a non-negative integer.

Example

For example, the following statement sets the database identification number to 20.

```
SET OPTION PUBLIC.GLOBAL_DATABASE_ID = 20
```

If the partition size for a particular column is 5000, default values for this database are selected from the range 100001–105000.

How default values are chosen

The public option GLOBAL_DATABASE_ID in each database must be set to a unique, non-negative integer. The range of default values for a particular database is $pn + 1$ to $p(n + 1)$, where p is the partition size and n is the value of the public option GLOBAL_DATABASE_ID. For example, if the partition size is 1000 and GLOBAL_DATABASE_ID is set to 3, then the range is from 3001 to 4000.

If GLOBAL_DATABASE_ID is set to a non-negative integer, Adaptive Server Anywhere chooses default values by applying the following rules:

- If the column contains no values in the current partition, the first default value is $pn + 1$.
- If the column contains values in the current partition, but all are less than $p(n + 1)$, the next default value will be one greater than the previous maximum value in this range.
- Default column values are not affect by values in the column outside of the current partition; that is, by numbers less than $pn + 1$ or greater than $p(n + 1)$. Such values may be present if they have been replicated from another database via MobiLink synchronization.

If the public option GLOBAL_DATABASE_ID is set to the default value of 2147483647, a null value is inserted into the column. Should null values not be permitted, the attempt to insert the row causes an error. This situation arises, for example, if the column is contained in the table’s primary key.

Because the public option GLOBAL_DATABASE_ID cannot be set to negative values, the values chosen are always positive. The maximum identification number is restricted only by the column data type and the partition size.

Null default values are also generated when the supply of values within the partition has been exhausted. In this case, a new value of GLOBAL_DATABASE_ID should be assigned to the database to allow default values to be chosen from another partition. Attempting to insert the null value causes an error if the column does not permit nulls. To detect that the supply of unused values is low and handle this condition, create an event of type **GlobalAutoincrement**.

Should the values in a particular partition become exhausted, you can assign a new database id to that database. You can assign new database id numbers in any convenient manner. However, one possible technique is to maintain a pool of unused database id values. This pool is maintained in the same manner as a pool of primary keys.

You can set an event handler to automatically notify the database administrator (or carry out some other action) when the partition is nearly exhausted. For more information, see "Defining trigger conditions for events" [*ASA Database Administration Guide*, page 308].

☞ For more information, see "Setting the GLOBAL_DATABASE_ID value" on page 58, and "GLOBAL_DATABASE_ID option [database]" [*ASA Database Administration Guide*, page 656].

Maintaining unique primary keys using key pools

One efficient means of solving this problem is to assign each user of the database a pool of primary key values to assign as the need arises. For example, you can assign each sales representative 100 new identification values. Each sales representative can freely assign values to new customers from his own pool.

❖ To implement a primary key pool

1. Add a new table to the consolidated database and to each remote database to hold the new primary key pool. Apart from a column for the unique value, these tables should contain a column for a user name, to identify who has been given the right to assign the value.

2. Write a stored procedure to ensure that each user is assigned enough new identification values. Assign more new values to remote users who insert many new entries or who synchronize infrequently.

3. Write a download_cursor script to select the new values assigned to each user and download them to the remote database.

4. Modify the application that uses the remote database so that when a user inserts a new row, the application uses one of the values from the pool.

The application must then delete that value from the pool so it is not used a second time.

5. Write an upload scripts. The MobiLink synchronization server will then delete rows from the consolidated pool of values that a user has deleted from his personal value pool in the remote database.

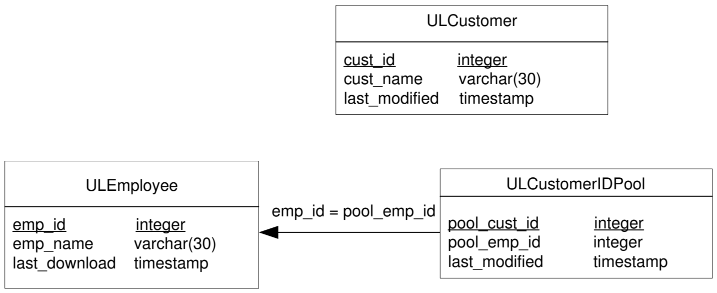
6. Write an end_upload script to call the stored procedure that maintains the pool of values. Doing so has the effect of adding more values to the user's pool to replace those deleted during upload.

Example

The sample application allows remote users to add customers. It is essential that each new row has a unique primary key value, and yet each remote database is disconnected when data entry is occurring.

The ULCustomerIDPool holds a list of primary key values that can be used by each remote database. In addition, the ULCustomerIDPool_maintain stored procedure tops up the pool as values are used up. The maintenance procedures are called by a table-level end_upload script, and the pools at each remote database are maintained by upload_cursor and download_cursor scripts.

1. The ULCustomerIDPool table in the consolidated database holds the pool of new customer identification numbers. It has no direct link to the ULCustomer table.



2. The ULCustomerIDPool_maintain procedure updates the ULCustomerIDPool table in the consolidated database. The following sample code is for an Adaptive Server Anywhere consolidated database.

```
CREATE PROCEDURE ULCustomerIDPool_maintain ( IN syncuser_id
         INTEGER )
BEGIN
    DECLARE pool_count INTEGER;

    -- Determine how may ids to add to the pool
    SELECT COUNT(*) INTO pool_count
    FROM ULCustomerIDPool
    WHERE pool_emp_id = syncuser_id;

       -- Top up the pool with new ids
    WHILE pool_count < 20 LOOP
      INSERT INTO ULCustomerIDPool ( pool_emp_id )
      VALUES ( syncuser_id );
      SET pool_count = pool_count + 1;
    END LOOP;
END
```

This procedure counts the numbers presently assigned to the current user, and inserts new rows so that this user has a sufficient supply of customer identification numbers.

This procedure is called at the end of the upload stream, by the end_upload table script for the ULCustomerIDPool table. The script is as follows:

```
CALL ULCustomerIDPool_maintain( ? )
```

3. The download_cursor script for the ULCustomerIDPool table downloads new numbers to the remote database.

```
SELECT pool_cust_id
FROM ULCustomerIDPool
WHERE pool_emp_id = ?
AND last_modified >= ?
```

4. To insert a new customer, the application using the remote database must select an unused identification number from the pool, delete this number from the pool, and insert the new customer information using this identification number. The following embedded SQL function for an UltraLite application retrieves a new customer number from the pool.

```
bool CDemoDB::GetNextCustomerID( void )
/*************************************/
{
    short   ind;

    EXEC SQL SELECT min( pool_cust_id )
    INTO :m_CustID:ind FROM ULCustomerIDPool;
    if( ind < 0 ) {
        return false;
    }
    EXEC SQL DELETE FROM ULCustomerIDPool
    WHERE pool_cust_id = :m_CustID;
    return true;
}
```

Handling conflicts

Caution
Don't update primary keys in a MobiLink environment. A primary key uniquely identifies a row for conflict detection purposes.

Conflicts arise during the upload of rows to the consolidated database. If two users modify the same row on different remote databases, a conflict is detected when the second of the rows arrives at the MobiLink synchronization server. When conflicts can occur, you should define a process to compute the correct values, or at least to log the conflict.

During the download stage of a synchronization, no conflicts arise in the remote database. If a downloaded row contains a new primary key, the values are inserted into a new row. If the primary key matches that of a pre-existing row, the values in the row are updated.

Conflicts are not the same as errors. Conflict handling is an integral part of a well-designed application.

Default behavior

Conflicts are detected only during updates of a row. An attempt to update a row that has been updated or deleted since that last synchronization is a conflict. By default:

- ♦ If an attempt to insert a row finds that the row has already been inserted, an error is generated.
- ♦ If an attempt to delete a row finds that the row has already been deleted, the attempt to delete is ignored.

These are the built-in behaviors that have been defined for convenience. If you need different behavior, you may implement it yourself using one or more of the upload events.

☞ For a description of how to handle syncrhonization conflicts, see "Tutorial: Writing MobiLink Scripts and Monitoring Synchronizations" [*MobiLink Tutorials,* page 13].

Detecting conflicts

When a MobiLink client sends an updated row to the MobiLink synchronization server, it includes not only the new values (the post-image), but also a copy of the old row values (the pre-image). When the pre-image does not match the current values in the consolidated database, a conflict is detected.

There are two ways to detect conflicts:

♦ Define an upload_fetch script.

♦ Define an upload_update script that includes all non-primary key columns in the WHERE clause.

If you define both of these scripts for the same table, only the upload_fetch script is used.

The MobiLink synchronization server detects conflicts only if an upload_fetch or appropriate upload_update script is applied. If an upload_fetch script is supplied, the MobiLink synchronization server compares the pre-image of an update to the values of the row returned by the upload_fetch script with the same primary key values. If values in the pre-image do not match the current consolidated values, the MobiLink synchronization server detects a conflict.

upload_fetch

The upload_fetch script typically selects a single row of data from a consolidated database table corresponding to the row being updated. A typical upload_fetch script has the following syntax:

SELECT *col1*, *col2*, . . .
FROM *table-name*
WHERE *pk1* **= ? AND** *pk2* **= ?** . . .

☞ For more information, see “upload_fetch table event” on page 461.

upload_update

The upload_update script provides a parameter for each column in the row.

The parameters for an upload_update event are arranged so that statements with the following syntax update rows correctly:

UPDATE *table-name*
SET *col1* **= ?,** *col2* **= ?,** . . .
WHERE *pk1* **= ? AND** *pk2* **= ?** . . .

In this statement, col1, col2 and so on are the non-primary key columns, while pk1, pk2 and so on are primary key columns.

For a conflict to be detected, the syntax must be as follows:

UPDATE *table-name*
SET *col1* **= ?,** *col2* **= ?,** . . .
WHERE *pk1* **= ? AND** *pk2* **= ?** . . .
AND *col1* **= ? AND** *col2* **= ?** . . .

The WHERE clause compares old values uploaded from the remote to current values in the consolidated database. If the values do not match, the update is ignored, thus preserving the values already on the consolidated database.

☞ For more information, see “upload_update table event” on page 475.

Resolving conflicts

You have several options for resolving conflicts:

- ♦ Resolve conflicts as they occur using temporary or permanent tables and a resolve_conflict script.

  ☞ See "Resolving conflicts with resolve_conflict scripts" on page 66.

- ♦ Resolve conflicts as they occur using an upload_update script.

  ☞ See "Resolving conflicts with upload_update scripts" on page 68.

- ♦ Resolve all conflicts at once using a table's end_upload event.

  ☞ See "end_upload table event" on page 404.

☞ For an example of conflict resolution using statement-based uploads, see "Synchronizing products in the Contact sample" [*MobiLink Tutorials,* page 95].

Example

For example, User1 starts with an inventory of ten items, and then sells three and updates the Remote1 inventory value to seven items. User2 sells four items and updates the Remote2 inventory to six. When Remote1 synchronizes, the consolidated database is updated to seven. When Remote2 synchronizes, a conflict is detected because the value of the inventory is no longer ten. To resolve this conflict programmatically, you need three row values:

1. The current value in the consolidated database.
2. The new row value that Remote2 uploaded.
3. The old row value that Remote2 obtained during the last synchronization.

In this case, the business logic would use the following to calculate the new inventory value and resolve the conflict:

```
current consolidated - (old remote - new remote)
-> 7 - (10-6) = 3
```

Resolving conflicts with resolve_conflict scripts

When the MobiLink synchronization server detects a conflict using an upload_fetch script, the following events take place.

- ♦ The MobiLink synchronization server inserts old row values uploaded from the remote database as defined by the upload_old_row_insert script. Typically, the old values are inserted into a temporary table.

  ☞ For more information, see "upload_old_row_insert table event" on page 467.

- The MobiLink synchronization server inserts the new row values uploaded from the remote database as defined by the upload_new_row_insert script. Typically, the new values are inserted into a temporary table.

  ☞ For more information, see "upload_new_row_insert table event" on page 465.

- The MobiLink synchronization server executes the resolve_conflict script. In this script you can either call a stored procedure, or define a sequence of steps to resolve the conflict using the new and old row values.

☞ For more information, see "resolve_conflict table event" on page 442.

Example

In the following example, you create scripts for six events and then you create a stored procedure.

- In the begin_synchronization script, you create two temporary tables called contact_new and contact_old. This is done in begin_synchronization instead of begin_connection so that the corresponding "end_" script will run even if the synchronization fails, ensuring that the temporary tables are deleted.
- In the upload_old_row_insert and upload_new_row_insert scripts, you populate the two temporary tables with the new and old data uploaded from the remote database.
- The upload_fetch script detects the conflict.
- The resolve_conflict script calls the stored procedure MLResolveConflict to resolve the conflict.

Event	Script

begin_synchronization	`CREATE TABLE #contact_new(`
	`   id  INTEGER,`
	`   location CHAR(36),`
	`   contact_date DATE);`
	`CREATE TABLE #contact_old(`
	`   id  INTEGER,`
	`   location CHAR(36),`
	`   contact_date DATE)`

upload_fetch	`SELECT id, location, contact_date`
	`   FROM contact`
	`   WHERE id = ?`

upload_old_row_insert	`INSERT INTO #contact_new( id,`
	`   location, contact_date )`
	`   VALUES ( ?, ?, ? )`

Event	Script
upload_new_row_insert	`INSERT INTO #contact_old( id, location, contact_date ) VALUES ( ?, ?, ? )`
resolve_conflict	`CALL MLResolveConflict( )`
end_synchronization	`DROP TABLE #contact_new;` `DROP TABLE #contact_old`

The stored procedure MLResolveConflict is as follows:

```
CREATE PROCEDURE MLResolveConflict( )
BEGIN
  --update the consolidated database only if the new contact
         date
  --is later than the existing contact date
  UPDATE contact c
    SET c.contact_date = cn.contact_date
      FROM #contact_new cn
      WHERE c.id = cn.id
        AND cn.contact_date > c.contact_date;
  --cleanup
  DELETE FROM #contact_new;
  DELETE FROM #contact_old;
END
```

Resolving conflicts with upload_update scripts

Instead of using the resolve_conflict script for conflict resolution, you can call a stored procedure in the upload_update script. With this technique, you must both detect and resolve conflicts programmatically.

For example, the following stored procedure, sp_update_customer, contains logic for conflict detection and resolution. It accepts new column values, primary keys, and old column values for parameters.

```
call ml_add_table_script( 'version_1', 'customer', 'upload_
         update', '
CALL sp_update_customer (?,?,?,?,?,?)
)
 CREATE PROCEDURE sp_update_customer(
     IN @new_first_name    varchar(30)
    ,IN @new_last_name     varchar(30)
    ,IN @new_nullable_col  varchar(30)
    ,IN @new_last_modified timestamp
    ,IN @cust_1st_pk       integer
    ,IN @cust_2nd_pk       integer
    ,IN @old_first_name    varchar(30)
    ,IN @old_last_name     varchar(30)
    ,IN @old_nullable_col  varchar(30)
    ,IN @old_last_modified timestamp
)
```

To detect a conflict you can check the number of rows affected by the following update operation. The WHERE clause compares old values uploaded from the remote to current values in the consolidated database. If the values match, the update succeeds and no conflict has occurred.

```
UPDATE customer
SET first_name                   = @new_first_name,
 last_name                         = @new_last_name,
 nullable_col                    = @new_nullable_col,
 last_modified                   = @new_last_modified

WHERE cust_1st_pk                = @cust_1st_pk
 AND cust_2nd_pk                 = @cust_2nd_pk
 AND first_name                  = @old_first_name
 AND last_name                   = @old_last_name
 AND COALESCE(nullable_col, '') = COALESCE(@old_nullable_col,
         '')
 AND last_modified               = @old_last_modified;
...
```

Note:
The COALESCE function returns the first non-NULL expression from a list, and is used in this case to compare values for a nullable column.

☞ For more information, see "COALESCE function [Miscellaneous]" [*ASA SQL Reference*, page 115].

You can use a mechanism in your consolidated database to determine the outcome of the update statement shown above. Using Adaptive Server Anywhere, for example, you can check the @@rowcount global variable to determine the number of rows affected by the update. If @@rowcount = 0 then a conflict has occurred, and you can programmatically resolve the conflict, as follows. In this example, the database with the most recent update wins the conflict. If the consolidated database wins the conflict, it retains its current values and no action is taken.

```
IF( @@rowcount = 0 ) THEN
-- A conflict has been detected. To resolve, you:
-- b) Use business logic to determine which values to use
-- c) Update the consolidated database with the final values

 IF( @new_last_modified > @current_last_modified ) THEN
 -- The remote has won the conflict
 -- use the values it uploaded.

  UPDATE customer
   SET first_name    = @new_first_name,
       last_name     = @new_last_name,
       nullable_col  = @new_nullable_col,
       last_modified = @new_last_modified
       WHERE cust_1st_pk  = @cust_1st_pk
        AND cust_2nd_pk   = @cust_2nd_pk;

 END IF;
END IF;
```

Forced conflicts

Forced conflict resolution is a special technique that forces every uploaded row to be treated as if it were a conflict.

The MobiLink synchronization server uses forced conflict resolution when the upload_insert, upload_update, and upload_delete script are all undefined. In this mode of operation, the MobiLink synchronization server attempts to insert all uploaded rows from that table using the statements defined by the upload_old_row_insert and upload_new_row_insert scripts. In essence, all uploaded rows are then treated as conflicts. You can write stored procedures or scripts to process the uploaded values in any way you want.

Without any of the upload_insert, upload_update, or upload_delete scripts, the normal conflict-resolution procedure is bypassed. This technique has two principal uses.

♦ **Arbitrary conflict detection and resolution** The automatic mechanism only detects errors when updating a row, and only then when the old values do not match the present values in the consolidated database.

  You can capture the raw uploaded data using the upload_old_row_insert and upload_new_row_insert scripts, then process the rows as you see fit.

♦ **Performance** When the upload_insert, upload_update, and upload_delete are not defined, the MobiLink synchronization server is relieved of its normal conflict-detection tasks, which involve querying the consolidated database one row at a time. Instead, it needs only to insert the raw uploaded information using the statements defined by the upload_old_row_insert and upload_new_row_insert scripts. Since only inserts are involved, the MobiLink synchronization server performs more efficiently.

Storing the user name

When you write upload_old_row_insert or upload_new_row_insert scripts, you can include an extra parameter in your INSERT statement. If you do so, the MobiLink synchronization server automatically inserts the user name into the first parameter, and then uses the rest of the parameters as usual. This mechanism is available because some database-management systems provide no convenient mechanism to store the identity of the current user.

You can use this feature to conveniently identify which user inserted each row. It is also useful when one or two tables are used to hold the new and old uploaded values from many different users at the same time. This information allows you to include user-specific logic in the resolve_conflict script.

For example, an ordinary upload_old_row_insert script is of the following form. The items in the select list correspond to the columns of the remote table.

```
INSERT INTO MyTable (c1, c2, . . . , cN) VALUES (?,?,...,?)
```

However, the following syntax is also permitted.

```
INSERT INTO MyTableWithUser (user_name, c1, c2, . . . , cN)
         VALUES (?,?,...,?)
```

Normally, the selected columns must match the columns of the remote table in both number and type. This case is an exception. The single extra parameter in the values list must be of a type suitable to hold the user name; for example, VARCHAR(128). The subsequent parameters in the list must match the columns of the remote table in order and type, as usual. If you include more than one extra parameter, an error results.

☞ For more information, see “upload_old_row_insert table event” on page 467 and “upload_new_row_insert table event” on page 465.

Data entry

In some databases, there are tables that are only used for data entry. One way of processing these tables is to upload all inserted rows at each synchronization, and remove them from the remote database on the download stream. After synchronization, the remote table is empty again, ready for another batch of data.

To achieve this model, you can upload rows into a temporary table and then insert them into a base table using an end_upload table script. The temporary table can be used in the download_delete_cursor to remove rows from the remote database following a successful synchronization.

Alternatively, you can allow the client application to the delete the rows, using the STOP SYNCHRONIZATION DELETE statement to stop the deletes being uploaded during the next synchronization.

☞ For more information, see “STOP SYNCHRONIZATION DELETE statement [MobiLink]” [*ASA SQL Reference,* page 637].

Handling deletes

When rows are deleted from the consolidated database, there needs to be a record of the row so it can be removed from any remote databases that have the row.

One technique is to not delete the row. Data that is no longer required can be marked as inactive by changing a status column in the row. In this case, called a logical delete, the download_cursor and download_delete_cursor can refer to the status of the row in the WHERE clause. This technique is used in the ULEmpCust table in the CustDB sample application, in which the action column holds a D for Delete. The scripts use this value to delete the record from the remote database, and delete the record from the consolidated database at the end of the synchronization. CustDB also uses this technique for the ULOrder table, and the Contact sample uses the technique on the Customer, Contact, and Product tables.

A second technique is to have a shadow table that stores the primary key values of deleted rows. When a row is deleted, a trigger can populate the shadow table. The download_delete_cursor can use the shadow table to remove rows from remote databases. The shadow table only needs to have the primary key columns from the real table.

☞ For more information, see "Writing download_delete_cursor scripts" on page 248.

Handling failed downloads

Bookkeeping information about what is downloaded must be maintained in the download transaction. This information is updated atomically with the download being applied to the remote database.

If a failure occurs before the entire download stream is applied to the remote database, and if you change SendDownloadAck to ON, then the MobiLink synchronization server does not get confirmation for the download and rolls back the download transaction. Since the bookkeeping information is part of the download transaction, it is also rolled back. Next time the download stream is built, it will use the original bookkeeping information.

☞ For more information, see "SendDownloadACK (sa) extended option" [*MobiLink Clients,* page 131] or "Send Download Acknowledgement synchronization parameter" [*MobiLink Clients,* page 331].

When testing your synchronization scripts, you should add logic to your end_download script that causes occasional failures. This will ensure that your scripts can handle a failed download.

Resuming failed downloads

Download failure is caused by a communication error during download or a user aborting the download. MobiLink has functionality that helps you recover from download failure, and may help you avoid having to retransmit the entire download. This functionality has separate implementations for Adaptive Server Anywhere and UltraLite remote databases.

Adaptive Server Anywhere remote databases

When synchronization fails during a download, none of the download is applied to the remote database. However, the part of the download that was successfully transmitted is stored in a temporary file on the remote device. You cannot access this file directly, but dbmlsync provides functionality that makes use of the file. When you use this functionality, you may be able to avoid lengthy retransmission of data. You may also be able to recover from download failure automatically.

Note
The download cannot be resumed when the SendDownloadACK extended option is set to ON (the default is OFF) or when the DownloadBufferSize extended option is set to 0 (which is also not the default).

There are three ways to implement this functionality. In all cases, the resumed download will fail if there is any new data to be uploaded, and dbmlsync will abort.

- **-dc** After a download fails, use -dc the next time you start dbmlsync to resume the download. If part of the failed download was transmitted, the MobiLink synchronization server will only transmit the remainder of the download.

  ☞ For more information, see "-dc option" [*MobiLink Clients,* page 103].

- **ContinueDownload (cd) extended option** When used on the dbmlsync command line, the cd extended option works just like the -dc option. You can also store this option in the database, or use sp_hook_dbmlsync_set_extended_options to set this option in a single synchronization.

  ☞ For more information, see "ContinueDownload (cd) extended option" [*MobiLink Clients,* page 112] and "sp_hook_dbmlsync_set_extended_options" [*MobiLink Clients,* page 225].

- **sp_hook_dbmlsync_end hook** You can use the **restart** parameter to cause a download to resume. You know a download is resumable if the **restartable download** parameter is set to true. You can also create logic in the hook to resume a download if a download file exists and is a certain size.

  ☞ For more information, see "sp_hook_dbmlsync_end" [*MobiLink Clients,* page 212].

UltraLite remote databases

You can control the behavior of UltraLite applications following a failed download as follows:

- If you set the Keep Partial Download synchronization parameter to true when you synchronize, and the download fails before completion, then UltraLite applies that portion of the changes that were downloaded. UltraLite also sets the Partial Download Retained synchronization parameter to true.

  The UltraLite database may be in an inconsistent state at this point. Depending on your application, you may wish to ensure that synchronization completes successfully or is rolled back before you allow changes to the data.

  ☞ For more information, see "Keep Partial Download synchronization parameter" [*MobiLink Clients,* page 321], and "Partial Download Retained synchronization parameter" [*MobiLink Clients,* page 324].

- To resume the download, set the Resume Partial Download synchronization parameter to true and synchronize again.

  ☞ For more information, see "Resume Partial Download synchronization parameter" [*MobiLink Clients,* page 327].

The restarted synchronization does not carry out an upload, and downloads only those changes that would have been downloaded by the failed download. That is, it completes the failed download, but does not synchronize changes made since the previous attempt. To get those changes, you would need to synchronize again once the failed download has completed, or and call Rollback Partial Download and synchronize with Resume Partial Download set to false.

When you restart the download, many of the synchronization parameters from the failed synchronization are used again automatically. For example, the publications parameter is ignored: the synchronization downloads those publications requested on the initial download. The only parameters that must be set are the Resume Partial Download parameter (which must be set to true) and the User Name parameter. In addition, settings for the following parameters are obeyed, if set:

- Keep Partial Download (in case of further interruption)
- DisableConcurrency
- Observr
- User Data

- To roll back the changes from the failed download without resuming synchronization, call the function to roll back the changes. This function is ULRollbackPartialDownload function for embedded SQL. For UltraLite components, it is a method on the Connection object.
  - **MobileVB** "RollbackPartialDownload method" [*UltraLite for MobileVB User's Guide,* page 98].
  - **UltraLite.NET** See "RollbackPartialDownload method" [*UltraLite.NET User's Guide,* page 95] (iAnywhere.Data.UltraLite namespace) and "RollbackPartialDownload method" [*UltraLite.NET User's Guide,* page 359] (iAnywhere.UltraLite namespace)..
  - **Native UltraLite for Java** **ianywhere.native_ultralite.Connection.rollbackPartialDownload** method in the API Reference.
  - **UltraLite C++ Component** "RollbackPartialDownload method" [*UltraLite C/C++ User's Guide,* page 316]
  - **UltraLite ActiveX** "RollbackPartialDownload method" [*UltraLite ActiveX User's Guide,* page 95].
  - **Embedded SQL** "ULRollbackPartialDownload function" [*UltraLite C/C++ User's Guide,* page 383].
  - **Static C++ API** "RollbackPartialDownload method" [*UltraLite C/C++ User's Guide,* page 316].
  - **Static Java API** This feature is not available in the Static Java API.

You may wish to roll back the changes from a failed download if synchronization cannot be completed (perhaps the server or network is unavailable) and if you want to maintain a consistent set of data while letting the end user carry on working on the application.

☞ For more information about communications errors, see *ASA Error Messages*.

Note
If the send_download_ack synchronization parameter is set to true (which is not the default), the setting will be ignored for the resumed download.

Downloading a result set from a stored procedure call

You can download a result set from a stored procedure call. For example, you might currently have a download_cursor for the following table:

```
CREATE TABLE MyTable (
    pk INTEGER PRIMARY KEY NOT NULL,
    col1 VARCHAR(100) NOT NULL,
    col2 VARCHAR(20) NOT NULL
)
```

The download_cursor table script might look as follows:

```
SELECT pk, col1, col2
   FROM MyTable
      WHERE last_modified >= ?
      AND employee = ?
```

If you want your downloads to MyTable to use more sophisticated business logic, you can now create your script as follows, where DownloadMyTable is a stored procedure taking two parameters (last-download timestamp and MobiLink user name) and returning a result set. (This example uses an ODBC calling convention for portability):

```
{call DownloadMyTable( ?, ? )}
```

Following are some simple examples for each supported consolidated database. Consult the documentation for your consolidated database for full details.

The following example works with Adaptive Server Anywhere, Adaptive Server Enterprise, and Microsoft SQL Server.

```
CREATE PROCEDURE SPDownload
    @last_dl_ts DATETIME,
    @u_name VARCHAR( 128 )
AS
BEGIN
     SELECT pk, col1, col2
       FROM MyTable
          WHERE last_modified >= @last_dl_ts
          AND employee = @u_name
END
```

The following example works with Oracle. Oracle requires that a package be defined. This package must contain a record type for the result set, and a cursor type that returns the record type.

Note
This example requires that Oracle return a result set. In the ODBC Oracle Driver Setup dialog, you must select the Procedure Returns Results option; or in the connection string, set ProcedureRetResults=1. For more information about setting up the Oracle ODBC driver, see "iAnywhere Solutions ODBC Driver for Oracle" [*ODBC Drivers for MobiLink and Remote Data Access*, page 31].

```
Create or replace package SPInfo as
Type SPRec is record (
    pk      integer,
    col1    varchar(100),
    col2    varchar(20)
);
Type SPCursor is ref cursor return SPRec;
End SPInfo;
```

Next, Oracle requires a stored procedure with the cursor type as the first parameter. Note that the download_cursor script only passes in two parameters, not three. For stored procedures returning result sets in Oracle, cursor types declared as parameters in the stored procedure definition define the structure of the result set, but do not define a true parameter as such.

```
Create or replace procedure
    DownloadMyTable( spcursor IN OUT SPInfo.SPCursor,
                     last_dl_ts IN DATE,
                     user_name IN VARCHAR ) As
Begin
    Open spcursor For
       select pk, col1, col2
         from MyTable
           where last_modified >= last_dl_ts
           and employee = user_name;
End;
```

The following example works with IBM DB2 UDB.

```
CREATE PROCEDURE DownloadMyTable(
   IN last_dl_ts TIMESTAMP,
   IN u_name VARCHAR( 128 ) )
        EXTERNAL NAME 'DLMyTable!DownloadMyTable'
        RESULT SETS 1
        FENCED
        LANGUAGE JAVA PARAMETER STYLE DB2GENERAL
```

The following example is a Java implementation of the stored procedure, in DLMyTable.java. To return a result set, you must leave the result set open when the method returns:

```
import COM.ibm.db2.app.*;
import java.sql.*;

public class DLMyTable extends StoredProc
{
  public void DownloadMyTable(
      Date last_dl_ts,
      String u_name ) throws Exception
  {
      Connection  conn = getConnection();
      conn.setAutoCommit( false );
      Statement   s    = conn.createStatement();
      // Execute the select and leave it open.
      ResultSet   r    = s.executeQuery(
            "select pk, col1, col2 from MyTable"
                  + " where last_modified >= ’"
                  + last_dl_ts
                  + "’ and employee = ’"
                  + u_name + "’" );
    }
}
```

Schema changes in remote databases

As your needs evolve, deployed remote databases may require schema changes. The most common schema changes are adding a new column to an existing table or adding a new table to the database.

Adaptive Server Anywhere remote databases

You can change the schema of remote Adaptive Server Anywhere databases after they are deployed.

❖ To add tables to Adaptive Server Anywhere remote databases

1. Add the associated table scripts in the consolidated database.

   The same script version may be used for the remote database without the new table and the remote database with the new table. However, if the presence of the new table changes how existing tables are synchronized, then you must create a new script version, and create new scripts for all tables being synchronized with the new script version.

2. Perform a normal synchronization.

   This step is optional, but recommended, before changing schema.

3. Use the ALTER PUBLICATION statement to add the table. For example,

   ```
   ALTER PUBLICATION your_pub
      ADD TABLE table_name
   ```

   ☞ For more information, see "ALTER PUBLICATION statement" [*ASA SQL Reference*, page 280].

4. Synchronize. Use the new script version, if required.

Changing table definitions in remote databases

Changing the number or type of columns in an existing table must be done carefully. When a MobiLink client synchronizes with a new schema, it expects scripts, such as upload_update or download_cursor, which have parameters for all columns in the remote table. An older remote database expects scripts that have only the original columns.

❖ To alter a published table in a deployed Adaptive Server Anywhere remote database

1. At the consolidated database, create a new script version.

   ☞ For more information, see "Script versions" on page 239.

2. For your new script version, create scripts for all tables in the publication(s) that contain the table that you want to alter and that are synchronized with the old script version.

3. At the remote database, perform a normal synchronization using the old script version.

4. At the remote database, use the ALTER PUBLICATION statement to temporarily drop the table from the publication. For example,

   ```
   ALTER PUBLICATION your_pub
     DROP TABLE table_name
   ```

   ☞ For more information, see "ALTER PUBLICATION statement" [*ASA SQL Reference*, page 280].

5. At the remote database, use the ALTER TABLE statement to alter the table.

   ☞ For more information, see "ALTER TABLE statement" [*ASA SQL Reference*, page 294].

6. At the remote database, use the ALTER PUBLICATION statement to add the table back into the publication.

   ☞ For more information, see "ALTER PUBLICATION statement" [*ASA SQL Reference*, page 280].

7. Synchronize with the new script version.

Note: Steps 4 through 6 may also be performed by the sp_hook_dbmlsync_schema_upgrade stored procedure. For more information, see "sp_hook_dbmlsync_schema_upgrade" [*MobiLink Clients*, page 223].

☞ For more information about changing schemas for Adaptive Server Anywhere remote databases, see "sp_hook_dbmlsync_schema_upgrade" [*MobiLink Clients*, page 223].

UltraLite remote databases

You can change the schema of a remote UltraLite database by deploying a new application or through a schema upgrade.

♦ If you deploy a new application without a schema upgrade, you need to repopulate the UltraLite database by synchronizing with the MobiLink synchronization server.

♦ In the schema upgrade case, your data will be preserved. It is usually impractical to have all users upgrade to the new version of the application at the same time.

You need to be able to have both versions co-existing in the field and synchronizing with a single consolidated database. You can create two or more versions of the synchronization scripts that are stored in the consolidated database and control the actions of the MobiLink synchronization server. Each version of your application can then select the appropriate set of synchronization scripts by specifying the correct version name when it initiates synchronization.

☞ For more information about schemas in UltraLite, see "Creating UltraLite databases and schemas" [*UltraLite Database User's Guide,* page 28].

CHAPTER 5

File-Based Downloads

About this chapter

This chapter describes an alternative way to download data to Adaptive Server Anywhere remote databases: downloads can be distributed as files, enabling offline distribution of synchronization changes. This allows you to create a file once and distribute it to many remote databases.

This chapter provides a complete description of how to use file-based downloads and then provides two in-depth examples that demonstrate how to set it up.

Contents

Introduction

With file-based downloads, you can put download synchronization changes in a file and transfer it to Adaptive Server Anywhere remote databases in any way a file can be transferred. For example,

- ♦ broadcast the data by satellite multicast
- ♦ apply the update using Sybase Manage Anywhere
- ♦ e-mail or ftp the file to users

You choose the users you want to receive the file. Full synchronization integrity is preserved in file-based downloads, including conflict detection and resolution. You can ensure that the file is secure by applying third-party encryption on the file.

When to use

File-based downloads are useful when a large amount of data changes on the consolidated database, but the remote database does not update the data frequently or does not do any updates at all. For example, price lists, product lists, and code tables.

File-based downloads are not useful when the downloaded data is updated frequently on the remote database or when you are running frequent upload-only synchronizations. In these situations, the remote sites may be unable to apply download files because of integrity checks that are performed when download files are applied.

File-based downloads cannot be used as the sole means of updating remote databases. You still need to regularly perform full synchronizations or upload-only synchronizations. Full or upload-only synchronizations are required to advance log offsets and to maintain the log file, which otherwise will grow large and slow down synchronization. A full synchronization may also be required to recover from errors.

File-based downloads currently can be used only with Adaptive Server Anywhere remote databases.

Setting up file-based downloads

To set up file-based download, you:

1. Create a file-definition database.
2. At the consolidated database, create scripts with a new script version.
3. Create a download file.
4. Apply the download file.

Complete instructions follow.

Create the file-definition database

To set up file-based downloads, you create a **file-definition database**. This is an Adaptive Server Anywhere database that has the same synchronization tables and publications as your remote databases. It can be located anywhere. This database contains no data or state information. It does not have to be backed up or maintained; in fact, you can delete it and recreate it as needed.

The file-definition database must include the following:

- ♦ the same publications as the remote databases, as well as the tables and columns used in the publication, the foreign key relationships and constraints of those tables and columns, and the tables required by those foreign key relationships.
- ♦ a MobiLink user name that identifies the group of remote databases that are to apply the download file. You will use this group MobiLink user name in your synchronization scripts to identify the group of remote databases.

Changes at the consolidated database

On the consolidated database, create a new script version for your file-based downloads, and implement any scripts required by your existing synchronization system into it. Upload scripts are not required. This script version will be used only for file-based downloads. For this script version, all scripts that take MobiLink user names as parameters will instead take a MobiLink user name that refers to a group of remote databases. This is the user name that is defined in the file-definition database.

For each script version that you have defined, implement a begin_publication script.

For timestamp-based downloads, implement a modify_last_download_timestamp script for each script version. How you implement this script depends on how much data you intend to send in each download file. For example, one approach is to use the earliest time that any user from the group last downloaded successfully. Remember that the ml_username parameter passed to this script is actually the group name.

☞ For more information, see "modify_last_download_timestamp connection event" on page 424.

Creating the download file

The download file contains the data to be synchronized. To create the download file, set up your file-definition database and consolidated database as described above. Run dbmlsync with the -bc option and supply a file name with the extension .df. For example,

```
dbmlsync -c "uid=dba;pwd=sql;eng=fbdl_eng;dbf=fdef.db" -v+
   -e "sv=filebased" -bc file1.df
```

Optionally, you can specify options when you create the download file:

♦ **-be option** Use -be to add a string to the download file that can be accessed at the remote database using the sp_hook_dbmlsync_validate_download_file stored procedure.

  ☞ For more information, see "-be option" [*MobiLink Clients*, page 101] "sp_hook_dbmlsync_validate_download_file" [*MobiLink Clients*, page 233]and .

♦ **-bg option** Use the -bg option to create a download file that can be used by remotes that have never synchronized.

  Use the -bg option to create a download file that can be used by remotes that have never synchronized.

Synchronizing new remotes

If you want to apply a download file to a remote database that has never synchronized using MobiLink, then before you apply the download file you need to either perform a normal synchronization on the remote database or use the dbmlsync -bg option when creating the download file.

For timestamp-based synchronization, doing either of these two things causes the download of an initial snapshot of the data. For both timestamp and snapshot based synchronization, this step sets the generation number to the value that is generated by the begin_publication script on the consolidated database.

Perform a normal synchronization

You can prepare a remote database to receive download files by performing a synchronization that does not use a download file.

Use the -bg option

Alternatively, you can create a download file with the -bg option to use with remotes that haven't yet synchronized. You apply this initial download file to prepare the remote database for file-based synchronization.

♦ **Snapshot downloads** If you are performing snapshot downloads, then the initial download file just needs to set the generation number. You may choose to include an initial snapshot of the data in this file, but since each snapshot download contains all the data and does not depend on previous downloads, this is not required.

  For snapshot downloads, using the -bg option is straightforward. Just specify -bg in the dbmlsync command line when you create the download file. You can use the same script version to create the initial download file as you use for subsequent download files.

♦ **Timestamp-based downloads** If you are performing timestamp-based downloads, then the initial download must set the generation number on the remote database and include a snapshot of the data. With timestamp-based downloads, each download builds on previous ones. Each download file contains a last download timestamp. All rows changed on the consolidated after the file's last download timestamp are included in the file. To apply a file, a remote database must already have received all the changes that occurred before the file's last download timestamp. This is confirmed by checking that the file's last download timestamp is greater than or equal to the remote database's last download timestamp (the time up to which the remote database has received all changes from the consolidated database).

  Before a remote can apply its first normal download file, it must receive all data changed before that file's last download timestamp and after January 1, 1900. The initial download file created with the -bg option must contain this data. The easiest way to select this data is to create a separate script version that uses the same download_cursor's as your normal file-based synchronization script version but does not have a modify_last_download_timestamp script. If no modify_last_download_timestamp script is defined, then the last download timestamp for a file-based download will default to January 1, 1900.

If you apply download files built with the -bg option to remote databases that have already synchronized, the -bg option causes the generation numbers on the remote database to be updated with the value on the consolidated database at the time the download file was created. This defeats the purpose of generation numbers, which is to prevent you from applying further file-based downloads until an upload has been performed in situations such as when recovering a consolidated database that is lost or corrupted.

☞ For more information about generation numbers, see "MobiLink generation numbers" on page 93.

Validation checks

Before applying a download file to a remote database, dbmlsync does a number of things to ensure that the synchronization is valid.

- ♦ dbmlsync checks the download file to ensure that the file-definition database that was used to create it has:
  - the same publication as the remote database
  - the same tables and columns used in the publication
  - the same foreign key relationships and constraints as those tables and columns
- ♦ dbmlsync checks to see if there is any data in the publication that has not been uploaded from the remote. If there is, the download file is not applied, because applying the download file could cause pending upload data to be lost.
- ♦ dbmlsync checks the last download timestamp, next last download timestamp, and creation time of the download file to ensure that:
  - newer data on the remote database will not be overwritten by older data contained in the download file.
  - a download file will not be applied if applying it means that the remote database would miss some changes that have occurred on the consolidated database. This situation might occur if the remote did not apply previous file-based downloads.

    ☞ For more information, see “Automatic validation” on page 91.
- ♦ Optionally, dbmlsync checks the generation number in the remote database to ensure it matches the generation number in the download file.

  ☞ For more information, see “MobiLink generation numbers” on page 93.
- ♦ Optionally, you can create custom validation logic with the sp_hook_dbmlsync_validate_download_file stored procedure.

  ☞ For more information, see “Custom validation” on page 93.

Automatic validation

Before applying a download file, dbmlsync performs special checks on the last download timestamp, next last download timestamp, download file creation time, and transaction log.

Last download timestamp and next last download timestamp

Each download file contains all changes to be downloaded that occurred on the consolidated database between the file’s last download timestamp, and its next last download timestamp. Both times are expressed in terms of the time at the consolidated database. By default the file’s last download time is Jan 1, 1900 12:00 AM and the file’s next last download timestamp is the time the download file was created. These defaults can be overridden by implementing the modify_last_download_timestamp and modify_next_last_download_timestamp scripts on the consolidated database.

A remote site can apply a download file only if the file’s last download timestamp is less than or equal to the remote’s last download timestamp. This ensures that a remote never misses operations that occur on the consolidated database. Usually when a file-based download fails based on this check, the remote has missed one or more download files. The situation can be corrected by applying the missing download files or by performing a full or download-only synchronization.

In addition, a remote site can apply a download file only if the file’s next last download timestamp is greater than the remote’s last download timestamp. The remote’s last download timestamp is the time (at the consolidated database) up to which the remote has received all changes that are to be downloaded. The remote database’s last download time is updated each time the remote successfully applies a download (normal or file-based). This check ensures that a download file will not be applied if more recent data has already been downloaded. A common case where this could happen occurs when download files are applied out of order. For example, suppose a download file F1.df is created, and another file F2.df is created later. This check ensures that F1.df cannot be applied after F2.df, because that could allow newer data in F2.df to be overwritten with older data in F1.df.

When a file-based download fails based on the next last download timestamp, no additional action is required other than to delete the file. Synchronization will succeed once a new file is received.

Creation time

The download file’s creation time indicates the time at the consolidated database when creation of the file began. A download file can only be applied if the file’s creation time is greater than the remote database’s last upload time. The remote’s last upload time is the time at the consolidated database when the remote’s last successful upload was committed. This check ensures that data that has been uploaded after the creation of the download (and hence is newer than the download) will not be overwritten by older data in the download file.

When a download file is rejected based on this check, no action is required. The remote site should be able to apply the next download file.

When an upload fails because dbmlsync sent an upload to the MobiLink synchronization server but got no acknowledgement, the remote database's last upload time may be incorrect. In this case, the creation time check cannot be performed and the remote is unable to apply download files until it completes a normal synchronization.

Transaction log

Before applying a download file, dbmlsync scans the remote database's transaction log and builds up a list of all changes that must be uploaded. Dbmlsync will only apply a download file if it does not contain any operations that affect rows with changes that must be uploaded.

MobiLink generation numbers

Generation numbers provide a mechanism for forcing remote databases to upload data before applying any more download files. This is especially useful when a problem on the consolidated database has resulted in data loss and you must recover lost data from the remote databases.

On the remote database, a separate generation number is automatically maintained for each subscription. On the consolidated database, the generation number for each subscription is determined by the begin_publication script. Each time a remote performs a successful upload, it updates the remote generation number with the value set by the begin_publication script in the consolidated database.

Each time a download file is created, the generation number set by the begin_publication script is stored in the download file. A remote site will only apply a download file if the generation number in the file is equal to the generation number stored in the remote database.

Note
Whenever the generation number generated by the begin_publication script for a file-based download changes, the remote databases must perform a successful upload before they can apply any new download files.

The sp_hook_dbmlsync_validate_download_file stored procedure can be used to override the default checking of the generation number.

For more information about managing MobiLink generation numbers, see:

- "begin_publication connection event" on page 356
- "end_publication connection event" on page 395
- "sp_hook_dbmlsync_validate_download_file" [*MobiLink Clients,* page 233]

Custom validation

You can create custom validation logic to determine if a download file should be applied to a remote database. You do this with the sp_hook_dbmlsync_validate_download_file stored procedure. With this stored procedure, you can both reject a download file and override the default checking of the generation number.

You can use the dbmlsync -be option to embed a string in the file. You use the -be option against the file-definition database when you create the download file This string is passed to the sp_hook_dbmlsync_validate_download_file through the #hook_dict table, and can be used in your validation logic.

☞ For more information, see "sp_hook_dbmlsync_validate_download_file" [*MobiLink Clients,* page 233].

Examples

This section contains two very simple examples. Each sets up a file-based download synchronization using a consolidated database with only one table. The first is a snapshot example and the second is a timestamp-based example.

Snapshot example

This example implements file-based download for snapshot synchronization. It set up the three databases that are required by the file-based download, and then demonstrates how to download data. This example is presented in such a way that you can either just read through it, or you can cut and paste the text to run the sample.

Create databases for the sample

The following commands create the three databases used in the example: a consolidated database, a remote database, and a file-definition database.

```
dbinit scons.db
dbinit sremote.db
dbinit sfdef.db
```

The following commands start the three databases, create a data source name for MobiLink to use to connect to the consolidated database, and start the MobiLink synchronization server.

```
dbeng9 -n sfdef_eng sfdef.db
dbeng9 -n scons_eng scons.db
dbeng9 -n sremote_eng sremote.db
dbdsn -y -w fbd_demo -c "eng=scons_eng;dbf=scons.db;uid=dba;
   pwd=sql;astart=off;astop=off"
start dbmlsrv9 -v+ -c "dsn=fbd_demo"
   -zu+ -ot scons.txt
```

Set up the snapshot example consolidated database

In this example, the consolidated database has one table, called T1. After connecting to the consolidated database, you can run the following SQL to create table T1:

```
CREATE TABLE T1 (
   pk   INTEGER PRIMARY KEY,
   c1   INTEGER
);
```

The following code creates a script version called filebased and creates a download script for that script version.

```
CALL ml_add_table_script( 'filebased',
  'T1', 'download_cursor',
    'SELECT pk, c1 FROM T1' );
```

The following code creates a script version called normal and creates upload and download scripts for that script version.

```
CALL ml_add_table_script ( 'normal', 'T1',
  'upload_insert',
  'INSERT INTO T1 VALUES (?,?)');
CALL ml_add_table_script( 'normal', 'T1',
  'upload_update',
  'UPDATE T1 SET c1 = ? WHERE pk = ? ' );

CALL ml_add_table_script( 'normal', 'T1',
 'upload_delete',
   'DELETE FROM T1 WHERE pk = ?' );

CALL ml_add_table_script( 'normal', 'T1',
  'download_cursor',
    'SELECT pk, c1 FROM T1' );

COMMIT;
```

The following command creates the stored procedure begin_pub and specifies that begin_pub is the begin_publication script for both the “normal” and “filebased” script versions:

```
CREATE PROCEDURE begin_pub (
       INOUT generation_num integer,
           IN    username       varchar(128),
           IN    pubname        varchar(128) )
BEGIN
   SET gnum=1;
END;

CALL ml_add_connection_script(
  'filebased',
  'begin_publication',
      '{ call begin_pub( ?, ?, ? ) }' );

CALL ml_add_connection_script( 'normal',
  'begin_publication',
      '{ call begin_pub( ?, ?, ? ) }' );
```

Create the snapshot example remote database

In this example, the remote database also contains one table, called T1. Connect to the remote database and run the following SQL to create the table T1, a publication called P1, and a user called U1. The SQL also creates a subscription for U1 to P1.

```
CREATE TABLE T1 (
   pk   INTEGER PRIMARY KEY,
   c1   INTEGER
);

CREATE PUBLICATION P1 (
    TABLE T1
);

CREATE SYNCHRONIZATION USER U1;

CREATE SYNCHRONIZATION SUBSCRIPTION
TO P1
FOR U1;
```

The following code creates an sp_hook_dbmlsync_validate_download_file hook to implement user-defined validation logic in the remote database:

```
CREATE PROCEDURE sp_hook_dbmlsync_validate_download_file()
BEGIN
    DECLARE udata   varchar(256);
SELECT value
    INTO udata
    FROM #hook_dict
    WHERE name = 'user data';
    IF udata <> 'ok' THEN
    UPDATE #hook_dict
      SET value = 'FALSE'
      WHERE name = 'apply file';
    END IF;
END
```

Create the snapshot example file-definition database

A file-definition database is required in MobiLink systems that use file-based downloads. This database has the same schema as the remote databases being updated by file-based download, and it contains no data or state information. The file-definition database is used solely to define the structure of the data that is to be included in the download file. One file-definition database can be used for many groups of remote databases, each defined by its own MobiLink group user name.

The following code defines the file-definition database for this sample. It creates a schema that is identical to the remote database, and also creates:

- a publication called P1 that publishes all rows of the T1 table. The same publication name must be used in the file-definition database and the remote databases.
- a MobiLink user called G1. This user represents all the remotes that are to be updated in the file-based download.
- a subscription to the publication

You must connect to sfdef.db before running this code.

```
CREATE TABLE T1 (
   pk   INTEGER PRIMARY KEY,
   c1   INTEGER
);

CREATE PUBLICATION P1 (
    TABLE T1
);

CREATE SYNCHRONIZATION USER G1;

CREATE SYNCHRONIZATION SUBSCRIPTION
TO P1
FOR G1;
```

Prepare for initial synchronization

To prepare your new remote database so that you can apply a download file, you need to either perform a normal synchronization or create the download file with the dbmlsync -bg option. This example shows you how to initialize your new remote database by performing a normal synchronization.

You can perform an initial synchronization of the remote database with the script version called normal that was created earlier:

```
dbmlsync -c "uid=dba;pwd=sql;eng=sremote_eng;
  dbf=sremote.db" -v+ -e "sv=normal"
```

Demonstrate the snapshot example file-based download

Connect to the consolidated database and insert some data that will be synchronized by file-based download, such as the following:

```
INSERT INTO T1 VALUES( 1, 1 );
INSERT INTO T1 VALUES( 2, 4 );
INSERT INTO T1 VALUES( 3, 9 );
COMMIT;
```

The following command must be run on the computer that holds the file-definition database. It does the following:

♦ the dbmlsync -bc option creates the download file, and names it file1.df.

♦ the -be option includes the string “OK” in the download file that will be accessible to the sp_dbmlsync_validate_download_file hook.

```
dbmlsync -c
   "uid=dba;pwd=sql;eng=sfdef_eng;dbf=sfdef.db"
   -v+ -e "sv=filebased" -bc file1.df -be ok -ot fdef.txt
```

To apply the download file, run dbmlsync with the -ba option on the remote database, supplying the name of the download file you want to apply:

```
dbmlsync -c "uid=dba;pwd=sql;eng=sremote_eng;
   dbf=sremote.db" -v+ -ba file1.df -ot remote.txt
```

The changes are now applied to the remote database. Open Interactive SQL, connect to the remote database, and run the following SQL command to verify that the remote has the data:

```
SELECT * FROM T1
```

Clean up the snapshot example

The following commands stop all three database engines and erase the files.

```
del file1.df
dbmlstop -h -w
dbstop -y -c eng=sfdef_eng
dbstop -y -c eng=scons_eng
dbstop -y -c eng=sremote_eng
dberase -y sfdef.db
dberase -y scons.db
dberase -y sremote.db
```

Timestamp-based example

This example implements file-based download for timestamp-based synchronization. It set up the three databases that are required by the file-based download, and then demonstrates how to download data with this functionality. This example is presented in such a way that you can either just read through it, or you can cut and paste the text to run the sample.

Create databases for the sample

The following commands create the three databases used in the example: a consolidated database, a remote database, and a file-definition database.

```
dbinit tcons.db
dbinit tremote.db
dbinit tfdef.db
```

The following commands start the three databases, create a data source name for MobiLink to use to connect to the consolidated database, and start the MobiLink synchronization server.

```
dbeng9 -n tfdef_eng tfdef.db
dbeng9 -n tcons_eng tcons.db
dbeng9 -n tremote_eng tremote.db
dbdsn -y -w tfbd_demo -c "eng=tcons_eng;dbf=tcons.db;uid=dba;
   pwd=sql;astart=off;astop=off"
start dbmlsrv9 -v+ -c "dsn=tfbd_demo" -zu+ -ot tcons.txt
```

Set up the timestamp example consolidated database

In this example, the consolidated database has one table, called T1. After connecting to the consolidated database, you can run the following code to create T1:

```
CREATE TABLE T1 (
   pk      INTEGER PRIMARY KEY,
   c1      INTEGER,
   last_modified   TIMESTAMP DEFAULT TIMESTAMP
);
```

The following code defines a script version called normal with a minimal number of scripts. This script version is used for synchronizations that do *not* use file-based download.

```
CALL ml_add_table_script( 'normal', 'T1',
   'upload_insert',
     'INSERT INTO T1( pk, c1) VALUES( ?, ? )' );

CALL ml_add_table_script( 'normal', 'T1',
   'upload_update',
   'UPDATE T1 SET c1 = ? WHERE pk = ? ' );

CALL ml_add_table_script( 'normal', 'T1',
   'upload_delete',
     'DELETE FROM T1 WHERE pk = ?' );

CALL ml_add_table_script( 'normal', 'T1',
   'download_cursor',
     'SELECT pk, c1 FROM T1 WHERE last_modified >= ?' );
```

The following code sets the generation number for all subscriptions to 1. It is good practice to use generation numbers in case your consolidated database ever becomes lost or corrupted and you need to force an upload.

```
CREATE PROCEDURE begin_pub (
       INOUT generation_num integer,
           IN    username       varchar(128),
           IN    pubname        varchar(128) )
BEGIN
   SET generation_num = 1;
END;

CALL ml_add_connection_script( 'normal',
    'begin_publication',
           '{ call begin_pub( ?, ?, ? ) }' );

COMMIT;
```

The following code defines the script version called filebased. This script version is used to create file-based downloads.

```
CALL ml_add_connection_script( 'filebased',
    'begin_publication',
            '{ call begin_pub( ?, ?, ? ) }' );

CALL ml_add_table_script( 'filebased', 'T1',
    'download_cursor',
      'SELECT pk, c1 FROM T1 WHERE last_modified >= ?' );
```

The following code sets the last download time so that all changes that occurred within the last five days will be included in download files. Any remote that has missed all the download files created in the last five days will have to perform a normal synchronization before being able to apply any more file-based downloads.

```
CREATE PROCEDURE ModifyLastDownloadTimestamp(
             INOUT last_download_timestamp TIMESTAMP,
             IN    ml_username             VARCHAR(128) )
BEGIN
    SELECT dateadd( day, -5, CURRENT TIMESTAMP )
    INTO last_download_timestamp;
END;


CALL ml_add_connection_script( 'filebased',
   'modify_last_download_timestamp',
         'CALL ModifyLastDownloadTimestamp( ?, ? )' );

COMMIT;
```

Create the timestamp example remote database

In this example, the remote database also contains one table, called T1. After connecting to the remote database, run the following code to create table T1, a publication called P1, and a user called U1. The code also creates a subscription for U1 to P1.

```
CREATE TABLE T1 (
   pk   INTEGER PRIMARY KEY,
   c1   INTEGER
);

CREATE PUBLICATION P1 (
    TABLE T1
);

CREATE SYNCHRONIZATION USER U1;

CREATE SYNCHRONIZATION SUBSCRIPTION
TO P1
FOR U1;
```

The following code defines a sp_hook_dbmlsync_validate_download_file stored procedure. This stored procedure prevents the application of download files that do not have the string “ok” embedded in them.

```
CREATE PROCEDURE sp_hook_dbmlsync_validate_download_file()
BEGIN
    DECLARE udata   varchar(256);

    SELECT value
    INTO udata
    FROM #hook_dict
    WHERE name = 'user data';

    IF udata <> 'ok' THEN
      UPDATE #hook_dict
        SET value = 'FALSE'
        WHERE name = 'apply file';
    END IF;
END
```

Create the timestamp example file-definition database

The following code defines the file-definition database for the timestamp example. It creates a table, a publication, a user, and a subscription for the user to the publication.

```
CREATE TABLE T1 (
   pk   INTEGER PRIMARY KEY,
   c1   INTEGER
);

CREATE PUBLICATION P1 (
    TABLE T1
);

CREATE SYNCHRONIZATION USER G1;

CREATE SYNCHRONIZATION SUBSCRIPTION
TO P1
FOR G1;
```

Prepare for initial synchronization

To prepare your new remote database so that you can apply a download file, you need to either perform a normal synchronization or create the download file with the dbmlsync -bg option. This example shows you how to use -bg.

The following code defines a script version called filebased_init for the consolidated database. This script version has a single begin_publication script.

```
CALL ml_add_table_script(
  'filebased_init', 'T1', 'download_cursor',
    'SELECT pk, c1 FROM T1' );

CALL ml_add_connection_script(
  'filebased_init',
  'begin_publication',
      '{ call begin_pub( ?, ?, ? ) }' );

COMMIT;
```

The following two command lines create and apply an initial download file using the script version called filebased_init and the -bg option.

```
dbmlsync -c "uid=dba;pwd=sql;eng=tfdef_eng;dbf=tfdef.db"
  -v+ -e "sv=filebased_init" -bc tfile1.df -be ok -bg
  -ot tfdef1.txt

dbmlsync -c "uid=dba;pwd=sql;eng=tremote_eng;dbf=tremote.db"
  -v+ -ba tfile1.df -ot tremote.txt
```

Demonstrate the timestamp example file-based download

Connect to the consolidated database and insert some data that will be synchronized by file-based download, such as the following:

```
INSERT INTO T1(pk, c1) VALUES( 1, 1 );
INSERT INTO T1(pk, c1) VALUES( 2, 4 );
INSERT INTO T1(pk, c1) VALUES( 3, 9 );
commit;
```

The following command line creates a download file containing the new data.

```
dbmlsync -c
   "uid=dba;pwd=sql;eng=tfdef_eng;dbf=tfdef.db"
   -v+ -e "sv=filebased" -bc tfile2.df -be ok -ot tfdef2.txt
```

The following command line applies the download file to the remote database.

```
dbmlsync -c "uid=dba;pwd=sql;eng=tremote_eng;dbf=tremote.db"
   -v+ -ba tfile2.df -ot tfdef3.txt
```

The changes are now applied to the remote database. Open Interactive SQL, connect to the remote database, and run the following SQL command to verify that the remote has the data:

```
SELECT * FROM T1
```

Clean up the timestamp example

The following commands stop all three database engines and then erase the files.

```
del file1.df
dbmlstop -h -w
dbstop -y -c eng=tfdef_eng
dbstop -y -c eng=tcons_eng
dbstop -y -c eng=tremote_eng
dberase -y tfdef.db
dberase -y tcons.db
dberase -y tremote.db
```

CHAPTER 6

MobiLink Performance

About this chapter

This chapter provides information that can help you improve the performance of your MobiLink synchronization.

☞ For more information about MobiLink performance, see the *MobiLink Performance* whitepaper at *http://my.sybase.com/detail?id=1009664*.

☞ For a hands-on description of how to monitor performance, see "Tutorial: Writing MobiLink Scripts and Monitoring Synchronizations" [*MobiLink Tutorials*, page 13].

Contents

Performance tips

Following are some suggestions to help you get the best performance out of MobiLink.

- **Test** Before deploying, perform volume testing using the same hardware and network that you plan to use for production. Use this time to experiment with the following performance tips.

- **Avoid contention** Avoid contention in your synchronization scripts. Another way of putting this is that you should maximize concurrency.

  For example, suppose a begin_download script increments a column in a table to count the total number of downloads. If multiple users synchronize at the same time, this script would effectively serialize their downloads. The same counter would be better in the begin_synchronization or end_synchronization script because these scripts are called just before a commit.

  ☞ For more information about contention, see "Contention" on page 111.

  ☞ For information on the transaction structure of synchronization, see "Transactions in the synchronization process" on page 19.

- **Use an optimal number of worker threads** Use the MobiLink –w option to set the number of MobiLink worker threads to the smallest number that gives you optimum throughput. You will need to experiment to find the best number for your situation.

  A larger number of worker threads can improve throughput by allowing more synchronizations to occur at the same time.

  Keeping the number of worker threads small reduces the chance of contention in the consolidated database, the number of connections to the consolidated database, and the memory required for optimal caching.

  For example, in tests with fast clients, it was discovered that approximately five worker threads gave optimum throughput. For slower clients, more worker threads were needed to maximize download throughput, and the best upload throughput was obtained by limiting the number that can simultaneously upload, via the -wu option. In tests with extremely slow clients, the best throughput for both uploads and downloads was obtained with hundreds of worker threads with only five allowed to upload simultaneously. Note that these numbers are from a specific set of tests. Every deployment has different characteristics, and you must test to determine the optimal values for -w and -wu.

  ☞ For more information about worker threads, see "Number of worker threads" on page 112.

☞ For more information, see "-w option" on page 212 and "-wu option" on page 214.

- **Enable the client-side download buffer for ASA clients** For Adaptive Server Anywhere clients, a download buffer allows a MobiLink worker thread to transmit the download without waiting for the client to apply the download. The download buffer is enabled by default. However, the download buffer cannot be used if download acknowledgement is enabled (see next bullet).

  ☞ For more information about setting the download buffer size, see the "DownloadBufferSize (dbs) extended option" [*MobiLink Clients,* page 114].

- **Do not enable download acknowledgement for ASA clients** By default, download acknowledgement is not enabled. This frees up MobiLink worker threads that otherwise would be waiting for confirmation of successful download from the client, which also frees up the connection that the worker thread is using. It also makes it possible for the MobiLink synchronization server to buffer the downloads.

  UltraLite clients do not buffer downloads, so the benefit of not acknowledging downloads is not as great.

  ☞ For more information about download acknowledgements, see the "SendDownloadACK (sa) extended option" [*MobiLink Clients,* page 131].

- **Set the upload cache size** To avoid the situation where the upload cache overflows to disk, set the upload cache size to be larger than the size of your largest upload stream times the number of worker threads. You set the upload cache size with the dbmlsrv9 –u option.

  ☞ For more information, see "-u option" on page 211.

- **Set the download cache size** To avoid the situation where the download buffer overflows to disk, set the download cache size to be larger than the size of your largest download times the number of worker threads. You set the download cache size with the dbmlsrv9 -d option.

  ☞ For more information about setting the memory allocated to the download buffer, see "-d option" on page 198.

- **Set the BLOB cache size** If your rows have data of type LONG VARCHAR or LONG BINARY, you can avoid having the BLOB cache access disk if you set the BLOB cache size to be larger than twice the largest BLOB data in a row times the number of worker threads. You set the BLOB cache size with the dbmlsrv9 -bc option.

  ☞ For more information, see "-bc option" on page 196.

- **Set maximum number of database connections** Set the maximum number of MobiLink database connections to be your typical number of synchronization script versions times the number of MobiLink worker threads, plus one. This reduces the need for MobiLink to close and create database connections. You set the maximum number of connections with the dbmlsrv9 -cn option.

  ☞ For more information, see "MobiLink database connections" on page 114 and "-cn option" on page 197.

- **Have sufficient physical memory** Ensure that the computer running MobiLink has enough physical memory to accommodate the upload, download and BLOB caches in addition to its other memory requirements.

- **Use sufficient processing power** Dedicate enough processing power to MobiLink so that the MobiLink server processing is not a bottleneck. In tests with an Adaptive Server Anywhere consolidated database, MobiLink required a third to a half of the processing required by Adaptive Server Anywhere when both were stressed and executing on the same computer.

- **Use minimum logging verbosity** Use the minimum logging verbosity that is compatible with your business needs. By default, verbose logging is off, and MobiLink does not write its log to disk. You can control logging verbosity with the -v option, and enable logging to a file with the -o or -ot options.

  As an alternative to verbose log files, you can monitor your synchronizations with the MobiLink Monitor. The Monitor does not need to be on the same computer as the MobiLink synchronization server, and a Monitor connection has negligible effect on MobiLink server performance. For more information, see "MobiLink Monitor" on page 117.

- **Java or .NET vs. SQL synchronization logic** No significant throughput difference has been found between using Java or .NET synchronization logic vs. SQL synchronization logic. However, Java and .NET synchronization logic have some extra overhead per synchronization and require more memory.

  In addition, SQL synchronization logic is executed on the computer that runs the consolidated database, while Java or .NET synchronization logic is executed on the computer that runs the MobiLink server. Thus, Java or .NET synchronization logic may be desirable if your consolidated database is heavily loaded.

- **Priority synchronization** If you have some tables that you need to synchronize more frequently than others, create a separate publication and subscription for them. You can synchronize this priority publication more frequently than other publications, and synchronize other publications at off-peak times.

- **Download only the rows you need** Take care to download only the rows that are required. It is easier to write synchronization scripts that download all rows upon each synchronization, but downloading unneeded rows affects synchronization performance.

- **Optimize script execution** The performance of your scripts in the consolidated database is an important factor. It may help to create indexes on your tables so that the upload and download cursor scripts can efficiently locate the required rows. However, too many indexes may slow uploads.

- **For large uploads from ASA clients, estimate the number of rows** You can significantly improve the speed of uploading a large number of rows by providing dbmlsync with an estimate of the number of rows that will be uploaded. You do this with the dbmlsync -urc option.

  ☞ For more information, see "-urc option" [*MobiLink Clients,* page 149].

Key factors influencing MobiLink performance

The overall performance of any system, including throughput for MobiLink synchronization, is usually limited by a bottleneck at one point in the system. For MobiLink synchronization, the following might be the bottlenecks limiting synchronization throughput:

- **The performance of the consolidated database** Of particular importance for MobiLink is the speed at which it can execute the MobiLink scripts. Multiple worker threads might execute scripts simultaneously, so for best throughput you need to avoid database contention in your synchronization scripts.

- **The bandwidth for MobiLink to consolidated communication** This is unlikely to be a bottleneck if both MobiLink and the consolidated database are running on the same computer, or if they are on separate computers connected by a high-speed network.

- **The speed of the computer running MobiLink** If the processing power of the computer running MobiLink is slow, or if it does not have sufficient memory for the MobiLink worker threads and buffers, then MobiLink execution speed could be a synchronization bottleneck. The MobiLink server's performance depends little on disk speed as long as the buffers and worker threads fit in physical memory.

- **The number of MobiLink worker threads** A smaller number of threads will involve fewer database connections, less chance of contention in the consolidated database and less operating system overhead. However, too small a number may leave clients waiting for a free worker thread, or have fewer connections to the consolidated database than it can overlap efficiently.

- **The bandwidth for client-to-MobiLink communications** For slow connections, such as those over dial-up or wide-area wireless networks, the network may cause clients and MobiLink worker threads to wait for data to be transferred.

- **The client processing speed** Slow client processing speed is more likely to be a bottleneck in downloads than uploads, since downloads involve more client processing as rows and indexes are written.

Tuning MobiLink for performance

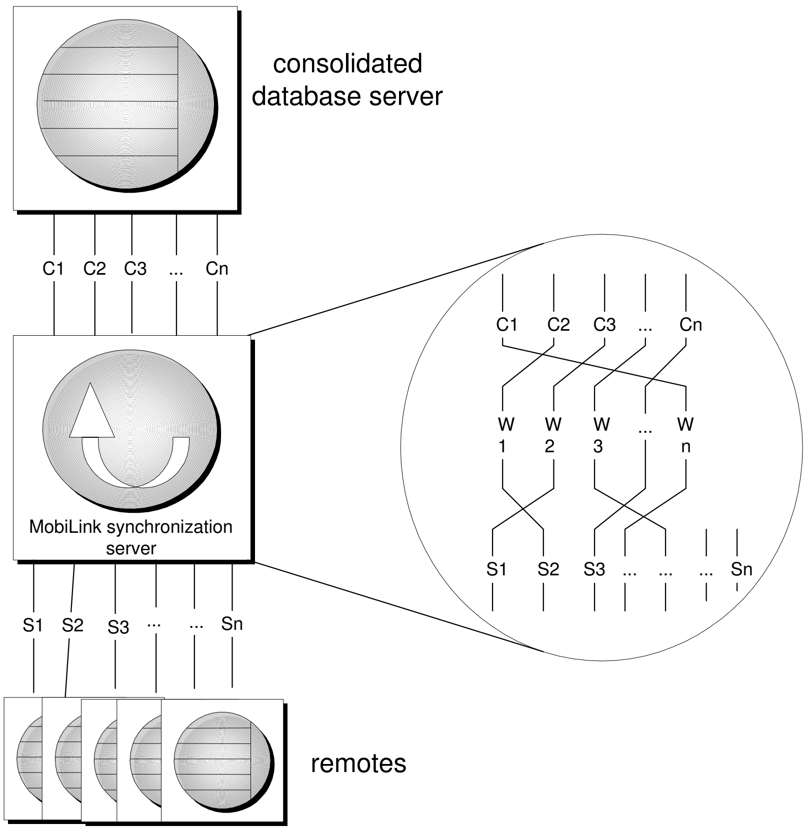
The key to achieving optimal MobiLink synchronization throughput is to have multiple synchronizations occurring simultaneously and executing efficiently. To enable multiple simultaneous synchronizations, MobiLink assigns a worker thread to each synchronization. A worker thread receives the changes uploaded from the client and applies them to the consolidated database. It then fetches the changes from the consolidated database, and downloads them to the client. Each worker thread uses a single connection to the consolidated database for applying and fetching changes, using your synchronization scripts.

Contention

The most important factor is to avoid database contention in your synchronization scripts. Just as with any other multi-client use of a database, you want to minimize database contention when clients are simultaneously accessing a database. Database rows that must be modified by each synchronization can increase contention. For example, if your scripts increment a counter, then updating that counter can be a bottleneck.

The figure below shows the following:

- a pool of connections to the consolidated database, shown as C1 to C*n*
- a number of synchronization requests, shown as S1 to S*n*
- MobiLink worker threads, shown as W1 to W*n*



If there are more synchronization requests than worker threads, the excess requests are queued until a worker thread becomes available after completing a synchronization. You can control the number of worker threads and connections, but MobiLink will always ensure that there is at least one connection per worker thread. If there are more connections than worker threads, the excess connections will be idle. Excess connections may be useful with multiple script versions, as discussed below.

Number of worker threads

Other than contention in your synchronization scripts, the most important factor for synchronization throughput is the number of worker threads. The number of worker threads controls how many synchronizations can proceed simultaneously.

Testing is vital to determine the optimum number of worker threads.

Increasing the number of worker threads allows more overlapping synchronizations, and increased throughput, but it will also increase resource and database contention between the overlapping synchronizations, and increase the time for individual synchronizations. As the number of worker threads is increased, the benefit of more simultaneous synchronizations becomes outweighed by the cost of longer individual synchronizations, and adding more worker threads decreases throughput. Experimentation is required to determine the optimal number of worker threads for your situation, but the following may help to guide you.

For uploads, performance testing shows that the best throughput happens with a relatively small number of worker threads: in most cases, three to ten worker threads. Variation depends on factors like the type of consolidated database, data volume, database schema, the complexity of the synchronization scripts, and the hardware used. The bottleneck is usually due to contention between worker threads executing the SQL of your upload scripts at the same time in the consolidated database.

For downloads, the optimum number of worker threads depends on the client to MobiLink bandwidth and the processing speed of clients. For slower clients, more worker threads are needed to get optimal download performance. This is because downloads involve more client processing and less consolidated database processing than uploads.

For Adaptive Server Anywhere clients, leaving download acknowledgement off (and not disabling the optional download buffering) can reduce the optimal number of worker threads for download, because worker threads do not have to wait for clients to apply downloads. There is little effect for UltraLite clients since UltraLite clients apply the download as it is received, without buffering.

☞ For more information on disabling the download acknowledgement, see the “SendDownloadACK (sa) extended option” [*MobiLink Clients*, page 131].

To get both the best download throughput and the best upload throughput, MobiLink provides two options. You can specify a total number of worker threads to optimize downloads. You can also limit the number that can simultaneously apply uploads to optimize upload throughput.

The -w option controls the total number of worker threads. The default is five.

The -wu option limits the number of worker threads that can simultaneously apply uploads to the consolidated database. By default, all worker threads can apply uploads simultaneously, but that can cause severe contention in the consolidated database. The -wu option lets you reduce that contention while still having a larger number of worker threads to optimize downloads and receive uploads. The -wu option only has an effect if the number is less than the total number of worker threads.

☞ For more information, see "-w option" on page 212 and "-wu option" on page 214.

MobiLink database connections

MobiLink creates a database connection for each worker thread. You can use the -cn option to specify that MobiLink create a larger pool of database connections, but any excess connections will be idle unless MobiLink needs to close a connection or use a different script version.

There are two cases where MobiLink will close a database connection and open a new one. The first case is if an error occurs. The second case is if the client requests a synchronization script version, and none of the available connections have already used that synchronization version.

> **Note**
> Each database connection is associated with a script version. To change the version, the connection must be closed and reopened.

If you have more than one synchronization version, you may want to set the maximum number of pooled connections to be larger than the number of worker threads, which is the default number. Then MobiLink will not need to close and open a new database connection each time a different synchronization version is requested.

If you routinely use more than one script version, you can reduce the need for MobiLink to close and open connections by increasing the number of connections. You can eliminate the need completely if the number of connections is the number of worker threads times the number of versions.

An example of tuning MobiLink for two script versions is given in the command line below:

```
dbmlsrv9 -c "dsn=ASA 9.0 Sample" -w 5 -cn 10
```

Since the maximum usable number of database connections is the number of script versions times the number of worker threads plus one, you can set -cn to 10 to ensure that database connections are not closed and opened to accommodate synchronization versions.

An example of tuning MobiLink for three script versions is:

```
dbmlsrv9 -c "dsn=ASA 9.0 Sample" -w 7 -cn 21
```

☞ For more information on setting the number of connections for any number of script versions, see "-cn option" on page 197.

Monitoring MobiLink performance

There are a variety of tools available to help you monitor the performance of your synchronizations.

The MobiLink Monitor is a graphical tool for monitoring synchronizations. It allows you to see the time taken by every aspect of the synchronization, sorted by MobiLink user or by worker thread.

☞ For more information, see "MobiLink Monitor" on page 117.

In addition, there are a number of MobiLink scripts that are available for monitoring synchronizations. These scripts allow you to use performance statistics in your business logic. You may, for example, want to store the performance information for future analysis, or alert a DBA if a synchronization takes too long. For more information, see

- ♦ "download_statistics connection event" on page 378
- ♦ "download_statistics table event" on page 381
- ♦ "synchronization_statistics connection event" on page 445
- ♦ "synchronization_statistics table event" on page 448
- ♦ "time_statistics connection event" on page 450
- ♦ "time_statistics table event" on page 453
- ♦ "upload_statistics connection event" on page 469
- ♦ "upload_statistics table event" on page 472

CHAPTER 7

MobiLink Monitor

About this chapter

The MobiLink Monitor is a tool for monitoring MobiLink synchronizations. This chapter describes how to use the MobiLink Monitor.

☞ For a hands-on description of how to use the Monitor, see "Tutorial: Writing MobiLink Scripts and Monitoring Synchronizations" [*MobiLink Tutorials*, page 13].

Contents

Introduction

The MobiLink Monitor is a MobiLink administration tool that provides you with detailed information about the performance of your synchronizations.

When you start the Monitor and connect it to a MobiLink synchronization server, the Monitor begins to collect statistical information about all synchronizations that occur in that monitoring session. The Monitor continues to collect data until you disconnect it or shut down the MobiLink server.

You can view the data in tabular or graphical form in the Monitor interface. You can also save the data in binary format for viewing with the Monitor later, or in .csv format to open in another tool, such as Microsoft Excel; or you can export it to an ODBC data source such as a MobiLink-supported relational database.

Monitor output allows you to see a wide variety of information about your synchronizations. For example, you can quickly identify synchronizations that result in errors, or that meet other criteria that you specify. You can identify possible contention in synchronization scripts by checking to see if synchronizations of differing durations have phases that end around the same time (because synchronizations are waiting for a previous phase to finish before they can continue).

The MobiLink Monitor can be used routinely in development and production, because monitoring does not degrade performance, particularly when the Monitor is run on a different computer from the MobiLink synchronization server.

Starting the MobiLink Monitor

If synchronization is already occurring when the MobiLink Monitor is started, the Monitor must wait until a worker thread is free before it can start monitoring. Therefore, you may want to start the Monitor before starting synchronizations. Once the Monitor is running it does not use a MobiLink worker thread.

You can have one instance of the Monitor running for each MobiLink synchronization server.

❖ To start monitoring data

1. From the Start menu, choose Programs ➤ SQL Anywhere 9 ➤ MobiLink ➤ MobiLink Monitor.

   Alternatively, you can type **dbmlmon** at a command prompt. For details, see below.

2. Start your consolidated database and MobiLink synchronization server, if they are not already running.

3. In the MobiLink Monitor, choose Monitor ➤ Connect to MobiLink Server.

   The Connect to MobiLink Server dialog appears.

   A Monitor connection starts like a synchronization connection to the MobiLink synchronization server. For example, if you started the MobiLink server with -zu+ then it doesn't matter what user ID you use here. For all MobiLink Monitor sessions, the script version is set to for_ML_Monitor_only.

   The Connect to MobiLink Server dialog should be completed as follows:

   ♦ **Host** is the computer where the MobiLink synchronization server is running. By default, it is the computer where the Monitor is running.

   ♦ **Network Protocol** should be set to the same protocol and port as the MobiLink synchronization server is using for synchronization requests.

   ♦ **Additional Network Parameters** allows you to set optional parameters. You can set the following parameters, separated by semi-colon if you need to specify multiple parameters:

     • **buffer_size=***number* (HTTP and HTTPS only)
     • **client_port=***nnnn*
     • **client_port=***nnnn-mmmmm*
     • **persistent=**{**0**|**1**}
     • **proxy_host=***proxy_hostname* (HTTP and HTTPS only)

- **proxy_port=***proxy_portnumber* (HTTP and HTTPS only)
- **url_suffix=***suffix* (HTTP and HTTPS only)
- **version=***versionnumber* (HTTP and HTTPS only)

☞ For more information about these network parameters, see "Network protocol options for UltraLite synchronization clients" [*MobiLink Clients*, page 341].

4. Start synchronizing.

   The data appears in the Monitor as it is collected.

Starting dbmlmon on the command line

Command line options allow you to have the Monitor open a file or connect to a MobiLink synchronization server on startup. Following is the syntax:

dbmlmon [*connect-options* | *inputfile*.{ **mlm** | **csv** } | **-?**]

where:

connect-options can be one or more of the following:

-u *ml_username*

-p *password*

-x { **tcpip** | **http** | **https** } [**(** *keyword*=*value*;... **)**]

The keyword=value pairs can be the host, protocol, and Additional Network Parameters as described above.

-o *outputfile*.{ **mlm** | **csv** }

The -o option closes the Monitor at the end of the connection and saves the session in the specified file.

-? You can type **dbmlmon -?** to view the dbmlmon syntax.

Note: The options -u and -x are required to connect.

❖ To stop the MobiLink Monitor

1. In the Monitor, choose Monitor ➤ Disconnect from MobiLink Server. This stops the collection of data.

   You can also stop collecting data by shutting down the MobiLink synchronization server or closing the Monitor.

   Before closing the Monitor, you can save the data for the session. For more information, see "Saving Monitor data" on page 126.

2. When you are ready to close the Monitor, choose File ➤ Close.

Using the MobiLink Monitor

The Monitor has three panes:

- **Details Table** is the top pane. It is a spreadsheet that shows the total time taken by each synchronization, with a breakdown showing the amount of time taken by each part of the synchronization.

  ☞ For more information, see "Details Table pane" on page 121.

- **Chart** is the middle pane. It provides a graphical representation of the data. The scale at the bottom of this pane represents time. You can select the data that is displayed in the Chart by drawing a box around data in the Overview pane; or by choosing View ➤ Go To.

  ☞ For more information, see "Chart pane" on page 123.

- **Overview** is the bottom pane. It shows an overview of all the data. To choose data to see in the Chart, click in the Overview and draw a box. The Chart will show everything that is located in the box.

  ☞ For more information, see "Overview pane" on page 124.

In addition, there is an Options dialog that you can use to customize the data, and properties dialogs for viewing more detailed information. For more information, see

- "Options dialog" on page 124
- "Session properties" on page 124
- "Synchronization properties" on page 125

Details Table pane

The Details Table provides information about the duration of each part of the synchronization. All times are measured by the MobiLink synchronization server. Some times may be non-zero even when you do not have the corresponding script defined.

You can choose the columns that appear in the Details Table pane by opening Tools ➤ Options and then opening the Table tab. For a description of the statistics that are available, see "MobiLink statistical properties" on page 130.

The following columns appear by default:

- **Sync** Identifies each synchronization.

- **Worker** Identifies the MobiLink worker thread that carried out the synchronization. The worker is identified as *n.m*, where *n* is the stream number and *m* is the thread number.
- **User** Identifies the synchronization user.
- **Version** The version of the synchronization script.

  ☞ For information about script versions, see "Script versions" on page 239.
- **Start_Time** The date and time when the MobiLink synchronization server started the synchronization. (This may be later than when the synchronization was requested by the client.)
- **Duration** The total duration of the synchronization, in seconds.
- **Verify_Upload** The time in seconds for MobiLink to validate the synchronization request, validate the user name, and validate the password (if your synchronization setup requires authentication).
- **Preload_Upload** The time in seconds for MobiLink to receive the uploaded data from the client.
- **Begin_Sync** The time in seconds to run your begin_synchronization script, if one was run.
- **Upload** The time in seconds to apply the upload to the consolidated database. This is the time between the begin_upload script and the end_upload script.
- **Prepare_for_Download** The time in seconds to run your prepare_for_download script, if one was run.
- **Download** The time in seconds to download the data. This is the time between the begin_download script and the end_download script. If download acknowledgement is enabled, this includes the time to apply the download on the remote database and return acknowledgement.
- **End_Sync** The time in seconds to run the end_synchronization script, if one was run.

To sort the table by a specific column, click on the column heading. If new data is appearing in the Monitor, it will be sorted as it is added.

You can close the Details Table pane by clearing View ➤ Details Table.

Chart pane

The Chart pane presents the same information as the Details Table, but in graphical format. The bars in the Chart represent the length of time taken by each synchronization, with subsections of the bars representing the phases of the synchronization.

Viewing data

Click a synchronization to select that synchronization in the Details Table.

Double-click a synchronization to open the Synchronization Session Properties for the synchronization. For more information, see "Synchronization properties" on page 125.

Grouping data by thread or user

You can group the data by worker thread or by user. Choose View ➤ By User or View ➤ By Worker Thread.

Zooming in on data

There are several ways to select the data that is visible:

- **Zoom options** There are zoom options in the View menu and zoom buttons on the toolbar that allow you to zoom in and out. To have a synchronization fill the available space, use Zoom to Selection.
- **Scrollbar** Click the scrollbar at the bottom of the Chart pane and slide it.
- **Go To dialog** Open this dialog by choosing View ➤ Go To. The Go To dialog appears.

  Start Date & Time lets you specify the start time for the data that appears in the Chart pane. If you change this setting, you must specify at least the year, month, and date of the date-time.

  Chart Range lets you specify the duration of time that is displayed. The chart range can be specified in milliseconds, seconds, minutes, hours, or days. The chart range determines the granularity of the data: a smaller length of time means that more detail is visible.
- **Overview Pane** The box in the Overview pane indicates the area being displayed in the Chart. It allows you to quickly select a portion of data to view. You can easily resize or move the box to see different data, or see data at different granularity. If you make the box smaller you shorten the interval of the visible data in the Chart, which makes more detail visible. Click to move the current box without changing the zoom. Drag in the Overview to redraw the box and select a different zoom and position.

Time axis

At the bottom of the Chart pane there is a scale showing time periods. The format of the time is readjusted automatically depending on the span of time that is displayed. You can always see the complete date-time by hovering your cursor over the scale.

Default color scheme

You can view or set the colors in the Chart pane by opening the Options dialog (available from the Tools menu). The default color scheme for the Chart pane uses green for uploads, red for downloads, and blue for begin and end phases, with a darker shade for earlier parts of a phase.

☞ For information about setting colors, see "Options dialog" on page 124.

Overview pane

The Overview pane shows you an overview of the entire Monitor session. The area that is currently displayed in the Chart pane is represented as a box in the Overview. Click in the Overview pane to move the box (and thus move the start time of the data shown in the Chart) or drag in the Overview to redraw the box to change the box's location and size (and thus change the start time and the range of data)

You can separate the Overview pane from the rest of the Monitor window. In the Options dialog, open the Overview tab and clear the Keep Overview Window Attached to Main Window checkbox.

☞ For more information, see "Options dialog" on page 124.

You can close the Overview pane by clearing View ➤ Overview Pane.

Options dialog

Options allow you to specify a number of settings, including colors and patterns for the graphical display in the Chart pane (the middle pane of the MobiLink Monitor) and the Overview pane (the bottom pane).

To open the Options dialog, open the Monitor and choose Tools ➤ Options.

Restoring defaults

To restore default settings, delete the file *.mlMonitorSettings*. This file is stored in your user profiles directory.

Session properties

The Session Properties dialog provides basic information about the monitoring session.

To open the Session Properties dialog, open the Monitor and choose File ➤ Properties.

Synchronization properties

Double-click a synchronization in either the Details Table or the Chart to see properties for that synchronization.

You can choose to see statistics for all tables (which is the sum for all tables in the synchronization), or for individual tables. The dropdown list provides a list of the tables that were involved in the synchronization.

☞ For an explanation of the statistics in Synchronization Properties, see "MobiLink statistical properties" on page 130.

Saving Monitor data

You can save the data from a Monitor session as a binary file (.mlm), as a text file with comma-separated values (.csv), as tables in a relational database, or as a Microsoft Excel file.

Saving to file

To save the data as a file, choose File ➤ Save As.

- ♦ Save the data as a binary (.mlm) file if you want to view the saved data in the MobiLink Monitor. To reopen, choose File ➤ Open.
- ♦ Save the data as a comma separated file (.csv) if you want to view it in another tool, such as Microsoft Excel. This will save all the information in the session and synchronization property sheets, except per table information and the session begin and end time. You can also open a .csv file in the Monitor.

  In the .csv file format, time durations are stored in milliseconds.

You can specify that you want data to be saved automatically to a file. To do this, choose Tools ➤ Options, and enter an output file name on the General tab. The output file is overwritten by new data.

Exporting to a relational database or Excel

You can also export Monitor data using an ODBC or JConnect connection. With JConnect, you can only export to Adaptive Server Anywhere or Adaptive Server Enterprise databases. With ODBC, you can export to any relational database that is supported by MobiLink, as well as to Excel.

When you export data, all the columns in your Monitor session are exported, as well as a column called export_time that identifies the time the export was performed.

The data source must have quoted identifiers enabled, because some of the columns are reserved words. The Monitor enables quoted identifiers automatically for Adaptive Server Anywhere, Adaptive Server Enterprise, and Microsoft SQL Server databases. If quoted identifiers is not enabled, the export will fail.

❖ To export the data to a database or Excel

1. After collecting Monitor information, disconnect from the MobiLink synchronization server.
2. In the MobiLink Monitor, choose File ➤ Export to Database.

   The Export to Database dialog appears.
3. Select options for the output.

- ♦ You can name the two tables that will be created to hold the data, or use the defaults (mlm_by_sync and mlm_by_table). If the tables do not exist, they will be created by the Monitor. For Excel output, the two table names identify the two worksheets that are created.
- ♦ Choose whether you want to overwrite data in existing tables. If you do not choose to overwrite the data, new data is appended to existing data.

4. Click OK.

   A Connect dialog appears that allows you to connect to the database or Excel spreadsheet using ODBC or JConnect.

Customizing your statistics

The Watch Manager allows you to visibly distinguish synchronizations that meet criteria that you specify. For example, you might want to highlight big synchronizations, long synchronizations, small synchronizations that take a long time, or synchronizations that receive warnings.

To open the Watch Manager, open the Monitor and then click Tools ➤ Watch Manager.

The left pane of the Watch Manager contains a list of all available watches. The right pane contains a list of active watches. To add or remove a watch from the active list, select a watch in the left pane and click the appropriate button.

There are three predefined watches (Active, Completed, and Failed). You can edit predefined watches to change the way they are displayed, and you can deactivate them by removing them from the right pane.

No synchronizations are displayed in the Chart unless they meet the conditions of a watch. If you disable all watches (by removing them from the Current Watches list), then no synchronizations are shown in the Chart or Overview.

The order of watches in the right pane is important. Watches that are closer to the top of the list are processed first. Use the Move Up and Move Down buttons to organize the order of watches in the right pane.

You can use the predefined watches, and create other watches. To edit a watch condition, remove it and then add the new watch condition.

❖ To create a new watch

1. In the Watch Manager, click New.

   The New Watch dialog appears.

2. Give the watch a name in the Name box.

3. Select a property, comparison operator, and value.

   ☞ For a complete list of properties, see "MobiLink statistical properties" on page 130.

4. Click Add. (You must click Add to save the settings.)

5. If desired, select another property, operator, and value, and click Add.

6. Select a pattern for the watch in the Chart pane. (The Chart pane is the middle pane in MobiLink Monitor.)

7. Select a color for the watch in the Overview pane. (The Overview pane is the bottom pane in the MobiLink Monitor.)

MobiLink statistical properties

Following is a list of the properties that are available in the MobiLink Monitor. These statistics can be viewed in the New Watch dialog, the Details Table pane, or the Synchronization Properties. In Synchronization Properties, the property names do not contain underscores.

☞ For more information about the New Watch dialog, see "Customizing your statistics" on page 128.

☞ For more information about the Details Table, see "Details Table pane" on page 121.

☞ For more information about the Synchronization Properties dialog, see "Synchronization properties" on page 125.

Property	Notes
active	True if the synchronization is in progress.
begin_sync	Time for the begin_synchronization event.
completed	True if the synchronization completed successfully.
conflicted_deletes	Number of uploaded deletes for which conflicts were detected.
conflicted_inserts	Number of uploaded inserts for which conflicts were detected.
conflicted_updates	Number of uploaded updates for which conflicts were detected.
connection_retries	Number of times the MobiLink synchronization server retried the connection to the consolidated database.
download	Time for the download.
download_bytes	Bytes downloaded to the synchronization client.
download_deleted_rows	Number of row deletions fetched from the consolidated database by the MobiLink synchronization server (using download_delete_cursor scripts).
download_errors	Number of errors that occurred during the download.

Property	Notes
download_fetched_rows	Number of rows fetched from the consolidated database by the MobiLink synchronization server (using download_cursor scripts).
download_filtered_rows	Number of fetched rows that were not downloaded to the MobiLink client because they matched rows that the client uploaded.
download_warnings	Number of warnings that occurred during the download.
duration	Total time for the synchronization, as measured by the MobiLink synchronization server. This does not include time when the synchronization request is queued waiting for an available worker thread.
end_sync	Time for the end_synchronization event.
ignored_deletes	Number of uploaded deletes that were ignored.
ignored_inserts	Number of uploaded inserts that were ignored.
ignored_updates	Number of uploaded updates that were ignored.
preload_upload	Time for the transfer of the upload data from the client to the MobiLink synchronization server.
prepare_for_download	Time for the prepare_for_download event.
start_time	Date-time (in ISO-8601 extended format) for the start of the synchronization. All fields of the format must be specified: *YYYY-MM-DD hh*:*mm*:*ss.sss* or *YYYY-MM-DD hh*:*mm*:*ss,sss*, depending on your locale setting.
sync	A number uniquely identifying the synchronization within the Monitor session.
sync_deadlocks	Total number of deadlocks in the consolidated database that were detected for the synchronization.
sync_errors	Total number of errors that occurred for the synchronization.

Property	Notes
sync_tables	Number of client tables that were involved in the synchronization.
sync_warnings	Total number of warnings that occurred for the synchronization.
upload	Time for data to be uploaded to the consolidated database.
upload_bytes	Number of bytes uploaded from the synchronization client.
upload_deadlocks	Number of deadlocks in the consolidated database that were detected during the upload.
upload_deleted_rows	Number of row deletions that were uploaded from the synchronization client.
upload_errors	Number of errors that occurred during the upload.
upload_inserted_rows	Number of row insertions that were uploaded from the synchronization client.
upload_updated_rows	Number of row updates that were uploaded from the synchronization client.
upload_warnings	Number of warnings that occurred during the upload.
user	Name of the MobiLink client.
verify_upload	Time for verifying the synchronization protocol and authenticating the synchronization client.
version	Name of the synchronization version.
worker	Identifier for the MobiLink worker thread used for the synchronization in the form *n.m*, where *n* is the stream number and *m* is the thread number.

CHAPTER 8

Synchronizing Through a Web Server With the Redirector

About this chapter

This chapter describes how to route MobiLink synchronization through a web server, focusing on the Redirector.

Contents

Introduction

MobiLink includes a web server extension called the **Redirector** that routes requests and responses between a client and the MobiLink synchronization server. A plug-in such as this is also commonly called a **reverse proxy**.

The main reason for routing requests through a web server is to use existing web server and firewall configurations for HTTP or HTTPS synchronization. However, a web server can operate as a proxy without the Redirector. The Redirector is most useful when you have more than one MobiLink synchronization server.

☞ For more information, see "Options when using a web server" on page 135.

Using the Redirector, you can configure your web server to route specific URL requests to one or more computers running the MobiLink synchronization server.

Web servers can be configured to pass requests with specific URLs or ranges of URLs to extension programs commonly written in the form of perl CGI scripts, DLLs, or other extension mechanisms. These extension programs may access external data sources and provide responses for the web server to deliver to its clients.

Load balancing and failover

The Redirector implements load balancing and failover using a simple round robin algorithm (servers are chosen in a fixed cyclic order). Each MobiLink synchronization server is tested at set intervals and requests are no longer sent to a server that is not responding. The Redirector detects when a MobiLink synchronization server is running again and resumes sending requests at that time.

HTTPS synchronization

In HTTPS synchronization, HTTP headers are encrypted over SSL/TLS using RSA encryption before being sent to or from the server. HTTPS is only used for the connection between the MobiLink client and the web server. The web server decrypts the HTTPS and sends HTTP to MobiLink via the Redirector.

The HTTPS protocol is slower than other secure protocols, so it is recommended that it be used only if the HTTPS protocol is required.

Supported web servers

Plug-ins are provided for the following web servers:

Redirector plug-in	...supports
ISAPI Redirector	Microsoft web servers
NSAPI Redirector	Sun One (previously Netscape) web servers on Windows
Servlet Redirector	Web servers that support the Java Servlet API 2.3, including Apache Tomcat and Sun One web servers on UNIX
Native Apache Redirector	Apache web server
M-Business Anywhere Redirector	M-Business Anywhere web server

Options when using a web server

The Redirector is one way to route MobiLink synchronization through a web server. The Redirector is particularly useful for synchronizing across a firewall or with multiple MobiLink synchronization servers.

The main reason for routing requests through a web server is to use existing web server and firewall configurations for HTTP or HTTPS synchronization. The Redirector is most useful when you have more than one MobiLink synchronization server.

You can also route synchronizations through a web server without using the Redirector. In this case, you might configure your web server as a proxy to route synchronizations to a MobiLink server. For more information on how to do this with your web server, see your web server documentation.

The following table contains recommendations to help you decide how best to route your MobiLink synchronizations.

	Direct connection possible	Direct connection not possible
One MobiLink synchronization server	Use TCP/IP instead of HTTP	Use an HTTP or HTTPS proxy to pass messages through the web server to the MobiLink synchronization server
Multiple MobiLink synchronization servers	Use the Redirector with HTTP or HTTPS	Use the Redirector with HTTP or HTTPS

☞ See "Synchronizing Through a Web Server With the Redirector" on page 133.

Setting up the Redirector

The following sections describe how to configure your web server to manage synchronization requests.

❖ Overview of the configuration process

1. Configure MobiLink clients and the MobiLink synchronization server.

   ☞ See "Configuring MobiLink clients and servers for the Redirector" on page 137.

2. Ensure that the Redirector configuration file is on the same computer as the web server.

   ☞ See "Configuring Redirector properties (all versions)" on page 139.

3. Modify the Redirector configuration file.

   ☞ See "Configuring Redirector properties (all versions)" on page 139.

4. Perform web server-specific configuration.

   ☞ See one of the following:

   ♦ "NSAPI Redirector for Netscape/Sun web servers" on page 141

   ♦ "ISAPI Redirector for Microsoft web servers" on page 144

   ♦ "Servlet Redirector" on page 146

   ♦ "Apache Redirector" on page 150

   ♦ "M-Business Anywhere Redirector" on page 153

Configuring MobiLink clients and servers for the Redirector

This section describes how to configure MobiLink clients and the MobiLink synchronization server for synchronization through a web server. The following procedures set the parameters required for requests directed through web servers.

MobiLink clients

❖ To configure MobiLink clients (Adaptive Server Anywhere and UltraLite)

1. Specify the communication type for MobiLink clients to HTTP or HTTPS.

   ☞ For more information about setting the communication type for Adaptive Server Anywhere clients, see "CommunicationType (ctp) extended option" [*MobiLink Clients*, page 111].

   ☞ For more information about setting the communication type for UltraLite clients, see "HTTP protocol options" [*MobiLink Clients*, page 346] or "HTTPS protocol options" [*MobiLink Clients*, page 347].

2. Specify the following HTTP/HTTPS synchronization protocol options for MobiLink clients:

   ♦ **host** the name or IP address of the web server.

   ♦ **port** the web server port accepting HTTP or HTTPS requests.

   ♦ **url_suffix** This setting depends on the type of web server you are using:

   - For ISAPI web servers:

     ```
     exe_dir/iaredirect.dll/ml/
     ```

     where *exe_dir* is the location of *iaredirect.dll.*

   - For NSAPI web servers:

     ```
     mlredirect/ml/
     ```

     where *mlredirect* is a name mapped in your *obj.conf* file.

   - For servers that support the Java Servlet API 2.2, including Apache with Tomcat using the servlet Redirector:

     ```
     iaredirect/ml/
     ```

   - For the native Redirector for Apache, set this to whatever you chose in the Redirector's <location> tag in the *httpd.conf* file. For example:

```
iaredirect/ml/
```

☞ For more information about setting protocol options for UltraLite clients, see "HTTP protocol options" [*MobiLink Clients*, page 346] or "HTTPS protocol options" [*MobiLink Clients*, page 347].

☞ For more information about setting protocol options for Adaptive Server Anywhere clients, see "CommunicationAddress (adr) extended option" [*MobiLink Clients*, page 106].

MobiLink synchronization server

❖ To configure MobiLink servers

1. The MobiLink synchronization server must be started with the HTTP protocol to use HTTP or HTTPS for communication between the client and the proxy. The Redirector cannot use HTTPS directly.

   For example, the HTTP protocol may be specified on the dbmlsrv9 command line as follows:

   dbmlsrv9 -x http

   ☞ For more information, see "-x option" on page 214.

2. In addition, you may want to set the following parameters for the MobiLink synchronization server:

   ♦ **port** for the HTTP protocol, MobiLink defaults to port 80. For the HTTPS protocol, MobiLink defaults to port 443. If the MobiLink synchronization server is running on the same machine as the web server, port 80 is normally in use by the web server. If this is the case you must specify a different port. For example, you could use port 2439, which is the Internet Assigned Numbers Authority (IANA)-registered port number for the MobiLink synchronization server.

   ♦ **contd_timeout** This is the number of seconds to wait to receive the next part of a partially completed synchronization before the synchronization is abandoned. This setting is optional and has a default value of 30 seconds.

   You may wish to increase the timeout parameters if your applications involve large synchronizations over slow networks.

   ☞ For more information about port and contd_timeout, see "-x option" on page 214.

Configuring Redirector properties (all versions)

This section describes generic web server configuration steps to configure Redirector properties.

❖ To configure Redirector properties

1. Complete the steps in "Configuring MobiLink clients and servers for the Redirector" on page 137.

2. Copy *redirector.config* to the web server.

   The file *redirector.config* is provided with the MobiLink synchronization server installation, in the *MobiLink\redirector* subdirectory of your SQL Anywhere installation.

   If the MobiLink synchronization server is not installed on the same computer as the web server, copy *redirector.config* to the computer that holds the web server.

   For Microsoft web servers, copy *redirector.config* to the directory *Inetpub/scripts*. For other web servers, you can copy *redirector.config* to any directory.

3. Configure the Redirector configuration file.

   To configure communications between the web server and MobiLink synchronization server, you must edit the file *redirector.config* on the computer that holds the web server.

   You can set the following directives in this file:

   ♦ **LOG_LEVEL** used to control the amount of output written to the log file. Values are 0, 1, and 2, with 1 being the default and 2 generating the most output. For the Apache Redirector, this setting has no effect; set the log level in the LogLevel section of the Apache configuration file, *httpd.conf*.

   ♦ **ML** used to list the computers running MobiLink synchronization server, in the form `ML=host:port`. ML is case sensitive.

   ♦ **ML_CLIENT_TIMEOUT** used to ensure that each step of a single synchronization is directed to the same MobiLink synchronization server. The default value is 600 seconds (ten minutes).

     Information is maintained by the MobiLink synchronization server for the duration of a synchronization, so each step of a synchronization should be handled by the same server. The Redirector maintains an association between client and server for the duration of ML_CLIENT_TIMEOUT. The value of this parameter should be greater than the longest step in any user's synchronization.

- ♦ **SLEEP** used to set the interval in seconds at which the Redirector checks that the servers are functioning. The default is 1800 (30 minutes). For example, `SLEEP=3600`. SLEEP is case sensitive.

  The following rules apply to *redirector.config*:

  - The maximum line length is 300 characters.
  - Comments start with the hash character (#).
  - You cannot include spaces or tabs in the directive definitions.

4. Complete web server-specific configuration in one of the following sections:
   - ♦ "ISAPI Redirector for Microsoft web servers" on page 144
   - ♦ "NSAPI Redirector for Netscape/Sun web servers" on page 141
   - ♦ "Servlet Redirector" on page 146
   - ♦ "Apache Redirector" on page 150

Example

Following is a sample *redirector.config* file. This file specifies the following:

- ♦ The Redirector should check every 1800 seconds that the servers are functioning.

- ♦ The three computers running MobiLink synchronization servers that are able to process requests. When you specify multiple servers, load balancing is automatically enabled.

```
SLEEP=1800
ML=myServ-pc:80
ML=209.123.123.1:8080
ML=myCompany.com:8081
```

NSAPI Redirector for Netscape/Sun web servers

The NSAPI Redirector is provided for the Sun One web server, which was previously the Netscape iPlanet Enterprise Edition web server. Following are setup instructions for Sun One, service pack 1.

This Redirector works only on Windows. To use the Redirector with Netscape/Sun web servers on UNIX, you can use the servlet Redirector. See "Servlet Redirector" on page 146.

❖ To configure the NSAPI Redirector for Sun One

1. Complete the steps in "Configuring Redirector properties (all versions)" on page 139.

2. If necessary, copy the file *iaredirect.dll* to the computer that holds the web server. This file is installed with the MobiLink synchronization server, in the *MobiLink\redirector\nsapi* subdirectory of your SQL Anywhere installation.

3. Update the Sun One web server configuration file *obj.conf* as follows.

   Sample file provided
   A complete sample copy of *obj.conf*, preconfigured for the MobiLink synchronization server, is provided in *MobiLink\redirector\nsapi*, and is called *obj.conf.example*. You can use this sample file to confirm where the following sections fit in to the file.

   Update the following sections of the files *magnus.conf* and *obj.conf*.

   ♦ In *magnus.conf*, specify where *iaredirect.dll* and *redirector.config* are located.

     At the end of the Init section, add the following text, where *location* is the actual location of the files. (*iaredirect.dll* and *redirector.config* can be in different locations, although both must be on the same computer as the web server.)

     ```
     Init fn="load-modules" shlib="location/iaredirect.dll"
     funcs="redirector,initialize_redirector"
     Init fn="initialize_redirector"
               configFile="location/redirector.config"
     ```

   ♦ In *obj.conf*, specify the name of the Redirector to be used in URLs.

     At the beginning of the "default object" section, add the following text. This section should appear exactly as provided below, except that you can change *mlredirect* to whatever you wish. All requests of the form *http://host:port/mlredirect/ml/** will be sent to one of the MobiLink synchronization servers running with the Redirector.

```
<Object name=default>
NameTrans fn="assign-name" from="/mlredirect/ml/*"
         name="redirectToML"
```

- In *obj.conf*, specify the objects that are called by the Redirector. After the “default object” section, add the following section:

  ```
  <Object name="redirectToML">
  Service fn="redirector" serverType="ml"
  </Object>
  ```

4. Set the buffer size for the MobiLink upload streams.

   Add a directive to your web server’s *magnus.conf* file to set the buffer size (in bytes) for the upload and download stream. For example:

   ```
   ChunkedRequestBufferSize 2000000
   ```

   This directive increases the buffer to 2 Mb. The value must be sufficient to accommodate the size of the uploaded data.

   If you are using any network protocol other than HTTPS, your configuration is complete.

5. If you are using HTTPS synchronization, configure your server as follows:

   - Start the Sun One web server Administration Server.

     Choose Start ➤ Programs ➤ iPlanet Web Server ➤ Start iWS Administration Server.

   - Log in to the Administration Server.

     Choose Start ➤ Programs ➤ iPlanet Web Server ➤ Administer Web Server.

     When prompted, enter your user ID and password.

   - On the Servers tab, select your server from the list and click Manage.
   - On the Security tab, click Request a Certificate.
   - Generate a certificate request and have it signed by a certificate authority or using gencert, which requires a separate license.

     - To have the certificate request signed by a certificate authority, fill out the form.
     - To use the gencert utility, fill out the form, supplying your own e-mail address instead of the e-mail address of a certificate authority. Save the text of the certificate request to a file, then run the gencert utility. For more information, see “Certificate generation utility” on page 496.

   - On the Security tab, click Install Certificate. Fill out the form and specify the location of your signed certificate.

- ♦ Click Manage Certificates to verify that your certificate has been installed correctly.
- ♦ On the Preferences tab, click Add Listen Socket. Specify the required parameters. The default port for HTTPS is 443. Select On from the Security dropdown list to activate HTTPS synchronization.

  ☞ For more information about using HTTPS, see "MobiLink Transport-Layer Security" on page 165.

Example

Following is an example of the section of *magnus.conf* that you need to customize.

```
Init fn="load-modules" shlib="D:/iaredirect.dll"
funcs="redirector,initialize_redirector"
Init fn=" initialize_redirector "
         configFile="D:/redirector.config"
```

Following is an example of the sections of *obj.conf* that you need to customize.

```
<Object name=default>
NameTrans fn="assign-name" from="/mlredirect/ml/*"
         name="redirectToML"
...
<Object name="redirectToML">
Service fn="redirector" serverType="ml"
</Object>
```

❖ To test your configuration

1. Call the Redirector using the following syntax:

   ```
   http://host:port/mlredirect/ml/
   ```

2. Check the log file to see if the Redirector logged a request.

   Note: This test does not make a connection to the MobiLink synchronization server.

ISAPI Redirector for Microsoft web servers

If you are using a Microsoft web server, you can use the ISAPI version of the Redirector. Following are setup instructions for IIS 5.0.

❖ To configure ISAPI Redirector for Microsoft web servers

1. Complete the steps in "Configuring Redirector properties (all versions)" on page 139.

2. Copy the file *iaredirect.dll* to *Inetpub/scripts* on the computer that holds the web server.

   The file *iaredirect.dll* is installed with the MobiLink synchronization server, in *MobiLink\redirector\isapi* under your SQL Anywhere directory.

   The directory *Inetpub/scripts* is in the Microsoft web server installation directory.

3. Copy the file *redirector.config* to *Inetpub/scripts* on the computer that holds the web server.

   If you are using any network protocol other than HTTPS, your configuration is complete. If your configuration is not successful, see the Note, below.

4. If you are using HTTPS synchronization, configure your server as follows:

   ♦ Right-click My Computer and select Manage from the popup menu.

   ♦ In the left pane, open the Services and Applications folder. Select Internet Information Services.

   ♦ In the right pane, right-click the default web site and select Configure from the popup menu.

   ♦ Click the Directory Security tab.

   ♦ Click Server Certificate.
   The Web Server Certificate wizard appears.

   ♦ Select Create a New Certificate to generate a certificate request. Follow the remaining prompts, choosing to output the certificate request to a file.

   ♦ Sign your certificate.
   You can sign the certificate using a third-party certificate authority or using the gencert utility, which requires a separate license. For more information, see "Certificate generation utility" on page 496.

- Click Server Certificate.

  The Web Server Certificate wizard appears with different prompts to allow you to install the signed certificate. Follow the prompts.
- Click View Certificate to verify that your certificate has been correctly installed.

  ☞ For more information about using HTTPS, see "MobiLink Transport-Layer Security" on page 165.

Note

The directory *Inetpub/scripts* should be created during the web server installation with execute permissions. You can put *redirector.config* and *iaredirect.dll* in a different directory only if you use Internet Information Services to give execute permissions to the directory.

If you are unable to connect to the MobiLink synchronization server, the problem may be that you do not have a virtual directory that points to the *Inetpub/scripts* directory. If this is the case, you must open Internet Information Services and manually create a virtual directory. This virtual directory should point to *Inetpub/scripts* and have Execute Permissions set to Scripts and Executables. See the IIS online help for instructions.

❖ To test your configuration

1. Call the ISAPI Redirector using the following syntax:

   protocol **://** *host* [**:** *port*] **/** *exec_dir* **/iaredirect.dll/ml/**

   where:

   - **protocol** is **http** or **https**.
   - **host** is the host name of the web server.
   - **port** is the port on which the web server is listening, if it is not the default port.
   - **exec_dir** is the directory where you installed the Redirector dll, *iaredirect.dll.* The default directory is *scripts.*

     For example,

     ```
     http://server:8080/scripts/iaredirect.dll/ml/
     ```

2. Check the log file to see if the Redirector logged a request.

   Note: This test does not make a connection to the MobiLink synchronization server.

Servlet Redirector

The servlet Redirector is supported for web servers that support the Java servlet specification version 2.3. The following procedure is an example for how to set up the servlet Redirector for Apache Tomcat 4.0.6.

☞ There is also a native Redirector for Apache web servers. For more information, see “Apache Redirector” on page 150.

To configure the servlet Redirector for Apache Tomcat

This section describes how to install the servlet version of the Redirector to work on an Apache web server in conjunction with the Tomcat servlet container. Testing of the Redirector software has been carried out using Tomcat version 4.0.6 and Apache 2.0.47.

Installation requires the following steps:

1. Complete the steps in “Configuring Redirector properties (all versions)” on page 139.
2. Install the servlet version of the Redirector in Tomcat.
3. Configure the Apache web server to run as a proxy.

This section uses *%CATALINA_HOME%* and *%APACHE_HOME%* as the root directory of your Tomcat and Apache installation respectively.

❖ To install the servlet Redirector in Tomcat

1. Install Tomcat as a standalone server.

   You can download Tomcat binaries from the Jakarta project on the Apache web site at *http://archive.apache.org/dist/jakarta/tomcat-4/archive/v4.0.6/bin/*.

2. Optionally, set the required Tomcat HTTP port.

   Tomcat binds to port 8080 by default. If there is a conflict, perhaps because another web server is using this port,
   - open the file: *%CATALINA_HOME%/conf/server.xml*
   - search for 8080 (which is in a <Connector> tag)
   - change it to a port that is not in use

3. Install the servlet Redirector as a web application.
   - Copy *iaredirect.war* file to *%CATALINA_HOME%/webapps*
   - Shut down and restart Tomcat.

     Tomcat expands the war file and creates the directory *iaredirect* for the Redirector web application.

- Edit the file *%CATALINA_HOME%/webapps/iaredirect/WEB-INF/web.xml*. Search for **redirector.config** (in an <init-param> tag), and correct the path for the *redirector.config* file.

  Change the entry **redirector.config** to read *drive*:/*path*/**redirector.config**. Even on Windows operating systems, use a forward slash as a path separator, as in *d:/redirector.config*.
- Shut down and restart Tomcat for the changes to take effect.

  Once the changes have taken effect, you no longer need the war file in the deployed location.
- The Redirector can now be invoked through the following URL:

  http://tc-host:tc-port/iaredirect/ml/

  where *tc-host* is the machine and *tc-port* the port on which Tomcat is listening.

❖ To configure the Apache web server as a proxy

1. Install the Apache web server.

   You can download binaries from the Apache web site at http://www.apache.org.

2. Optionally, change the Apache web server port.

   Edit the file *%APACHE_HOME%/conf/httpd.conf* and change the **Port** setting to the desired port.

3. Configure Apache to run as a proxy.

   In *%APACHE_HOME%/conf/httpd.conf*, add the following two directives:

   ```
   LoadModule proxy_module {module-path}/mod_proxy.so
   LoadModule proxy_connect_module {module-path}/mod_proxy_
            connect.so
   LoadModule proxy_http_module {module-path}/mod_proxy_
            http.so
   ```

   For example, the path may be *modules/mod_proxy.so* (the default).

4. Configure Apache to forward Redirector URLs to Tomcat.

   In *%APACHE_HOME%/conf/httpd.conf*, add the following two directives so that Apache forwards URLs of the form http://localhost/iaredirect/* to the Tomcat 4 Connector listening on port 8080:

   ```
   ProxyPass /iaredirect http://localhost:8080/iaredirect
   ```

The port number must match the port number used for Tomcat. If Tomcat and Apache are not running on the same machine, provide the machine name where Tomcat is running instead of **localhost**.

If you are using any network protocol other than HTTPS, your configuration is complete.

5. If you are using HTTPS synchronization, configure your server as follows. (Note that this HTTPS configuration is identical to the HTTPS instructions for the native Redirector for Apache and M-Business Anywhere.)

   - ♦ Download and install binaries for mod_ssl and OpenSSL. You can find them using the Apache Module Registry at *http://modules.apache.org/*. *mod_ssl.so* must be copied to *%APACHE_HOME%\modules. libeay32.dll* and *ssleay32.dll* must be copied to *%APACHE_HOME%\bin.*
   - ♦ Generate a server certificate and private key either by generating a request with reqtool.exe and sending it to a third party certificate authority to sign it, or by generating a certificate directly using gencert.exe. The private key can either be in the same file as the server certificate or in its own file.
   - ♦ Add the following lines to *%APACHE_HOME%\conf\httpd.conf*:

     ```
     LoadModule ssl_module modules/mod_ssl.so
        SSLEngine on
        SSLCertificateFile certificate_file
     ```

     where *certificate_file* is the path and file name of the server's certificate file.

     If the server's private key is in a separate file from the server's certificate, add the additional line

     ```
     SSLCertificateKeyFile private_key_file
     ```

     where *private_key_file* is the path and file name of the server's private key.

     If the private key is encrypted using a pass phrase and you are running under win32, add the additional line

     ```
     SSLPassPhraseDialog exec:exe_name
     ```

     where *exe_name* is the path and file name of an executable that will return the pass phrase on stdout.

     Alternatively, the pass phrase can be removed from the private key using openssl:

     ```
     openssl rsa -in src_file -out dst_file
     ```

where *src_file* is the path and file name of the private key protected by a pass phrase, and *dst_file* is the path and file name of the output file that will contain the unprotected private key. Note that this may reduce server security.

☞ For more information about using HTTPS, see "MobiLink Transport-Layer Security" on page 165.

Example of HTTPS Configuration

Following is an example of how to configure Apache for HTTPS. This example uses Apache's virtual host feature to read HTTPS from port 443 (the default HTTPS port) and HTTP from port 80 at the same time.

```
LoadModule ssl_module modules/mod_ssl.so

        Listen 80
        Listen 443

        NameVirtualHost *:443
        <VirtualHost _default_:443>
            ServerName server_name:443
            ErrorLog logs/https_error
            CustomLog logs/https_access common

            SSLEngine on
            SSLCertificateFile rsaserver.crt
            SSLCertificateKeyFile rsaserver.key
        </VirtualHost>
```

Verifying your setup

❖ To check your configuration

1. Call the Redirector using the following syntax:

   ```
   http://host:port/iaredirect/ml/
   ```

2. Check the log file to see if the Redirector logged a request.

   Note: This test does not make a connection to the MobiLink synchronization server.

Apache Redirector

The Apache Redirector is a native Redirector for Apache web servers version 2.0.x. It has been tested with version 2.0.47. The Apache Redirector is supported for Windows, Solaris and Linux.

If you are using Tomcat, you can also use the servlet Redirector. For more information, see "Servlet Redirector" on page 146.

❖ To configure the Apache Redirector

1. Complete the steps in "Configuring Redirector properties (all versions)" on page 139.

2. Copy the file *mod_iaredirect.dll* or *mod_iaredirect.so* to the appropriate directory in your web server, as follows:

   ♦ For Apache on Windows, the file *mod_iaredirect.dll* is located in the *MobiLink\redirector\apache\v20\* subdirectory of your SQL Anywhere installation. Copy this file to the *%apache-home%\modules* directory on the computer that holds the web server.

   ♦ For Apache for Solaris or Linux, the file *mod_iaredirect.so* is located in the *MobiLink/redirector/apache/v20/OS* subdirectory of your SQL Anywhere installation, where *OS* is **linux** or **solaris**. Copy it to the *%apache-home%/modules* directory on the computer that holds the web server.

3. Update the Apache web server configuration file *httpd.conf* as follows.

   ♦ In the LoadModule section, add the following line:

   ```
   (Windows) LoadModule iaredirect_module pathmodules/mod_
            iaredirect.dll
   (Solaris and Linux) LoadModule iaredirect_module
            pathmodules/mod_iaredirect.so
   ```

   where *path* is the location of the Apache *modules* directory.

   ♦ Add the following section to the file:

   ```
   <Location /iaredirect/ml>
      SetHandler iaredirect-handler
      iaredirectorConfigFile location/redirector.config
   </Location>
   ```

   where */iaredirect/ml* is the path that you will use to invoke the Redirector, and *location* is the directory where *redirector.config* is located.

   ♦ If you are using Apache on Solaris or Linux, you may also want to add the following optional directives to the <Location> section you just created:

- **MaxSyncUsers *number*** The maximum number of MobiLink users synchronizing through the Redirector. This number is used to allocate necessary resources to the Redirector. This number cannot be less than 60. The default is 1000. Only change this setting if the default number of users is less than the actual number.
- **ShmemDiagnosis on|off** If set to on, allows debugging of the memory resource. The default is off.

4. To help with debugging, you may want to increase the amount of logging information that the Redirector outputs. To do this, modify the LogLevel directive in *httpd.conf* and set it to **LogLevel info**. The log level can be (from most to least verbose): debug, info, notice, warn, error, crit, alert, and emerg.

   If you are using any network protocol other than HTTPS, your configuration is complete.

5. If you are using HTTPS synchronization, configure your server as follows. (Note that this HTTPS configuration is identical to the HTTPS instructions for the servlet Redirector on Apache Tomcat and the native Redirector for M-Business Anywhere.)

   ♦ Download and install binaries for mod_ssl and OpenSSL. You can find them using the Apache Module Registry at http://modules.apache.org/. The file *mod_ssl.so* must be copied to *%APACHE_HOME%\modules*. The files *libeay32.dll* and *ssleay32.dll* must be copied to *%APACHE_HOME%\bin*.

   ♦ Generate a server certificate and private key either by generating a request with reqtool.exe and sending it to a third party certificate authority to sign it, or by generating a certificate directly using gencert.exe. The private key can either be in the same file as the server certificate or in its own file.

   ♦ Add the following lines to *httpd.conf*:

   ```
   LoadModule ssl_module modules/mod_ssl.so
      SSLEngine on
      SSLCertificateFile certificate_file
   ```

   where *certificate_file* is the path and file name of the server's certificate file.

   If the server's private key is in a separate file from the server's certificate, add the additional line

   ```
   SSLCertificateKeyFile private_key_file
   ```

   where *private_key_file* is the path and file name of the server's private key.

If the private key is encrypted using a pass phrase and you are running under win32, add the additional line

```
SSLPassPhraseDialog exec:exe_name
```

where *exe_name* is the path and file name of an executable that will return the pass phrase on stdout.

Alternatively, the pass phrase can be removed from the private key using openssl:

```
openssl rsa -in src_file -out dst_file
```

where *src_file* is the path and file name of the private key protected by a pass phrase, and *dst_file* is the path and file name of the output file that will contain the unprotected private key. Note that this may reduce server security.

☞ For more information about using HTTPS, see "MobiLink Transport-Layer Security" on page 165.

Example

Following are examples of the sections of *httpd.conf* that configure the Apache web server to route requests to the MobiLink synchronization server. The line starting with `AddModule` applies only to version 1.3.x. This example works for Windows. For UNIX and Linux, change *mod_iaredirect.dll* to *mod_iaredirect.so.*

```
LoadModule iaredirect_module modules/mod_iaredirect.dll
...
AddModule mod_iaredirect.c
...
<Location /iaredirect/ml>
    SetHandler iaredirect-handler
    iaredirectorConfigFile c:/redirector.config
</Location>
```

❖ To test your configuration

1. Call the Redirector using the following syntax:

   ```
   http://host:port/iaredirect/ml/
   ```

   where *iaredirect/ml* is the path you specified in the <Location> tag of *httpd.conf.*

2. Check the log file to see if the Redirector logged a request.

   Note: This test does not make a connection to the MobiLink synchronization server.

M-Business Anywhere Redirector

The M-Business Anywhere Redirector is supported on Windows, Solaris, and Linux. It has been tested for M-Business Anywhere version 5.5.

❖ To configure the M-Business Anywhere Redirector

1. Complete the steps in "Configuring Redirector properties (all versions)" on page 139.

2. Copy the file *mod_iaredirect.dll* or *mod_iaredirect.so* to the *%avantgo-home%\bin* directory on the computer that holds the web server. This file is located in the *MobiLink\redirector\avantgo\OS* subdirectory of your SQL Anywhere installation, where *OS* is **linux** or **solaris**. (For Windows, there is no *OS* sub-directory.)

3. For Windows, update the M-Business Anywhere web server configuration file *sync.conf* as follows:

   ♦ In the LoadModule section, add the following line:

   ```
   LoadModule iaredirect_module path/bin/mod_iaredirect.dll
   ```

   where *path* is the location of the M-Business Anywhere *bin* directory.

   ♦ Add the following section to the file:

   ```
   <Location /iaredirect/ml>
      SetHandler iaredirect-handler
      iaredirectorConfigFile location/redirector.config
   </Location>
   ```

   where */iaredirect/ml* is the path that you will use to invoke the Redirector, and *location* is the directory where *redirector.config* is located.

   ♦ In the SyncLoadFile section, add the following line:

   ```
   SyncLoadFile path/bin/mod_iaredirect.dll
   ```

   where *path* is the location of the M-Business Anywhere *bin* directory.

4. For Solaris and Linux, update the M-Business Anywhere web server configuration file *sync.conf* as follows:

   ♦ In the LoadModule section, add the following line:

   ```
   LoadModule iaredirect_module pathbin/mod_iaredirect.so
   ```

   where *path* is the location of the M-Business Anywhere *bin* directory.

   ♦ Add the following section to the file:

```
<Location /iaredirect/ml>
   SetHandler iaredirect-handler
   iaredirectorConfigFile location/redirector.config
</Location>
```

where */iaredirect/ml* is the path that you will use to invoke the Redirector, and *location* is the directory where *redirector.config* is located.

- ♦ You may also want to add the following optional directives to the <Location> section you just created:
  - **MaxSyncUsers *number*** The maximum number of MobiLink users synchronizing through the Redirector. This number is used to allocate necessary resources to the Redirector. This number cannot be less than 60. The default is 1000. Only change this setting if the default number of users is less than the actual number.
  - **ShmemDiagnosis on|off** If set to on, allows debugging of the memory resource. The default is off.

5. To help with debugging, you may want to increase the amount of logging information that the Redirector outputs. To do this, modify the LogLevel directive in *sync.conf* and set it to **LogLevel info**. The log level can be (from most to least verbose): debug, info, notice, warn, error, crit, alert, and emerg.

   If you are using any network protocol other than HTTPS, your configuration is complete.

6. If you are using HTTPS synchronization, you must implement both ECC and RSA security.

   ☞ For more information, see the chapters “Security on Windows” or “Security on UNIX” in the *Administrator Guide for M-Business Server.*

Example

Following are examples of the sections of *sync.conf* that configure the M-Business Anywhere web server to route requests to the MobiLink synchronization server.

This example works on Windows:

```
LoadModule iaredirect_module "c:\program files\M-Business
         Anywhere\bin\mod_iaredirect.dll"
...
SyncLoadFile "c:\program files\M-Business Anywhere\bin\mod_
         iaredirect.dll"
...
<Location \iaredirect\ml>
    SetHandler iaredirect-handler
    iaredirectorConfigFile "c:\AvantGoServer\conf\
         redirector.config"
</Location>
```

The following example works on UNIX and Linux:

```
LoadModule iaredirect_module modules/mod_iaredirect.so
...
<Location /iaredirect/ml>
    SetHandler iaredirect-handler
    iaredirectorConfigFile "/redirector.config"
</Location>
```

❖ To test your configuration

1. Call the Redirector using the following syntax:

   ```
   http://host:port/iaredirect/ml/
   ```

   where *iaredirect* is the path you specified in the <Location> tag of *sync.conf*.

2. Check the log file to see if the Redirector logged a request.

   Note: This test does not make a connection to the MobiLink synchronization server.

CHAPTER 9

Running MobiLink Outside the Current Session

About this chapter

This chapter describes how to run the MobiLink synchronization server as a daemon or service.

You can set up MobiLink synchronization server to be available all the time. To make this easier, you can run the MobiLink synchronization server for Windows and for UNIX in such a way that, when you log off the computer it remains running. The way you do this depends on your operating system.

- **UNIX daemon** You can run the MobiLink synchronization server as a daemon using the `-ud` command line option, enabling the MobiLink server to run in the background, and to continue running after you log off.
- **Windows service** You can run the Windows MobiLink server as a service.

Contents

Running the UNIX MobiLink server as a daemon

To run the UNIX MobiLink server in the background, and to enable it to run independently of the current session, you run it as a daemon.

❖ To run the UNIX MobiLink server as a daemon

1. Use the -ud option when starting the MobiLink server. For example:

```
dbmlsrv9 -c "dsn=ASA 9.0 Sample;uid=DBA;pwd=SQL" -ud
```

☞ For more information, see "-ud option" on page 211.

Running the Windows MobiLink server as a service

To run the Windows MobiLink server in the background, and to enable it to run independently of the current session, you run it as a **service**.

You can carry out the following service management tasks from the command line, or on the Services tab in Sybase Central:

- Add, edit, and remove services.
- Start, stop, and pause services.
- Modify the parameters governing a service.
- Add databases to a service, so you can run several databases at one time.

Adding, modifying, and removing services

The service icons in Sybase Central display the current state of each service using a traffic light icon that displays running, paused, or stopped.

❖ To add a new service (Sybase Central)

1. In Sybase Central, click the server in the left pane, and then open the Services tab in the right pane.
2. Double-click Add Service.
3. Follow the instructions in the wizard.

You can also use the dbsvc utility to create the service. For more information, see "Managing services using the dbsvc command-line utility" [*ASA Database Administration Guide,* page 569].

❖ To remove a service (Sybase Central)

1. In Sybase Central, click the server in the left pane, and then open the Services tab in the right pane.
2. In the right pane, right-click the icon of the service you want to remove and choose Delete from the popup menu.

❖ To change the parameters for a service

1. In Sybase Central, click the server in the left pane, and then open the Services tab in the right pane.
2. In the right pane, right-click the service you want to change and choose Properties from the popup menu.
3. Alter the parameters as needed on the tabs of the Service property sheet.
4. Click OK when finished.

Changes to a service configuration take effect the next time the service is started.

Setting the startup option The following options govern startup behavior for MobiLink services. You can set them on the General tab of the service property sheet.

- ♦ **Automatic** If you choose **Automatic**, the service starts whenever the Windows operating system starts. This setting is appropriate for database servers and other applications running all the time.
- ♦ **Manual** If you choose **Manual**, the service starts only when a user with Administrator permissions starts it. For information about Administrator permissions, see your Windows documentation.
- ♦ **Disabled** If you choose **Disabled**, the service will not start.

The startup option is applied the next time Windows is started.

Specifying command line options The Configuration tab of the service property sheet provides a text box for typing command line options for a service. Do not type the name of the program executable in this box.

For example, to start a MobiLink synchronization service with verbose logging and three worker threads, type the following in the Parameters box:

```
-c "dsn=ASA 9.0 Sample;uid=DBA;pwd=SQL"
-vc
-w 3
```

☞ The command line options for a service are the same as those for the executable. For a full description of the command line options for MobiLink, see "MobiLink Synchronization Server Options" on page 189.

Setting account options You can choose which account the service runs under. Most services run under the special LocalSystem account, which is the default option for services. You can set the service to log on under another account by opening the Account tab on the Service property sheet, and typing the account information.

If you choose to run the service under an account other than LocalSystem, that account must have the “log on as a service” privilege. This can be granted from the Windows User Manager application, under Advanced Privileges.

Whether or not an icon for the service appears on the taskbar or desktop depends on the account you select, and whether Allow Service to Interact with Desktop is checked, as follows:

♦ If a service runs under LocalSystem, and Allow Service to Interact with Desktop is checked in the service property sheet, an icon appears on the desktop of every user logged in to Windows NT/2000/XP on the computer running the service. Consequently, any user can open the application window and stop the program running as a service.

♦ If a service runs under LocalSystem, and Allow Service to Interact with Desktop is unchecked in the service property sheet, no icon appears on the desktop for any user. Only users with permissions to change the state of services can stop the service.

♦ If a service runs under another account, no icon appears on the desktop. Only users with permissions to change the state of services can stop the service.

Changing the executable file

To change the program executable file associated with a service in Sybase Central, click the Configuration tab on the Service property sheet and type the new path and file name in the File Name box.

If you move an executable file to a new directory, you must modify this entry.

Starting, stopping, and pausing services

❖ To start, stop, or pause a service

1. In Sybase Central, click the server in the left pane, and then open the Services tab in the right pane.

2. Right-click the service and choose Start, Stop, or Pause from the popup menu.

   To resume a paused service, right-click the service and select Continue from the popup menu.

If you start a service, it keeps running until you stop it. Closing Sybase Central or logging off does not stop the service.

Stopping a service closes all connections to the database and stops the database server. For other applications, the program closes down.

Pausing a service prevents any further action being taken by the application. It does not shut the application down or (in the case of server services) close any client connections to the database. Most users do not need to pause their services.

Running more than one service at a time

Although you can use the Windows Service Manager in the Control Panel for some tasks, you cannot install or configure a MobiLink service from the Windows Service Manager. You can use Sybase Central to carry out all the service management for MobiLink.

When you open the Windows Service Manager from the Windows Control Panel, a list of services appears. The names of the Adaptive Server Anywhere services are formed from the Service Name you provided when installing the service, prefixed by Adaptive Server Anywhere. All the installed services appear together in the list.

This section describes topics specific to running more than one service at a time.

Service dependencies

In some circumstances you may wish to run more than one executable as a service, and these executables may depend on each other. For example, you must run the MobiLink synchronization server and the database server in order to synchronize.

In cases such as these, the services must start in the proper order. If a MobiLink synchronization service starts up before the consolidated database server has started, it fails because it cannot find the consolidated database server. The sequence must be such that the database server is running when you start the MobiLink server. (This does not apply if the consolidated database server is on another computer.)

You can prevent these problems using service groups, which you manage from Sybase Central.

Service groups

You can assign each service on your system to be a member of a service group. By default, each service belongs to a group. The default group for the MobiLink synchronization server is ASANYMobiLink.

Before you can configure your services to ensure they start in the correct order, you must check that your service is a member of an appropriate group. You can check which group a service belongs to, and change this group, from Sybase Central.

❖ To check and change which group a service belongs to

1. In Sybase Central, click the server in the left pane, and then open the Services tab in the right pane.
2. Right-click the service and choose Properties from the popup menu.
3. Click the Dependencies tab. The top text box displays the name of the group the service belongs to.
4. Click Change to display a list of available groups on your system.
5. Select one of the groups, or type a name for a new group.
6. Click OK to assign the service to that group.

Managing service dependencies

With Sybase Central, you can specify dependencies for a service. For example:

- ♦ You can ensure that at least one group has started before the current service.
- ♦ You can ensure that any service starts before the current service.

❖ To add a service or group to a list of dependencies

1. In Sybase Central, click the server in the left pane, and then open the Services tab in the right pane
2. Right-click the service and choose Properties from the popup menu.
3. Click the Dependencies tab.
4. Click Add Services or Add Service Groups to add a service or group to the list of dependencies.
5. Select one of the services or groups from the list.
6. Click OK to add the service or group to the list of dependencies.

Troubleshooting MobiLink server startup

This section describes some common problems when starting the MobiLink server.

Ensure that network communication software is running

Appropriate network communication software must be installed and running before you run the MobiLink server. If you are running reliable network software with just one network installed, this should be straightforward. You should confirm that other software requiring network communications is working properly before running the MobiLink server.

If you are running under the TCP/IP protocol, you may want to confirm that ping and telnet are working properly. The ping and telnet applications are provided with many TCP/IP protocol stacks.

Debugging network communications startup problems

If you are having problems establishing a connection across a network, you can use debugging options at both client and server to diagnose problems. The startup information appears on the server window: you can use the `-o` option to log the results to an output file.

CHAPTER 10

MobiLink Transport-Layer Security

About this chapter

This chapter shows you how to secure communications between the MobiLink synchronization server and MobiLink clients using MobiLink transport-layer security.

Separately licensable option required

Transport-layer security requires that you obtain the separately-licensable SQL Anywhere Studio security option and is subject to export regulations.

☞ To order this component, see "Separately-licensable components" [*Introducing SQL Anywhere Studio*, page 5].

Contents

Introduction

Separately licensable option required
Transport-layer security requires that you obtain the separately-licensable SQL Anywhere Studio security option and is subject to export regulations.

☞ To order this component, see "Separately-licensable components" [*Introducing SQL Anywhere Studio*, page 5].

Transport-layer security, an IETF standard protocol, secures client/server applications using digital certificates and public-key cryptography.

Clients use trusted public certificates to encrypt data and authenticate servers in the initial client/server handshake. Data transmitted by the client can only be decrypted by the matching private key, which is stored in your MobiLink synchronization server certificate.

For server authentication, the MobiLink synchronization server sends its public certificate to the client. The client verifies the identity of the server using certificate fields and the digital signature embedded in the certificate.

Efficiency

The transport-layer security standard overcomes the inefficiencies associated with public-key cryptography. Once a secure connection is established, the client and server exchange a common key. They use a highly efficient symmetric cipher for the rest of their communication.

Supported platforms

FIPS-certified security options are available on Windows only, and do not work on UltraLite.

FIPS 140-2 certification

Federal Information Processing Standard (FIPS) 140-2 specifies requirements for security algorithms. It does not, however, specify requirements for security protocols such as SSL or transport-layer security. FIPS 140-2 is granted by the American and Canadian governments through the National Institute of Standards and Testing (NIST) and the Canadian Communication Security Establishment (CSE). Certicom has earned FIPS certification for security algorithms implemented on Windows.

SQL Anywhere Studio offers transport-layer security with the option of using the underlying FIPS-certified algorithms in the Certicom software.

To use transport-layer security, you must purchase a separate security option.

☞ For information about how to order transport-layer security, see "Separately-licensable components" [*Introducing SQL Anywhere Studio*, page 5].

You can use FIPS-certified security algorithms to encrypt your database files, or to encrypt communications for database client/server communication, web services, and MobiLink client/server communication.

☞ For more information, see:

- ♦ "Encrypting a database" [*SQL Anywhere Studio Security Guide,* page 15]
- ♦ "Using transport-layer security for web services" [*SQL Anywhere Studio Security Guide,* page 44]
- ♦ "Starting the MobiLink synchronization server with transport-layer security" on page 177
- ♦ "Configuring MobiLink clients to use transport-layer security" on page 179

Setting up transport-layer security

To set up MobiLink transport-layer security, perform the following steps:

♦ **Create digital certificates** Create public certificates and server certificates. Public certificates are distributed to MobiLink clients, while server certificates are stored securely with MobiLink synchronization servers.

  ☞ See “Creating digital certificates” on page 169.

♦ **Start the MobiLink synchronization server with transport-layer security** Use dbmlsrv9 command-line options to specify the type of security, the server certificate, and the password to protect the private key.

  ☞ See “Starting the MobiLink synchronization server with transport-layer security” on page 177.

♦ **Configure MobiLink clients to use transport-layer security** Specify the path and file name of trusted public certificates.

  Supply the appropriate security or network protocol options with the MobiLink synchronization client utility (dbmlsync) or UltraLite application.

  ☞ See “Configuring MobiLink clients to use transport-layer security” on page 179.

Creating digital certificates

To set up transport-layer security you must generate digital certificates.

You can create self-signed certificates, use enterprise root certificates and certificate chains, or have your certificates signed by a Certificate Authority (CA).

- **Self-signed certificates** Self-signed certificates can be used for simple setups involving a single MobiLink synchronization server. In this case, the private key used to create trusted public certificates is stored with your MobiLink synchronization server instead of a commercial Certificate Authority or dedicated facility.

  ☞ See "Self-signed root certificates" on page 169.

- **Enterprise root certificates** Enterprise root certificates increase data integrity and extensibility for multi-server deployments.

  - You can store the private key used to create trusted public certificates in a secure central location.
  - You can add MobiLink synchronization servers without reconfiguring clients.

  See "Certificate chains" on page 170.

- **Commercial Certificate Authorities** You can use a third-party Certificate Authority instead of an enterprise root certificate. Commercial Certificate Authorities have dedicated facilities to store private keys and create high-quality server certificates.

  ☞ See "Certificate chains" on page 170 and "Globally-signed certificates" on page 173.

Certificate utilities

The SQL Anywhere Studio certificate generation utility, gencert, creates certificates. It prompts you for certificate identification and file information and uses RSA or elliptic-curve encryption technology. You can use the certificate reader utility, readcert, to display certificate values and validate a chain of certificates.

Self-signed root certificates

Self-signed root certificates can be used for simple setups involving a single MobiLink synchronization server. In this case, the private key used to create trusted public certificates is stored with your MobiLink synchronization server instead of a commercial Certificate Authority or dedicated facility.

Tip
Use enterprise level certificate chains if you operate multiple MobiLink synchronization servers or are looking for a higher level of certificate integrity.

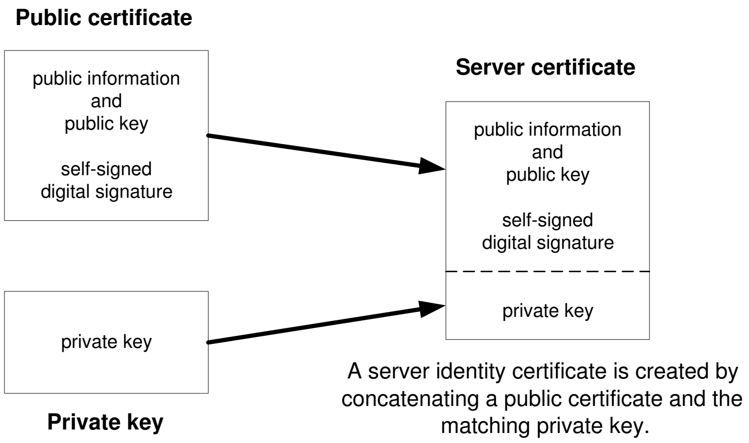
☞ For more information about setting up certificate chains, see "Certificate chains" on page 170.

To set up self-signed certificates, you generate the following certificates using the gencert utility:

- ♦ **Public certificate** The self-signed public certificate is distributed to MobiLink clients. It is an electronic document including identity information, the public key of the MobiLink synchronization server, and a self-signed digital signature used for server authentication.

  ☞ For more information about digital signatures and server authentication, see "Server authentication" on page 179.

- ♦ **Server certificate** The server certificate is stored securely with a MobiLink synchronization server. It is a combination of the self-signed public certificate (that is distributed to MobiLink clients) and the corresponding private key. The private key gives the MobiLink synchronization server the ability to decrypt messages sent by Adaptive Server Anywhere or UltraLite clients.



☞ For information about how to generate self-signed root certificates, see "Certificate generation utility" on page 496.

Certificate chains

You can improve the security and extensibility of a multi-server environment using certificate chains instead of self-signed certificates. Certificate chains require a Certificate Authority or an enterprise root certificate to sign MobiLink synchronization server certificates.

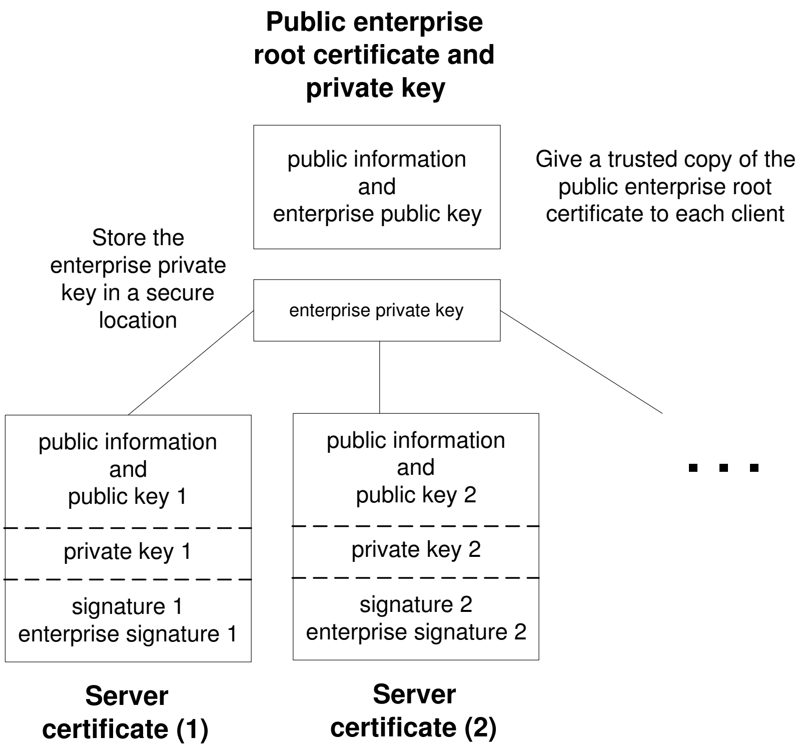
☞ For more information about self-signed certificates, see "Self-signed root certificates" on page 169.

Benefits of using certificate chains

Certificate chains provide the following advantages:

- **Extensibility** You can configure MobiLink synchronization clients to trust any certificate signed by an enterprise root certificate or Certificate Authority. If you add a new MobiLink synchronization server, clients do not require a copy of the new public certificate.
- **Security** The enterprise root certificate's private key does not reside with MobiLink synchronization servers. Storing the root certificate's private key in a high-security location, or using a Certificate Authority with dedicated facilities, protects the integrity of server authentication.

The following diagram provides the basic enterprise root certificate architecture.


Public enterprise root certificate and private key
public information and enterprise public key
Give a trusted copy of the public enterprise root certificate to each client
Store the enterprise private key in a secure location
enterprise private key
public information and public key 1
private key 1
signature 1 enterprise signature 1
public information and public key 2
private key 2
signature 2 enterprise signature 2
. . .
Server certificate (1)
Server certificate (2)

To create certificates used in a multi-server environment:

- ♦ Generate a public enterprise root certificate and enterprise private key.

  You distribute the public enterprise root certificate to MobiLink clients. You store the enterprise private key in a secure location, preferably a dedicated facility.

- ♦ Generate server certificates for each MobiLink synchronization server.

  Use the public enterprise root certificate and enterprise private key to sign each server certificate.

You can also use a third-party Certificate Authority to sign your server certificates. Commercial Certificate Authorities have dedicated facilities to store private keys and create high-quality server certificates.

☞ For more information, see “Globally-signed certificates” on page 173.

Enterprise root certificates

Enterprise root certificates increase data integrity and extensibility for multi-server deployments.

- ♦ You can store the private key used to create trusted public certificates in a secure central location.
- ♦ You can add MobiLink synchronization servers without reconfiguring clients.

To set up enterprise root certificates, you create the enterprise root certificate and the enterprise private key that you use to sign server certificates.

☞ For information about creating server certificates, see “Signed server certificates” on page 172.

☞ For information about how to generate enterprise root certificates, see “Certificate generation utility” on page 496.

Signed server certificates

You generate server certificates for each MobiLink synchronization server. Since these certificates are signed by an enterprise root certificate, you use the gencert -s option.

☞ For information about generating signed server certificates, see “Certificate generation utility” on page 496.

☞ For information about how to generate signed server certificates fo each MobiLink synchronization server, see “Certificate generation utility” on page 496.

Globally-signed certificates

A commercial Certificate Authority is an organization that is in the business of creating high-quality certificates and using these certificates to sign your certificate requests.

Globally-signed certificates have the following advantages:

- ♦ In the case of inter-company communication, common trust in an outside, recognized authority may increase confidence in the security of the system. A Certificate Authority must guarantee the accuracy of the identification information in any certificate that it signs.
- ♦ Certificate Authorities provide controlled environments and advanced methods to generate certificates.
- ♦ The private key for the root certificate must remain private. Your organization may not have a suitable place to store this crucial information, whereas a Certificate Authority can afford to design and maintain dedicated facilities.

Setting up globally-signed certificates

To set up globally signed certificates, you:

- ♦ Create a certificate request using Certicom's reqtool utility.

  ☞ See "Using reqtool to obtain global certificates" on page 173.
- ♦ Use a Certificate Authority to sign each MobiLink synchronization server certificate request.

  ☞ See "Using a global certificate as a server certificate" on page 174.

Globally-signing enterprise root certificates
You might be able to globally-sign an enterprise root certificate. This is only applicable if your Certificate Authority generates certificates that can be used to sign other certificates.

Using reqtool to obtain global certificates

MobiLink transport-layer security is based on Certicom SSL/TLS Plus libraries, which require elliptic-curve or RSA certificates. You can obtain a global certificate from any Certificate Authority that can supply certificates in the correct format.

There are several ways to obtain certificates. One way is to use the reqtool utility, which is installed when you install the security component. This tool creates a server's private key and a global certificate request.

Example

The following example creates an elliptic-curve certificate request:

```
> reqtool
-- Certicom Corp. Certificate Request Tool 3.0d1 --
Choose certificate request type:
  E - Personal email certificate request.
  S - Server certificate request.
  Q - Quit.
Please enter your request [Q] : S
Choose key type:
  R - RSA key pair.
  D - DSA key pair.
  E - ECC key pair.
  Q - Quit.
Please enter your request [Q] : E
Using curve ec163a02. Generating key pair (please wait)...
Country: CA
State: Ontario
Locality: Waterloo
Organization: Sybase, Inc.
Organizational Unit: IAS
Common Name: IAS_Waterloo
Enter password to protect private key : mypwd123
Enter file path to save request : global.req
Enter file path to save private key : serv1_private_key.pri
```

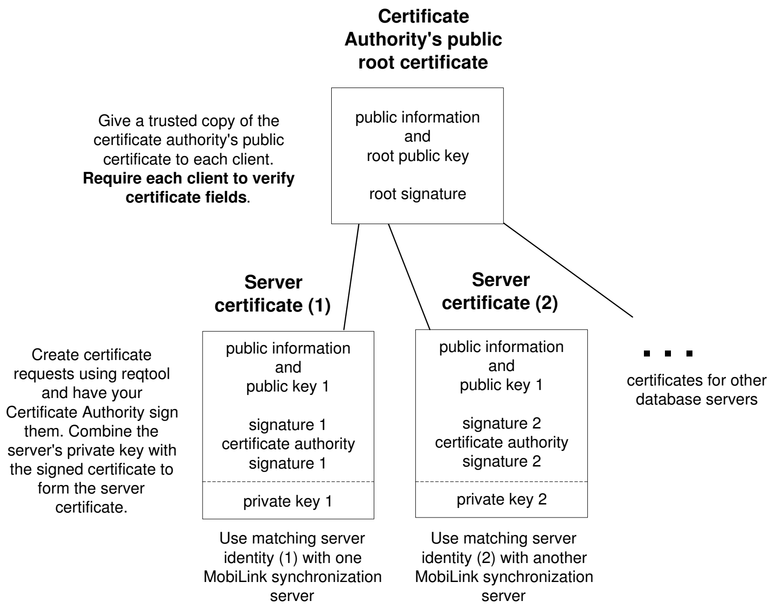
The file *global.req* contains the public certificate and request information. Paste the contents of this file into a form on the certificate-issuing web site. The Certificate Authority will sign the request and create the public certificate *global.crt*.

The file *serv1_private_key.pri* contains the corresponding private key. This file is protected by the password you entered, but since the protection provided by the password is weak, you must store this file in a secure location.

☞ For more information about using reqtool, see the document *reqtool.pdf*, located in the *win32* subdirectory of your SQL Anywhere 9 installation.

Using a global certificate as a server certificate

You can use globally-signed certificates directly as MobiLink synchronization server certificates. The following diagram shows the configuration for a multi-server deployment:


Certificate Authority's public root certificate
public information and root public key
root signature
Give a trusted copy of the certificate authority's public certificate to each client. **Require each client to verify certificate fields**.
Server certificate (1)
Server certificate (2)
public information and public key 1
signature 1 certificate authority signature 1
private key 1
public information and public key 1
signature 2 certificate authority signature 2
private key 2
. . .
certificates for other database servers
Create certificate requests using reqtool and have your Certificate Authority sign them. Combine the server's private key with the signed certificate to form the server certificate.
Use matching server identity (1) with one MobiLink synchronization server
Use matching server identity (2) with another MobiLink synchronization server

You can use globally-signed certificates directly as database server certificates. The following diagram shows the configuration for a multi-server deployment:

To create the server identity, you must concatenate the public certificate signed by the Certificate Authority and private key created using the reqtool utility.

☞ For more information about the reqtool utility, see "Using reqtool to obtain global certificates" on page 173.

The following example concatenates the globally-signed public certificate *global.crt* and the private key *serv1_private_key.pri* to create the server certificate *server1_certificate.crt*.

```
copy global.crt+serv1_private_key.pri server1_certificate.crt
```

You reference the server certificate *server1_certificate.crt* and the password for the private key *serv1_private_key.pri* at the dbmlsrv9 command line.

☞ For more information, see "Starting the MobiLink synchronization server with transport-layer security" on page 177.

Setting up clients to trust the certificate authority's public certificate

You must ensure that clients contacting your MobiLink synchronization server trust the root certificate in the chain. In the case of globally-signed certificates, the root certificate is the Certificate Authority's public certificate.

Certificate field verification

When using a globally-signed certificate, each MobiLink client must verify field values to avoid trusting certificates that the same Certificate Authority has signed for other clients.

☞ For more information about verifying certificate fields for globally-signed certificates, see "Verifying certificate fields" on page 180.

☞ For more information about configuring MobiLink clients to trust server certificates, see "Configuring MobiLink clients to use transport-layer security" on page 179.

☞ For more information about using globally-signed certificates to establish trust, see "Globally-signed certificates" on page 173.

Starting the MobiLink synchronization server with transport-layer security

To start the MobiLink synchronization server with transport-layer security, supply the server certificate and the password protecting the server's private key.

☞ For an overview of the steps required to set up transport-layer security, see "Setting up transport-layer security" on page 168.

Securing the server over TCP/IP

If you are using TCP/IP, use the dbmlsrv9 -x server option to specify certificate and certificate_password parameters. Following is a partial dbmlsrv9 command line:

-x tcpip(security=*cipher***(certificate=***server-certificate***;certificate_password=***password***;**...**))**

- ***cipher*** can be **rsa_tls** or **ecc_tls** for RSA and elliptic-curve encryption, respectively. For FIPS-approved RSA encryption, specify **rsa_tls_fips**. rsa_tls_fips uses separate FIPS 140-2 certified software from Certicom. It is compatible with clients using rsa_tls (with version 9.0.2 or later), and clients using rsa_tls_fips are also compatible with servers using rsa_tls. rsa_tls_fips can only be used with Adaptive Server Anywhere databases on Windows.

  The cipher must match the encryption used to create your certificates.

- ***server-certificate*** is the path and file name of the server certificate.

  ☞ For more information about creating the server certificate, which can be self-signed, or signed by a Certificate Authority or enterprise root certificate, see "Creating digital certificates" on page 169.

- ***password*** is the password for the server certificate's private key. You specify this password when you create the server certificate.

☞ For more information about the dbmlsrv9 -x option, see "-x option" on page 214.

Securing the server over HTTPS

If you are using HTTPS, use the dbmlsrv9 -x option to specify certificate and certificate_password parameters directly. Following is a partial dbmlsrv9 command line:

-x *protocol***(certificate=***server-certificate***;certificate_password=***password***;**...**))**

- ***protocol*** can be **https**; or **https_fips** for FIPS-approved RSA encryption. https_fips uses separate uses separate FIPS 140-2 certified software from Certicom but is compatible with clients using https (and version 9.0.2 or later). https_fips can only be used with Adaptive Server Anywhere databases on Windows.

- ***server-certificate*** is the path and file name of the server certificate.

  For HTTPS, you must use an RSA certificate.

  ☞ For more information about creating the server certificate, see "Creating digital certificates" on page 169.

- ***password*** is the password for the server certificate's private key. You specify this password when you create the server certificate.

☞ For more information about the dbmlsrv9 -x option, see "-x option" on page 214.

Static Java UltraLite applications

Static Java UltraLite uses a separate protocol for transport-layer security.

☞ For more information about using transport-layer security for static Java UltraLite, see "Using transport-layer security" [*UltraLite Static Java User's Guide*, page 42].

Examples

The following example specifies the type of security (RSA), the server certificate, and the password protecting the server's private key:

```
dbmlsrv9 -c "dsn=my_cons"
 -x tcpip(security=rsa_tls(certificate=c:\test\serv_
         rsa1.crt;certificate_password=pwd))
```

For elliptic-curve certificates, enter:

```
dbmlsrv9 -c "dsn=my_cons"
 -x tcpip(security=ecc_tls(certificate=c:\test\serv_
         ecc1.crt;certificate_password=pwd))
```

☞ For more information about the dbmlsrv9 -x option, see "-x option" on page 214.

☞ For more information about creating the server certificate, in this case *serv1.crt*, see "Creating digital certificates" on page 169.

☞ You can hide the command-line options using a configuration file and the File Hiding utility, dbfhide. For more information, see "@data option" on page 194.

Configuring MobiLink clients to use transport-layer security

You can configure Adaptive Server Anywhere or UltraLite clients for MobiLink transport-layer security. For each client, you specify trusted public certificates, the type of encryption, and the network protocol.

☞ For an overview of the steps required to set up transport-layer security, see "Setting up transport-layer security" on page 168.

Server authentication

Server authentication allows a remote client to verify the identity of a MobiLink synchronization server. Digital signatures and certificate field verification work together to achieve server authentication.

Digital signatures

A MobiLink synchronization server certificate contains one or more digital signatures used to maintain data integrity and protect against tampering. Following are the steps used to create a digital signature:

- An algorithm performed on a certificate generates a unique value or hash.
- The hash is encrypted using a signing certificate's or Certificate Authority's private key.
- The encrypted hash, called a digital signature, is embedded in the certificate.

A digital signature can be self-signed or signed by an enterprise root certificate or Certificate Authority.

When a MobiLink client contacts a MobiLink synchronization server, and each is configured to use transport-layer security, the server sends the client a copy of its public certificate. The client decrypts the certificate's digital signature using the server's public key included in the certificate, calculates a new hash of the certificate, and compares the two values. If the values match, this confirms the integrity of the server's certificate.

☞ For more information about self-signed certificates, see "Self-signed root certificates" on page 169.

☞ For more information about enterprise root certificates and Certificate Authorities, see "Certificate chains" on page 170.

Verifying certificate fields

When using a globally signed certificate, each client must verify certificate field values to avoid trusting certificates that the same Certificate Authority has signed for other clients. This is resolved by requiring your clients to test the value of fields in the identity portion of the certificate. A Certificate Authority must guarantee the accuracy of the identification information in any certificate that it signs.

☞ For more information about globally signed certificates, see "Globally-signed certificates" on page 173.

When creating a certificate using the gencert utility, you enter values for the organization, organizational unit, and common name fields. You verify these fields using corresponding MobiLink client connection parameters.

- **Organization** The organization field corresponds to the certificate_company MobiLink client connection parameter.

  ☞ See "certificate_company" [*MobiLink Clients,* page 37].

- **Organizational unit** The organizational unit field corresponds to the certificate_unit MobiLink client connection parameter.

  ☞ See "certificate_unit" [*MobiLink Clients,* page 40].

- **Common name** The common name field corresponds to the certificate_name MobiLink client connection parameter.

  ☞ See "certificate_name" [*MobiLink Clients,* page 38].

  The common name field corresponds to the certificate_name encryption connection parameter.

☞ For more information about setting up MobiLink synchronization clients, see:

- "Configuring UltraLite clients to use transport-layer security" on page 183.
- "Client security options" on page 180.

☞ For more information about creating digital certificates, see "Creating digital certificates" on page 169.

Client security options

Adaptive Server Anywhere and UltraLite use a common set of connection parameters to configure transport-layer security.

trusted_certificates parameter

MobiLink clients use the trusted_certificates connection parameter to specify trusted MobiLink synchronization server certificates. The trusted certificate can be a server's self-signed public certificate, a public enterprise root certificate, or the public certificate belonging to a commercial Certificate Authority.

☞ For more information, see:

- ♦ "trusted_certificates" [*MobiLink Clients*, page 53]
- ♦ "Creating digital certificates" on page 169

Verifying certificate fields

The certificate_company, certificate_unit, and certificate_name connection parameters are used to verify certificate fields, an important step for server authentication. It is strongly recommended that you verify certificate fields if you are using a third-party Certificate Authority to globally-sign certificates.

☞ For more information about verifying certificate fields, see:

- ♦ "Verifying certificate fields" on page 180
- ♦ "Globally-signed certificates" on page 173
- ♦ "Server authentication" on page 179

Configuring Adaptive Server Anywhere clients to use transport-layer security

This section shows you how to configure Adaptive Server Anywhere clients to use transport-layer security over HTTPS or TCP/IP.

Transport-layer security over HTTPS

MobiLink transport-layer security is an inherent feature of the MobiLink HTTPS protocol. To use transport-layer security over HTTPS, specify the trusted_certificates connection parameter using the ADR extended option. Following is a partial dbmlsync command line.

-e "ctp=*protocol***;adr=trusted_certificates=***public-certificate*...**"**

- ♦ ***protocol*** can be **https**, or **https_fips** for FIPS-approved RSA encryption. https_fips uses separate FIPS 140-2 certified software from Certicom but is compatible with MobiLink synchronization servers using https and version 9.0.2 or later.
- ♦ ***public-certificate*** is the path and file name of a trusted public certificate.

  For HTTPS or HTTPS_FIPS, you must use certificates created using RSA encryption.

☞ For more information about trusted_certificates and other client security parameters, see “Client security options” on page 180.

☞ For more information about creating or obtaining the public certificate, see “Creating digital certificates” on page 169.

☞ For more information about HTTPS parameters, see the HTTPS section in “CommunicationAddress (adr) extended option” [*MobiLink Clients*, page 106].

Examples

The following example specifies RSA security over HTTPS at the dbmlsync command line.

```
dbmlsync -c "eng=rem1;uid=dba;pwd=mypwd"
  -e "ctp=https;adr=trusted_certificates=c:\temp\public_
         cert.crt;certificate_company=Sybase, Inc.;certificate_
         unit=IAS;certificate_name=MobiLink)"
```

Alternatively, you can specify the CommunicationAddress extended option using the CREATE SYNCHRONIZATION SUBSCRIPTION or ALTER SYNCHRONIZATION SUBSCRIPTION statement. This method provides the same information but stores it in the database.

```
CREATE SYNCHRONIZATION SUBSCRIPTION
 TO pub1
 FOR user1
 ADDRESS 'trusted_certificates=c:\temp\public_cert.crt;
    certificate_company=Sybase, Inc.;
    certificate_unit=IAS;
  certificate_name=MobiLink'
 OPTION scriptversion='ver1';
```

Transport-layer security over TCP/IP

To configure Adaptive Server Anywhere clients to use transport-layer security over TCP/IP, use the CommunicationAddress (adr) extended option to specify trusted certificates.

Unlike HTTPS, where you can specify trusted_certificates and certificate fields directly, you must use the security option and the following syntax for TCP/IP:

adr=security=*cipher***(trusted_certificates=***public-certificate***;** ...**)**

♦ ***cipher*** can be **rsa_tls** or **ecc_tls** for RSA and elliptic-curve encryption, respectively.

  For FIPS-approved RSA encryption, specify **rsa_tls_fips**. rsa_tls_fips uses a separate approved library but is compatible with servers using rsa_tls. Clients using rsa_tls are also compatible with servers using rsa_tls_fips. rsa_tls_fips can be used on SQL Anywhere Studio databases on Windows.

The cipher must match the encryption used to create your certificates.

♦ ***public-certificate*** is the path and file name of a trusted public certificate.

☞ For more information about trusted_certificates and other client security parameters, see "Client security options" on page 180.

☞ For more information about creating or obtaining the public certificate, see "Creating digital certificates" on page 169.

☞ For more information about the CommunicationAddress (adr) extended option, including a list of security parameters, see "CommunicationAddress (adr) extended option" [*MobiLink Clients*, page 106].

Examples

The following example specifies RSA security and TCP/IP at the dbmlsync command line:

```
dbmlsync -c "eng=rem1;uid=myuid;pwd=mypwd"
  -e "ctp=tcpip;adr=port=3333;security=rsa_tls(trusted_
         certificates=c:\test\public_cert.crt;certificate_
         company=Sybase, Inc.;certificate_unit=IAS;certificate_
         name=MobiLink)"
```

Another option is to specify the CommunicationAddress extended option using the CREATE SYNCHRONIZATION SUBSCRIPTION or ALTER SYNCHRONIZATION SUBSCRIPTION statement:

```
CREATE SYNCHRONIZATION SUBSCRIPTION
  TO pub1
  FOR user1
  ADDRESS 'port=3333;security=rsa_tls(trusted_
         certificates=public_cert.crt;
    certificate_company=Sybase, Inc.;
    certificate_unit=IAS;
  certificate_name=MobiLink )'
     OPTION scriptversion='ver1';
```

☞ For more information see "CREATE SYNCHRONIZATION SUBSCRIPTION statement [MobiLink]" [*MobiLink Clients*, page 162] and "ALTER SYNCHRONIZATION SUBSCRIPTION statement [MobiLink]" [*MobiLink Clients*, page 156].

Configuring UltraLite clients to use transport-layer security

This section shows you how to configure UltraLite clients to use transport-layer security over HTTPS or TCP/IP.

Transport-layer security over HTTPS

MobiLink transport-layer security is an inherent feature of the MobiLink HTTPS protocol. If you use HTTPS and UltraLite clients, you can specify trusted certificates and certificate fields directly as network protocol options.

For HTTPS, you must use RSA certificates.

☞ For more information about specifying the HTTPS protocol for your UltraLite interface, see "HTTPS protocol options" [*MobiLink Clients*, page 347].

The ul_synch_info security_stream and security_parms security synchronization parameters are only used for TCP/IP and have no effect when you use HTTPS.

☞ For more information about the security synchronization parameter, see "Security synchronization parameter" [*MobiLink Clients*, page 328] and "Transport-layer security over TCP/IP" on page 186.

❖ To configure your UltraLite client to use transport-layer security over HTTPS

1. Specify trusted root certificates in the UltraLite schema.

   ♦ **Static UltraLite interfaces** For static UltraLite interfaces, including UltraLite for static C/C++, Embedded SQL, and static Java, use the ulgen -r option. The following example specifies the trusted root certificate *ent_root.crt* for static C/C++ UltraLite:

   ```
   ulgen -a -c "dsn=myCons" -j Product -s ISampleSQL
     -f SampleDB -r ent_root.crt
   ```

   For more information about the ulgen -r option, see "The UltraLite Generator" [*UltraLite Database User's Guide*, page 89].

   ♦ **UltraLite component interfaces** For UltraLite component interfaces, specify trusted certificates using the Schema Painter. The Schema Painter's Database Schema Properties dialog includes a Certification tab.

   ☞ For more information, see "Database Schema property sheet: Certification tab" [*SQL Anywhere Studio Help*, page 267].

2. Specify the HTTPS protocol for synchronization.

   The following example uses the ULHTTPSStream() function for C/C++ UltraLite:

```
auto ul_synch_info synch_info;
conn.InitSynchInfo( &synch_info );
synch_info.user_name = UL_TEXT( "50" );
synch_info.version = UL_TEXT( "ul_default" );
...
synch_info.stream = ULHTTPSStream();
...
```

☞ For more information about ULHTTPSStream(), see "ULHTTPSStream function" [*UltraLite C/C++ User's Guide,* page 374].

3. Specify HTTPS protocol options.

   The following example uses the stream_parms field of ul_synch_info to specify standard HTTPS parameters and verify certificate fields for C/C++ UltraLite.

```
auto ul_synch_info synch_info;
...
synch_info.stream = ULHTTPSStream();
synch_info.stream_parms = TEXT(
      "port=9999;
       certificate_company=Sybase, Inc.;
       certificate_unit=IAS;
       certificate_name=MobiLink");
    synch_info.security_stream = NULL;
    synch_info.security_parms = NULL;
```

   The certificate_company, certificate_unit, and certificate_name parameters are used to verify certificate fields.

   ☞ For more information about verifying certificate fields, see "Verifying certificate fields" on page 180.

   You can also specify the trusted_certificates HTTPS protocol option, which overrides any trusted certificate information embedded in the UltraLite schema (step 1 of this procedure). You can only use this parameter for Windows NT/2000/XP or Windows CE.

```
auto ul_synch_info synch_info;
...
synch_info.stream = ULHTTPSStream();
synch_info.stream_parms = TEXT(
      "port=9999;
       trusted_certificates=\rsaroot.crt;
       certificate_company=Sybase, Inc.;
       certificate_unit=IAS;
       certificate_name=MobiLink");
    synch_info.security_stream = NULL;
    synch_info.security_parms = NULL;
```

   For more information about HTTPS options, see "HTTPS protocol options" [*MobiLink Clients,* page 347]

Transport-layer security over TCP/IP

To configure UltraLite clients to use transport-layer security over TCP/IP, specify trusted root certificates and the appropriate security stream.

❖ To configure your UltraLite client to use transport-layer security over TCP/IP

1. Specify trusted root certificates in the UltraLite schema.
   - **Static UltraLite interfaces** For static UltraLite interfaces, including UltraLite for static C/C++, Embedded SQL, and static Java, use the ulgen -r option. The following example specifies trusted root certificates for static C/C++ UltraLite:

     ```
     ulgen -a -c "dsn=myCons" -j Product -s ISampleSQL
       -f SampleDB -r ent_root.crt
     ```

     For more information about the ulgen -r option, see "The UltraLite Generator" [*UltraLite Database User's Guide,* page 89].
   - **UltraLite component interfaces** For UltraLite component interfaces, specify trusted certificates using the Schema Painter. The Schema Painter's Database Schema Properties dialog includes a Certification tab.

     ☞ For more information, see "Database Schema property sheet: Certification tab" [*SQL Anywhere Studio Help,* page 267].
2. Supply the appropriate security or protocol options in the programming interface.
   - **Static Java UltraLite** Static Java UltraLite requires a separate stream for transport-layer security and does not provide a separate security parameter. The following example use the UlSecureSocketStream as the protocol for elliptic-curve security:

     ```
     UlSynchOptions opts = new UlSynchOptions();
     opts.setStream(new UlSecureSocketStream() );
     opts.setStreamParms( "host=myserver;"
        + "port=2439;"
        + "certificate_company=Sybase Inc.;"
        + "certificate_unit="IAS;"
        + "certificate_name=Mobilink");
     // set other options here
     conn.synchronize( opts );
     ```

     For more information about using Transport-Layer Security for static Java UltraLite, see "Using transport-layer security" [*UltraLite Static Java User's Guide,* page 42].
   - **UltraLite Interfaces using the security synchronization parameter** Set the security synchronization parameter for applicable UltraLite

interfaces including UltraLite for static C++, Embedded SQL, and the C++ UltraLite component. For other component interfaces, you must use HTTPS for transport-layer security.

You can use ULSecureRSATLSStream() or ULSecureCerticomTLSStream() to specify RSA and elliptic-curve security, respectively.

☞ For more information, see "Security synchronization parameter" [*MobiLink Clients,* page 328].

The following example specifies RSA security and TCP/IP for C/C++ UltraLite.

```
ul_synch_info synch_info;
...
synch_info.stream = ULSocketStream();
synch_info.security = ULSecureRSATLSStream();
synch_info.security_parms = TEXT(
      "port=9999;
       certificate_company=Sybase, Inc.;
       certificate_unit=IAS;
       certificate_name=MobiLink");
```

The certificate_company, certificate_unit, and certificate_name parameters are used to verify certificate fields.

☞ For more information about verifying certificate fields, see "Verifying certificate fields" on page 180.

You can also specify the trusted_certificates security parameter, which overrides any trusted certificate information embedded in the UltraLite schema (step 1 of this procedure). You can only use this parameter for Windows NT/2000/XP or Windows CE.

```
ul_synch_info synch_info;
...
synch_info.stream = ULSocketStream();
synch_info.security = ULSecureRSATLSStream();
synch_info.security_parms = TEXT(
      "port=9999;
       trusted_certificates=\rsaroot.crt;
       certificate_company=Sybase, Inc.;
       certificate_unit=IAS;
       certificate_name=MobiLink");
```

For more information about the ul_synch_info security_parms field, see "security_parms synchronization parameter" [*UltraLite C/C++ User's Guide,* page 435].

- **UltraLite interfaces using ActiveSync** For information about using Transport-Layer Security and ActiveSync, see "ActiveSync protocol options" [*MobiLink Clients,* page 341].
- **UltraLite interfaces using HotSync** For information about using

Transport-Layer Security and HotSync, see "HotSync protocol options" [*MobiLink Clients,* page 343].

CHAPTER 11

MobiLink Synchronization Server Options

About this chapter

This chapter describes the options that can be set when starting the MobiLink synchronization server, dbmlsrv9.

Contents

MobiLink synchronization server

The MobiLink synchronization server lets you synchronize remote databases or applications with an ODBC-compliant consolidated database.

Function

Start a MobiLink synchronization server.

Syntax

dbmlsrv9 -c "*connection-string***"** [*options*]

Option	Description
@*data*	Read in options from the specified environment variable or configuration file. See "@data option" on page 194.
–a	Disable automatic reconnection upon synchronization error. See "-a option" on page 194.
-b	Trim blank padding of strings. See "-b option" on page 195.
–bc *size*	Specify the amount of memory to reserve for BLOB caching. See "-bc option" on page 196.
–bn *size*	Specify the maximum number of bytes to consider when comparing BLOBs for conflict detection. See "-bn option" on page 196.
–c *"keyword=value; . . . "*	Supply ODBC database connection parameters for your consolidated database. See "-c option" on page 196.
–cn *connections*	Set the maximum number of simultaneous connections with the consolidated database server. See "-cn option" on page 197.
–cr *count*	Set the maximum number of database connection retries. See "-cr option" on page 197.
–ct *connection-timeout*	Set the length of time a connection may be unused before it is timed out. See "-ct option" on page 198.
–d *number*	Specify the size of the download cache. See "-d option" on page 198.

Option	Description
-dd *directory*	Specify where download streams are stored. See "-dd option" on page 198.
–dl	Display all log messages on the console. See "-dl option" on page 199.
-ds *size*	Specify the amount of disk storage that can be used to store restartable download streams. See "-ds option" on page 199.
–e *filename*	Store remote error logs sent into the named file. See "-e option" on page 199.
–et *filename*	Truncate the file and append remote synchronization logs to the new file. See "-et option" on page 200.
–f	Assume synchronization scripts do not change. See "-f option" on page 200.
–fr	If table data scripts are missing, synchronization will not abort, but just issue a warning. See "-fr option" on page 200.
-m [*filename*]	Enables QAnywhere messaging. See "-m option" on page 201.
-notifier	Starts a Notifier for server-initiated synchronization. See "-notifier option" on page 202.
–o *logfile*	Log messages to a file. See "-o option" on page 203.
-on *size*	Set maximum size for log file. See "-on option" on page 204.
–oq	Prevent the popup dialog on startup error. See "-oq option" on page 205.
–os *size*	Maximum size of output file. See "-os option" on page 205.
–ot *logfile*	Log messages to a file, but truncate it first. See "-ot option" on page 206.
–ps *num*	Set maximum number of prepared statements to cache per connection. See "-ps option" on page 206.

Option	Description
–q	Minimize the synchronization server window. See "-q option" on page 206.
–r *retries*	Retry deadlocked uploads at most this many times. See "-r option" on page 206.
–rd *delay*	Set maximum delay, in seconds, before retrying a deadlocked transaction. See "-rd option" on page 207.
–s *count*	Specify the maximum number of rows to be fetched or sent at once. See "-s option" on page 207.
–sl dnet *script-options*	Set the .NET CLR options and force loading of the virtual machine on startup. See "-sl dnet option" on page 207.
–sl java *script-options*	Set the Java virtual machine options and force loading of the virtual machine on startup. See "-sl java option" on page 209.
–t *ODBC-output-file*	Log ODBC calls issued by MobiLink to this file. See "-t option" on page 210.
–tt *ODBC-output-file*	Log ODBC calls issued by MobiLink to this file, but first delete the file if it exists. See "-tt option" on page 210.
–u *size*	Specify the amount of memory to reserve for caching upload streams. See "-u option" on page 211.
–ud	On UNIX platforms, run as a daemon. See "-ud option" on page 211.
-us	Prevents MobiLink from invoking upload scripts for tables for which there is no upload. See "-us option" on page 211.
–v [*levels*]	Controls the type of messages written to the log file. See "-v option" on page 211.
–w *count*	Set the number of worker threads. See "-w option" on page 212.

Option	Description
–wu *count*	Set the maximum number of worker threads permitted to process uploads concurrently. See "-wu option" on page 214.
–x *protocol*[**(***network-parameters***)**]	Specify the communications protocol. Optionally, specify network parameters in form *parameter*=*value*, with multiple parameters separated by semicolons. See "-x option" on page 214.
–za	Allow generation of active scripts. See "-za option" on page 219.
–ze	Allow generation of sample scripts. See "-ze option" on page 220.
–zp	In the event of a timestamp conflict between the consolidated and remote database, this option allows timestamp values with a precision higher than the lowest-precision to be used for conflict detection purposes. See "-zp option" on page 221.
–zs *name*	Specify a server name. See "-zs option" on page 221.
–zt *number*	Specify the maximum number of processors used to run the MobiLink synchronization server. See "-zt option" on page 221.
–zu { + | – }	Controls the automatic addition of users when the authenticate_user script is undefined. See "-zu option" on page 222.
-zw 1,. . . 5	Controls which levels of warning message to display. See "-zw option" on page 222.
-zwd *code*	Disables specific warning codes. See "-zwd option" on page 223.
-zwe *code*	Enables specific warning codes. See "-zwe option" on page 223.

Description

The MobiLink synchronization server opens connections, via ODBC, with your consolidated database server. It then accepts connections from client applications and controls the synchronization process.

The MobiLink synchronization server is compatible with a variety of database-management systems, including Adaptive Server Anywhere, Adaptive Server Enterprise, Oracle, Microsoft SQL Server, and IBM DB2.

You must supply connection parameters for the consolidated database using the –c option. The command line options may be presented in any order. The –c option is shown here as the first item in a command string as a convention only. It can be anywhere in a list of options, but must precede a connection string.

Unless your ODBC data source is configured to automatically start the consolidated database, the database must be running before you start the MobiLink server.

dbmlsrv9 options

This section lists all MobiLink synchronization server command line options.

@data option

Function — Reads in options from the specified environment variable or configuration file.

Syntax — **dbmlsrv9 -c "***connection-string***"** *@data* . . .

Description — Use this option to read in dbmlsrv9 command line options from the specified environment variable or configuration file. If both exist with the same name that is specified, the environment variable is used.

☞ For more information about configuration files, see “Using configuration files” [*ASA Database Administration Guide,* page 495].

If you want to protect passwords or other information in the configuration file, you can use the File Hiding utility to obfuscate the contents of the configuration file.

☞ See “Hiding the contents of files using the dbfhide command-line utility” [*ASA Database Administration Guide,* page 524].

-a option

Function — Instructs the MobiLink synchronization server to not reconnect on synchronization error.

Syntax — **dbmlsrv9 -c "***connection-string***" -a** . . .

Description — Should an error occur during synchronization, the MobiLink synchronization server automatically disconnects from the consolidated database, and then re-establishes the connection. Reconnecting ensures that the following synchronization starts from a known state. When this behavior is not required, you can use this option to disable it. The maintenance of state information depends on programmer requirements and may vary depending on the ways in which the programmer configures MobiLink scripting to work with the DBMS. This applies even if that database is an Oracle, Adaptive Server Anywhere database, or other supported product. Some status information may need to be re-initialized depending on the client application.

-b option

Function

For columns of type VARCHAR, CHAR, LONG VARCHAR, or LONG CHAR, removes trailing blanks from strings during synchronization.

Syntax

dbmlsrv9 -c "*connection-string***" -b** . . .

Description

This option is intended to help resolve differences between the Adaptive Server Anywhere CHAR data type and the CHAR or VARCHAR data type used by the consolidated database. The Adaptive Server Anywhere CHAR data type is equivalent to VARCHAR. However, in most consolidated databases that are not Adaptive Server Anywhere, the CHAR(n) data type is blank-padded to n characters.

When -b is specified, the MobiLink synchronization server removes trailing blanks from strings for columns of type CHAR, VARCHAR, LONG CHAR, or LONG VARCHAR if the column on the remote is a string. It does this before filtering rows that were uploaded in the current synchronization. The trimmed data is then downloaded to the remote databases.

This option can also be used to detect conflict updates. For each upload update row, the MobiLink synchronization server fetches the row from the consolidated database for the given primary key, compares the row with the pre-image of the update, and then determines whether the update is a conflict update. When -b is used, MobiLink trims trailing blanks from columns of type CHAR, VARCHAR, LONG CHAR, or LONG VARCHAR before doing the comparison.

Example

If the -b option is not used, a primary-key value of ‘abc’ uploaded from an Adaptive Server Anywhere or UltraLite remote to a CHAR(10) column in the consolidated database will become ‘abc’ followed by seven blank spaces. If the same row is downloaded, then it will appear on the remote as ‘abc’ followed by seven spaces. If the remote database is not blank-padded, then the remote will now have two rows: both ‘abc’ and ‘abc’ followed by seven spaces. There is now a duplicate row on the remote.

If the -b option is used, a primary-key value of ‘abc’ uploaded from an Adaptive Server Anywhere or UltraLite remote to a CHAR(10) column in the consolidated database will become ‘abc’ followed by seven spaces. Seven spaces still pad the value to ten characters, but if the same row is downloaded, then MobiLink server will strip the trailing spaces, and the value will appear on the remote as ‘abc’. The -b option thus fixes the duplicate row problem.

-bc option

Function

Sets the BLOB cache size.

Syntax

dbmlsrv9 -c "*connection-string***" -bc** *size* [**k** | **m** | **g**] . . .

Description

The amount of memory to use for caching BLOBs. If more memory is required, the MobiLink synchronization server uses disk space, instead. For this reason, too small a value can degrade performance. To calculate the minimum recommended size, multiply the maximum size of all BLOB data in any one row by the number of worker threads, then multiply the result by 4, which provides for a large reserve of memory.

The *size* is the amount of memory to reserve in bytes. Use the suffix k, m, or g to specify units of kilobytes, megabytes, or gigabytes, respectively. The default is 524 288 bytes.

-bn option

Function

Sets the maximum number of BLOB bytes to compare during conflict detection.

Syntax

dbmlsrv9 -c "*connection-string***" -bn** *size*. . .

Description

When two BLOBs contain similar or identical values, the operation of comparing them for filtering or conflict detection can be expensive due to the amount of data involved. This option tells the MobiLink synchronization server to consider only the first *size* bytes of two BLOBs when making the comparison. The default is to compare the two BLOBs in their entirety.

Under some situations, limiting the maximum amount of data compared can speed synchronization substantially; however, it can also cause errors. For example, if two large BLOBs differ only in the last few bytes, the MobiLink synchronization server may consider them identical when in fact they are not.

-c option

Function

Specifies connection parameters for the consolidated database.

Syntax **dbmlsrv9 -c "***connection-string***"**. . .

Description The connection string must give the MobiLink synchronization server information sufficient to connect to the consolidated database. The connection string is required.

The connection string must specify connection parameters in the form *keyword=value*, separated by semicolons, with no spaces between parameters.

Connection parameters must be included in the ODBC data source specification if not given in the command line. Check your RDBMS and ODBC data source to determine required connection data.

☞ For a complete list of SQL Anywhere connection parameters, see "Connection parameters" [*ASA Database Administration Guide,* page 176].

☞ For information about how to hide the password, see "The File Hiding utility" [*ASA Database Administration Guide,* page 524].

Example

```
dbmlsrv9 -c "dsn=odbcname;uid=DBA;pwd=sql"
```

-cn option

Function Sets the maximum number of simultaneous consolidated database connections.

Syntax **dbmlsrv9 -c "***connection-string***" -cn** *value*. . .

Description Specifies the maximum number of simultaneous connections that the MobiLink synchronization server should make to the consolidated database. The minimum and the default value are one greater than the number of worker threads. A warning is issued if the supplied value is too small.

A value larger than the number of worker threads may speed performance, particularly if connecting to the consolidated database is slow or if multiple script versions are in use. The optimum maximum number of database connections is the number of script versions times the number of worker threads, plus one. Connections above this optimum value will not necessarily speed synchronization, and will needlessly consume resources in both the MobiLink synchronization server and the consolidated database server.

-cr option

Function Sets the maximum number of database connection retries.

Syntax **dbmlsrv9 -c "***connection-string***" -cr** *value*. . .

Description — Set the maximum number of times that the MobiLink synchronization server will attempt to connect to the database, before quitting, when a connection goes bad. The default value is three connection retries.

-ct option

Function — Sets the length of time, in minutes, that a connection may be unused before it is timed out and disconnected by the MobiLink synchronization server.

Syntax — **dbmlsrv9 -c "***connection-string***" -ct** *connection-timeout*. . .

Description — MobiLink database connections that go unused for a specified amount of time are freed by the server. The timeout can be set using the -ct option. A default timeout period of 60 minutes is used.

-d option

Function — Sets the size of the download cache.

Syntax — **dbmlsrv9 -c "***connection-string***" -d** *number* [**k** | **m** | **g**]. . .

Description — When no download acknowledgement is required, MobiLink buffers the download stream in a download cache. Since no acknowledgement is required from the client to commit the download transaction, the buffered download stream is sent to the client after the commit.

Use the suffix k, m, or g to specify units of kilobytes, megabytes, or gigabytes, respectively. The default size for the download cache is 0.5 megabytes.

-dd option

Function — For use with restartable downloads. Specifies where download streams are stored.

Syntax — **dbmlsrv9 -c "***connection-string***" -dd** *directory* . . .

Description — When the MobiLink synchronization server is shut down, the download directory is cleaned. This means that a download cannot be restarted across a server restart.

The default directory is your TEMP directory.

See also — "Resuming failed downloads" on page 74

"-dc option" [*MobiLink Clients,* page 103]

"-ds option" on page 199

-dl option

Function	Displays all log messages on screen.
Syntax	**dbmlsrv9 -c "***connection-string***" -v -dl** ...
Description	Display all log messages in the MobiLink synchronization server window. By default, only a subset of all messages is shown in the window when a log file is being output (using -o). In circumstances with many messages, this option can degrade performance.

-ds option

Function	For use with restartable downloads. Specifies the amount of disk space that can be used to store the restartable download stream.
Syntax	**dbmlsrv9 -c "***connection-string***" -ds** *size* [**k** | **m** | **g**]. . .
Description	Use the suffix k, m, or g to specify units of kilobytes, megabytes, or gigabytes, respectively. The default is 10 Mb.
See also	"Resuming failed downloads" on page 74 "-dc option" [*MobiLink Clients,* page 103] "-dd option" on page 198

-e option

Function	Stores error logs sent from Adaptive Server Anywhere MobiLink clients.
Syntax	**dbmlsrv9 -c "***connection-string***" -e** *filename*. . .
Description	With no -e option, error logs from Adaptive Server Anywhere MobiLink clients are stored in a file named *dblmsrv.mle*. The -e option instructs the MobiLink synchronization server to store the error logs in the named file. By default, dbmlsync sends, on the occurrence of an error on the remote site, up to 32 kilobytes of remote log messages to a MobiLink synchronization server. This option provides centralized access to remote error logs to help diagnose synchronization issues. The amount of information delivered from a remote site can be controlled by the dbmlsync extended option ErrorLogSendLimit.
See also	♦ "-et option" on page 200 ♦ "ErrorLogSendLimit (el) extended option" [*MobiLink Clients,* page 117]

-et option

Function	Stores error logs sent from Adaptive Server Anywhere MobiLink clients in the named file after truncating the existing file.
Syntax	**dbmlsrv9 -c "***connection-string***" -et** *filename*. . .
Description	The -et option is the same as the -e option, except that the error log file is truncated before any new errors are added to it. The amount of information delivered from a remote site can be controlled by the dbmlsync extended option ErrorLogSendLimit.
See also	♦ "ErrorLogSendLimit (el) extended option" [*MobiLink Clients*, page 117] ♦ "-e option" on page 199

-f option

Function	Loads synchronization scripts only once, for better performance.
Syntax	**dbmlsrv9 -c "***connection-string***" -f**. . .
Description	Without the -f option, the MobiLink synchronization server checks to see if synchronization scripts have changed during regular operation. This checking is helpful during development, but can have an unnecessary performance impact in a production environment. With the -f option, the MobiLink synchronization server loads the synchronization scripts only once per MobiLink session..

-fr option

Function	If table data scripts are missing, synchronization will not abort, but just issue a warning.
Syntax	**dbmlsrv9 -c "***connection-string***" -fr**. . .
Description	Without the -fr option, the MobiLink synchronization server aborts if a synchronization does not include at least one script that uploads or downloads data. This option causes MobiLink to issue a warning instead of aborting. For bi-directional or upload-only synchronization, -fr will cause an error to occur if data is uploaded and there is no corresponding upload script: ♦ if you attempt to upload an INSERT with no upload_insert, upload_new_row_insert, upload_cursor, or new_row_cursor script defined

- if you attempt to upload a DELETE with no upload_delete, upload_old_row_insert, upload_cursor, or old_row_cursor script defined

  if you attempt to upload an UPDATE with no upload_update, upload_new_row_insert, upload_cursor, or new_row_cursor script defined

For download-only synchronization, -fr will cause an error if any tables in the synchronization are missing a download script (download_cursor or download_delete_cursor).

-m option

Function

Enables QAnywhere messaging.

Syntax

dbmlsrv9 -c "*connection-string***" -m** [*message-properties-file*] . . .

Description

The optional *message-properties-file* specifies property settings for QAnywhere messaging, including the name of files that specify property settings for connectors and transmission rules.

In the *message-properties-file*, each property must appear on its own line and consist of a property name, the = character, and then a property value.

Following are the properties you can set in the QAnywhere properties file:

- **ianywhere.qa.server.logLevel** The logging level of the messaging. The property value may be one of 1, 2, 3, or 4. 1 indicates that only message errors are logged. 2 additionally causes warnings to be logged. 3 additionally causes informational messages to be logged. 4 additionally causes more verbose informational messages to be logged, including details about each QAnywhere message that is transmitted with the MobiLink synchronization server. The default is 2.

  These logging messages are output to the MobiLink synchronization server console. If the dbmlsrv9 -o or -ot option was specified, the messages are output to the MobiLink synchronization server log file.

- **ianywhere.qa.compressionLevel** The default amount of compression applied to each message received by a QAnywhere connector. The compression is an integer between 0 and 9, with 0 being no compression and 9 being the most compression. The default is 0.

  If you also set the compression level for a connector in the connector properties file, this setting is overridden for that connector. For more information, see "Configuring the JMS connector properties file" [*QAnywhere User's Guide*, page 43].

- **ianywhere.qa.server.connectorPropertiesFiles** A list of one or more files that specify the configuration of QAnywhere connectors to an external message system such as JMS. The default is no connectors.

For more information, see "Using JMS Connectors" [*QAnywhere User's Guide,* page 42].

- **ianywhere.qa.server.transmissionRulesFile** A file used to specify rules for governing the transmission and persistence of messages. By default, there are no filters for messages, and messages are deleted when the final status of the message has been transmitted to the message originator.

- **ianywhere.qa.server.autoRulesEvaluationPeriod** The time in milliseconds between evaluations of rules, including message transmission and persistence rules. Since, typically, rules are evaluated on the fly as messages are transmitted to the server store, the rule evaluation period is only for rules that are timing-sensitive. The default value is 60000 (one minute).

See also

- "Introduction to QAnywhere" [*QAnywhere User's Guide,* page 1]
- "QAnywhere Transmission Rules" [*QAnywhere User's Guide,* page 101]
- "Configuring the JMS connector properties file" [*QAnywhere User's Guide,* page 43]
- "Starting the MobiLink synchronization server for QAnywhere messaging" [*QAnywhere User's Guide,* page 33]
- "Starting the MobiLink server for JMS integration" [*QAnywhere User's Guide,* page 43]

Example

For example, the following lines could be contained in a QAnywhere message properties file called *qanymsgprop.ini*; it tells QAnywhere to:

- read connector properties from a file called *connector.ini*
- read server transmission rules from a file called *rules.ini*
- set the logging level to 4 (verbose)

```
ianywhere.qa.server.connectorPropertiesFiles=connector.ini
ianywhere.qa.server.transmissionRulesFiles=rules.ini
ianywhere.qa.server.logLevel=4
```

To run these settings, start the MobiLink synchronization server with the -m option and the message properties filename. Following is a partial command line:

```
dbmlsrv9 -m qanymsgprop.ini ...
```

-notifier option

Function

Starts the Notifier for server-initiated synchronization.

Syntax

dbmlsrv9 -c "*connection-string***" -notifier** [*notifier-properties-file*] . . .

Description

If you specify a Notifier configuration file name, or if you do not specify a file name but you have a default Notifier properties file called *config.notifier*, the Notifier is configured using that file. This overrides any configuration information that is stored in the ml_properties table in the consolidated database.

Otherwise, MobiLink uses the configuration information that is stored in the ml_properties table in the consolidated database.

When you use the -notifier option, you start every Notifier that you have enabled.

For more information about enabling Notifiers, see "enable property" [*MobiLink Server-Initiated Synchronization User's Guide,* page 61].

The -notifier option cannot be used on AIX 4.3.3. For more information, see *http://www-106.ibm.com/developerworks/eserver/articles/JavaPart1.html*.

See also

- ♦ "Setting properties in more than one place" [*MobiLink Server-Initiated Synchronization User's Guide,* page 15]
- ♦ "Notifier properties file" [*MobiLink Server-Initiated Synchronization User's Guide,* page 16]
- ♦ "Notifiers" [*MobiLink Server-Initiated Synchronization User's Guide,* page 18]
- ♦ "MobiLink Notification Properties" [*MobiLink Server-Initiated Synchronization User's Guide,* page 55]

-o option

Function

Logs output messages to a MobiLink synchronization server message log file, and limits the data logged to the console window.

Syntax

dbmlsrv9 -c "*connection-string***" -o** *logfile*. . .

Description

Write all log messages to the specified file. Note that the MobiLink synchronization server window, if present, usually shows a subset of all messages logged.

The MobiLink synchronization server gives the full error context in its output file if errors occur during synchronization. The error context may include the following information:

- ♦ **User Name** This is the actual user name that is provided by MobiLink Adaptive Server Anywhere applications during synchronization.
- ♦ **Modified User Name** This is the user name as modified by the modify_user script.

- **Transaction** This lists the transaction the error occurs in. The transaction could be authenticate_user, begin_synchronization, upload, prepare_for_download, download, or end_synchronization.
- **Table Name** This shows the table name if it is available or NULL.
- **Row Operation** The operation could be INSERT, UPDATE, DELETE or FETCH.
- **Row Data** This shows all the column values of the row that caused the error.
- **Script Version** This is the script version currently used for synchronization.
- **Script** This is the script that caused the error.

Error context information appears in the log regardless of your chosen level of verbosity.

See also

- "-os option" on page 205
- "-dl option" on page 199
- "-ot option" on page 206
- "-on option" on page 204
- "-v option" on page 211

-on option

Function

Specifies a maximum size for the MobiLink synchronization server message log file, after which the file is renamed with the extension .old and a new file is started.

Syntax

dbmlsrv9 -c "*connection-string***" -on** *size* [**k** | **m**]. . .

Description

The *size* is the maximum file size for the output log, in bytes. Use the suffix k or m to specify units of kilobytes or megabytes, respectively. The minimum size limit is 10 Kb.

When the log file reaches the specified size, the MobiLink synchronization server renames the output file with the extension .old, and starts a new one with the original name.

Note
If the .old file already exists, it is overwritten. To avoid losing old log files, use the -os option instead.

This option cannot be used with the -os option.

See also

- "-o option" on page 203

- ♦ “-ot option” on page 206
- ♦ “-on option” on page 204
- ♦ “-os option” on page 205
- ♦ “-v option” on page 211

-oq option

Function

On Windows, prevents the appearance of the error dialog when a startup error occurs.

Syntax

dbmlsrv9 -c "*connection-string***" -oq** . . .

Description

By default, the MobiLink synchronization server displays a message box dialog if a startup error occurs. The -oq option prevents this dialog from being displayed.

-os option

Function

Sets the maximum size of the MobiLink synchronization server message log file, after which a new log file with a new name is created and used.

Syntax

dbmlsrv9 -c "*connection-string***" -os** *size* [**k** | **m**]. . .

Description

The *size* is the maximum file size for logging output messages. The default unit is bytes. Use the suffix k or m to specify units of kilobytes or megabytes, respectively. The minimum size limit is 10 kb.

Before the MobiLink synchronization server logs output messages to a file, it checks the current file size. If the log message will make the file size exceed the specified size, the MobiLink synchronization server renames the message log file to *yymmddxx.mls*, where *xx* is a number from 00 to 99, and *yymmdd* represents the current year, month, and day.

You can use this option to prune old message log files to free up disk space. The latest output is always appended to the file specified by -o or -ot.

You cannot use this option with the -on option.

Note
This option will make an unlimited number of log files. To avoid this situation, use -o or -on.

See also

- ♦ “-o option” on page 203
- ♦ “-on option” on page 204
- ♦ “-ot option” on page 206
- ♦ “-v option” on page 211

-ot option

Function	Logs output messages to the MobiLink synchronization server message log file, but truncates it first.
Syntax	**dbmlsrv9 -c "***connection-string***" -ot** *logfilename* . . .
Description	Truncate the message log file and then append output messages to it. The default is to send output to the screen.
See also	♦ "-on option" on page 204 ♦ "-os option" on page 205 ♦ "-v option" on page 211 ♦ "-o option" on page 203

-ps option

Function	Sets the maximum number of prepared statements to cache per connection.
Syntax	**dbmlsrv9 -c "***connection-string***" -ps** *num* . . .
Description	Controls the maximum number of ODBC prepared statements kept in the prepared statement cache. Caching prepared statements improves performance, but consumes resources. Some consolidated database types have configurable limits on the number of prepared statements, so this option may be set accordingly.

-q option

Function	Instructs MobiLink to run in a minimized window on startup.
Syntax	**dbmlsrv9 -c "***connection-string***" -q** . . .
Description	Minimize the MobiLink synchronization server window.

-r option

Function	Sets the maximum number of deadlock retries.
Syntax	**dbmlsrv9 -c "***connection-string***" -r** *retries* . . .
Description	By default, MobiLink synchronization server retries uploads that are deadlocked, for a maximum of 10 attempts. If the deadlock is not broken, synchronization fails, since there is no guarantee that the deadlock can be overcome. This option allows an arbitrary retry limit to be set. To stop the server from retrying deadlocked transactions, specify **–r 0**. The upper bound on this setting is 2 to the power 32, minus one.

-rd option

Function

Sets the maximum delay time between deadlock retries.

Syntax

dbmlsrv9 -c "*connection-string***" -rd** *delay* . . .

Description

When upload transactions are deadlocked, the MobiLink synchronization server waits a random length of time before retrying the transaction. The random nature of the delay increases the likelihood that future attempts will succeed. This option allows you to specify the maximum delay in units of seconds. The value 0 (zero) makes retries instantaneous, but larger values are recommended because they yield more successful retries. The default and maximum delay value is 30.

-s option

Function

Sets the maximum number of rows fetched, inserted, updated, or deleted at once.

Syntax

dbmlsrv9 -c "*connection-string***" -s** *count* . . .

Description

Set the maximum number of rows transferred between the MobiLink synchronization server and the consolidated database to *count*.

Set this option to no less than the number of rows specified in the ODBC prefetch option, if this option is set. The default value is 10.

The number of rows fetched at once can be viewed in the log file as **rowset size**.

Note: The actual maximum number of rows transferred is influenced by settings in your ODBC data source, and/or database client software.

-sl dnet option

Function

Sets the .NET Common Language Runtime (CLR) options and forces the CLR to load on startup.

Syntax

dbmlsrv9 -c "*connection-string***" -sl dnet** *options* . . .

Description

Sets options to pass directly to the .NET CLR. The options are:

Option	**Description**

Option	**Description**
-D*name*=*value*	Set an environment variable. For example, `-Dsynchtype=far -Dextra_rows=yes` For more information, see the .NET framework class System.Environment.
-MLAutoLoadPath=*path*	Set the location of base assemblies. Only works with private assemblies. To tell MobiLink where assemblies are located, use this option or -MLDomConfigFile, but not both. When you use -MLAutoLoadPath, you cannot specify a domain in the event script. The default is the current directory.
-MLDomConfigFile=*file*	Set the location of base assemblies. Use when you have shared assemblies, or you don't want to load all assemblies in the directory, or you can't use MLAutoLoadPath for some other reason. To tell MobiLink where assemblies are located, use -MLDomConfigFile or -MLAutoLoadPath, but not both.
-MLStartClasses= *classnames*	At server startup, load and instantiate user-defined start classes in the order listed.
-clrConGC	Enable concurrent garbage collection in the CLR.
-clrFlavor=(wks | **svr)**	Flavor of the .NET CLR to load. The flavor is **svr** for server and **wks** for workstation. By default, **wks** is loaded.
-clrVersion=*version*	Version of the .NET CLR to load. This must be prefixed with **v**. For example, **v1.0.3705** loads the directory *\WINNT\Microsoft.NET\Framework\v1.0.3705.*

To display this list of options, use the following command:

```
dbmlsrv9 -sl dnet (?)
```

See also

♦ "Writing Synchronization Scripts in .NET" on page 281

-sl java option

Function

Sets the Java virtual machine options and forces the virtual machine to load on startup.

Syntax

dbmlsrv9 -c "*connection-string***" -sl java (** *options* **)** . . .

Description

Sets -jrepath and other options to pass directly to the Java virtual machine. The options are:

Option	Description
{ **-hotspot** | **-server** | **-classic** }	Override the default choice for the Java VM to use.
{ **–cp** | **–classpath** } *location*;. . .	Specify a set of directories or jar files in which to search for classes. You must enclose -cp or -classpath in either curly braces or round brackets.
–D*name*=*value*	Set a system property. For example, `-Dsynchtype=far -Dextra_rows=yes`
–DMLStartClasses=*class*, . . .	At server startup, load and instantiate user-defined start classes in the order listed.
–jrepath *path*	Override the default JRE path, which is the *sun\jre142* directory under the *Sybase\shared* directory.
–verbose [**:class** |**:gc** | **:jni**]	Enable verbose output.
–X *vm-option*	Set a VM-specific option as described in the file *Xusage.txt*, which by default is installed to *Sybase\Shared\Sun\jre142\bin\hotspot*.

Options must be enclosed in brackets. These can be round brackets, as shown in the syntax above, or curly braces { }.

To display this list of options, use the following command:

```
dbmlsrv9 -sl java (?)
```

To display a list of Java options you can use, type:

```
java
```

Description

The -jrepath option is only available on Windows. On UNIX, if you want to load a specific JRE, you should set the LD_LIBRARY_PATH (LIBPATH on AIX, SHLIB_PATH on HP-UX) to include the directory containing the JRE. The directory must be listed before any of the Adaptive Server Anywhere install directories.

See also

- "Writing Synchronization Scripts in Java" on page 255

Examples

For example, the following partial dbmlsrv9 command line sets the Java virtual machine option that enables system asserts:

```
dbmlsrv9 -sl java {-cp ;\myclasses; -esa} ...
```

The following partial dbmlsrv9 command line defines the LDAP_SERVER system property:

```
dbmlsrv9 -sl java { -cp ;\myclasses; -DLDAP_SERVER=huron-ldap }
          ...
```

-t option

Function

Creates a file containing all the ODBC calls issued by MobiLink.

Syntax

dbmlsrv9 -c "*connection-string***" -t** *ODBC-output-file* . . .

Description

This option can be used to create a file containing all of the ODBC calls issued by MobiLink. If used on UNIX, with the Adaptive Server Anywhere driver used as a driver manager, this feature is ignored. The feature is useful for tracing what was called, passed, and retrieved. It has a severe impact on performance, so should not be used in production.

To prevent the file from becoming large, use the "-tt option" on page 210.

See also

- "-tt option" on page 210

-tt option

Function

Logs ODBC calls issued by MobiLink to a file. If the file already exists, it first deletes it.

Syntax

dbmlsrv9 -c "*connection-string***" -tt** *ODBC-output-file* . . .

Description

This option is used to create a file containing all of the ODBC calls issued by MobiLink. If used on UNIX with the Adaptive Server Anywhere driver used as a driver manager, this feature is ignored. The feature is useful for tracing what was called, passed, and retrieved. It has a severe impact on performance, so should not be used in production.

See also

- "-t option" on page 210

-u option

Function	Sets the upload cache size.
Syntax	**dbmlsrv9 -c "***connection-string***" -u** *size*[**k** | **m** | **g**] . . .
Description	The amount of space, in bytes, to reserve for caching upload streams that are being processed. Use the suffix k, m, or g to specify units of kilobytes, megabytes, or gigabytes respectively. You should consider enlarging this value if your clients upload large streams, or many clients synchronize at once, or both. The suggested size is the maximum expected size of an upload stream multiplied by the number of worker threads. The default value is 500 Kb.
See also	♦ "-bc option" on page 196

-ud option

Function	Instructs MobiLink to run as a daemon.
Syntax	**dbmlsrv9 -c "***connection-string***" -ud** . . .
Description	UNIX platforms only.
See also	♦ "Running MobiLink Outside the Current Session" on page 157

-us option

Function	Prevents the MobiLink synchronization server from invoking upload scripts for a table when no rows are uploaded for the table.
Syntax	**dbmlsrv9 -c "***connection-string***" -us** . . .
Description	This option is particularly useful when dbmlsync transaction-level uploads are used.
See also	"-tu option" [*MobiLink Clients*, page 147]

-v option

Function	Allows you to specify what information is logged to the message log file and displayed in the synchronization window.
Syntax	**dbmlsrv9 -c "***connection-string***" -v**[*levels*] . . .
Description	This option is particularly useful when dbmlsync transaction-level uploads are used.

This option controls the type of messages written to the message log file.

If you specify –v alone, the MobiLink synchronization server writes a minimal amount of information about each synchronization.

The values of levels are as follows. You can use one or more of these options at once; for example, -vnrsu.

- **+** Turn on all logging options that increase verbosity.
- **c** Show the content of each synchronization script when it is invoked. This level implies s.
- **f** Show first-read errors. This will log errors caused when load-balancing devices check for server liveness by making connections that don't send any data, and thus result in failed synchronizations.

  For TCP/IP connections, you might be better off using the TCP/IP option **ignore**. For more information, see "-x option" on page 214.
- **h** Show the remote schema as uploaded during synchronization.
- **n** Show row-count summaries.
- **p** Show progress offsets.
- **r** Display the column values of each row uploaded or downloaded.
- **s** Show the name of each synchronization script as it is invoked.
- **t** Show the translated SQL that results from scripts that are written in ODBC canonical format. This level implies c. The following example shows the automatic translation of a statement for Adaptive Server Anywhere.

  ```
  I. 02/11 11:02:14. [102]: begin_upload synch2
  { call SynchLogLine( ?, ?, 'begin_upload' ) }
  I. 02/11 11:02:14. [102]: Translated SQL:
  call SynchLogLine( ?, ?, 'begin_upload' )
  ```

  The following example shows the translation of the same statement for Microsoft SQL Server.

  ```
  I. 02/11 11:03:21. [102]: begin_upload synch2
  { call SynchLogLine( ?, ?, 'begin_upload' ) }
  I. 02/11 11:03:21. [102]: Translated SQL:
  EXEC SynchLogLine ?, ?, 'begin_upload'
  ```
- **u** Show undefined table scripts. This may help new users understand the synchronization process.

-w option

Function Sets the number of worker threads.

Syntax

dbmlsrv9 -c "*connection-string***" -w** *count* . . .

Description

Each worker thread accepts synchronization requests one at a time. Each worker thread is associated with a network protocol. If you have more than one protocol defined, the worker threads are divided evenly among the protocols.

Each worker thread uses one connection to the consolidated database. The MobiLink synchronization server opens one additional connection for administrative purposes. Hence, the minimum number of connections from the MobiLink synchronization server to the consolidated database is *count* + 1.

The number of worker threads has a strong influence on MobiLink synchronization throughput, and you need to run tests to determine the optimum number for your particular synchronization setup. The number of worker threads determines how many synchronizations can be active simultaneously; the rest will be queued waiting for worker threads to become available. Thus adding worker threads should increase throughput, but it will also increase the possibility of contention between the active synchronizations. At some point adding more worker threads will decrease throughput, because the increased contention outweighs the benefit of overlapping synchronizations.

☞ For more information, see the MobiLink Performance whitepaper at *http://my.sybase.com/detail?id=1009664*, and "MobiLink Performance" on page 105.

The value set for this option is also the default setting for the -wu option, which can be used to limit the number of threads that can simultaneously upload. This is useful if the optimum number of worker threads for downloading is larger than the optimum number for uploading, as is typically the case with remote databases on slow computers or with slow connections to the MobiLink server. Tests have shown that for slow synchronization clients (such as Palm devices or computers connected by dialup), the best throughput is achieved with a large number of worker threads (via -w) with a small number allowed to apply uploads simultaneously (via -wu). In general, the optimum number for -wu depends on the consolidated database, and is relatively independent of the processing or network speeds for the remote databases. Therefore, when you increase the number of threads with -w, you may want to use -wu to restrict the number that can upload simultaneously. For more information, see "-wu option" on page 214.

The default number of worker threads is 5.

-wu option

Function	Sets the maximum number of worker threads that can apply uploads simultaneously.
Syntax	**dbmlsrv9 -c "***connection-string***" -wu** *count* . . .

Description

Use the -wu option to limit the number of worker threads that can simultaneously apply uploads. When the limit is reached, a worker thread that is ready to apply its upload must wait until another finishes its upload. The excess worker threads are still free to receive uploads or to download.

The most common cause of contention in the consolidated database is having too many worker threads applying uploads simultaneously. This can be an issue when the network connection is slow, or when the client device has low processing speed. For example, when working over a wide-area wireless network or using a Palm device you may want to increase the total number of threads (-w) but limit the number that can apply uploads simultaneously.

Consider the following example. In a pilot setup using a LAN and remote databases on PCs, you find that the optimum number of worker threads is approximately 10 for both upload-only and download-only synchronizations, and that corresponds to 100% CPU utilization on the consolidated database. With fewer worker threads you find that throughput is less and the CPU utilization for the consolidated database is lower. With more worker threads, throughput does not increase because the consolidated database is already processing as fast as it can with 10 workers. When you switch to using a dialup network with 10 MobiLink worker threads, you will probably find that throughput is lower and the consolidated CPU utilization has dropped. You may find that you can get throughput (and consolidated CPU utilization) to approach the values obtained with the LAN by increasing the number of worker threads (via -w) while keeping the number that apply uploads simultaneously limited to 10 (via -wu).

By default, all worker threads can apply uploads simultaneously. The number of worker threads that are used is set by the -w option. The default is 5.

-x option

Function	Sets network protocol and protocol options for MobiLink clients. These are used by the MobiLink synchronization server to listen for synchronization requests.
Syntax	**dbmlsrv9 -c "***connection-string***"** **-x** *protocol*[*protocol-options*]. . .

protocol-options : **(** *keyword***=***value*;... **)**

Description

Specify communications protocol through which to communicate with client applications. The default is TCPIP with port 2439.

Note for UltraLite users
If you are using an UltraLite Java application and you are using TLS security, the syntax of -x is slightly different. For details, see "Using transport-layer security" [*UltraLite Static Java User's Guide,* page 42].

The allowed values of *protocol* are as follows:

- **tcpip** Accept connections from applications via TCP/IP.
- **http** Accept connections via the standard Web protocol. Client applications can pick their HTTP version and the MobiLink synchronization server adjusts on a per-connection basis.
- **https** Accept connections via a variant of HTTP that handles secure transactions. The HTTPS stream implements HTTP over SSL/TLS using RSA encryption, and is compatible with any other HTTPS server.
- **https_fips** Accept connections using the HTTPS protocol and FIPS-approved algorithms for encryption. HTTPS_FIPS uses separate FIPS 140-2 certified software. Servers using rsa_tls are compatible with clients using rsa_tls_fips, and servers using rsa_tls_fips are compatible with clients using rsa_tls. rsa_tls_fips can only be used with Adaptive Server Anywhere databases on Windows.

Optionally, you can also specify network protocol options, in the form *option=value*. You should separate multiple options with semicolons. Which options you specify depends on the protocol you choose.

- **TCP/IP options** If you specify the **tcpip** protocol, you can optionally specify the following protocol options:
  - **backlog=number-of-connectionsă** The maximum number of remote connections before MobiLink should reject new synchronization requests, causing synchronization to fail on the client side. By specifying a backlog size, you can prevent clients from waiting to synchronize when the server is busy. If you do not specify a backlog size, clients will attempt to synchronize regardless of the size of the backlog.
  - **host=hostname** The host name or IP number on which the MobiLink synchronization server should listen. The default value is **localhost**.

- **ignore=hostname** A host name or IP number that will be ignored by the MobiLink synchronization server if it makes a connection. This option allows you to ignore requests from load balancers at the lowest possible level, preventing excessive output in the MobiLink server log and MobiLink Monitor output files. You can specify multiple hosts to ignore; for example `-x tcpip(ignore=lb1;ignore=123.45.67.89)`.
- **liveness_timeout=n** The amount of time, in seconds, after the last communication with a client before MobiLink aborts the synchronization. A value of 0 means that there is no timeout. This option is only effective if download acknowledgement on the client is set to off (the default). The default is 120 seconds.
- **port=portnumber** The socket port number on which the MobiLink synchronization server should listen. The default port is 2439, which is the IANA registered port number for the MobiLink synchronization server.
- **security=cipher(keyword=value;. . .)** All communication through this connection is to be encrypted and authenticated using transport-layer security. The cipher can be one of **ecc_tls**, **rsa_tls**, or **rsa_tls_fips**, for elliptic-curve encryption, RSA encryption, or RSA encryption that is FIPS-approved, respectively. rsa_tls_fips uses separate FIPS 140-2 certified software from Certicom but is compatible with clients using https (and version 9.0.2 or later). rsa_tls_fips can only be used with Adaptive Server Anywhere clients on Windows.

  For backwards compatibility, **ecc_tls** can also be specified as **certicom_tls**.

  The security parameters are **certificate** (the path and file name of the certificate that is to be used for server authentication), and **certificate_password**. You must use a certificate that matches the cipher suite you choose.

  ☞ For more information, see "Starting the MobiLink synchronization server with transport-layer security" on page 177.

  Separately licensable option required
  Transport-layer security requires that you obtain the separately-licensable SQL Anywhere Studio security option and is subject to export regulations.

  ☞ To order this component, see "Separately-licensable components" [*Introducing SQL Anywhere Studio*, page 5].

♦ **HTTP options** If you specify the **http** protocol, you can optionally specify the following protocol options:

- **backlog=number-of-connectionsǎ** The maximum number of remote connections before MobiLink should reject new synchronization requests, causing synchronization to fail on the client side. By specifying a backlog size, you can prevent clients from waiting to synchronize when the server is busy. If you do not specify a backlog size, clients will attempt to synchronize regardless of the size of the backlog.
- **buffer_size=number** The maximum body size for an HTTP message sent from MobiLink server, in bytes. Changing the option will decrease or increase the amount of memory allocated for sending HTTP messages. The default is 65 535 bytes.
- **contd_timeout=seconds** The number of seconds the MobiLink synchronization server waits to receive the next part of a partially completed synchronization before the synchronization is abandoned. You can tune this option to free MobiLink worker threads when the wait time indicates that the client will never continue the connection. The default value is 30 seconds.
- **host=hostname** The host name or IP number on which the MobiLink synchronization server should listen. The default value is **localhost**.
- **port=portnumber** The socket port number on which the MobiLink synchronization server should listen. The port number must be a decimal number that matches the port the MobiLink synchronization server is setup to monitor. The default port is 80.
- **unknown_timeout=seconds** The number of seconds the MobiLink synchronization server waits to receive HTTP headers on a new connection before the synchronization is abandoned. You can tune this option to free MobiLink worker threads when the wait time indicates that a network failure has occurred. The default value is 30 seconds.
- **url_suffix=suffix** The suffix to add to the URL on the first line of each HTTP request. This parameter can be used to help ensure that a particular client connects to the intended server. Values must match or synchronization will not be successful.
- **version=http-version** The MobiLink synchronization server automatically detects the HTTP version used by a client. This parameter is a string specifying the default version of HTTP to use in case the server cannot detect the method used by the client. You have a choice of **1.0** or **1.1**. The default value is **1.1**.

♦ **HTTPS or HTTPS_FIPS options** The https communication stream uses RSA digital certificates for transport-layer security. The https_fips stream uses separate FIPS 140-2 certified software but is compatible with https.

☞ For more information, see "Starting the MobiLink synchronization server with transport-layer security" on page 177.

Separately licensable option required
Transport-layer security requires that you obtain a separately-licensable SQL Anywhere Studio security option and is subject to export regulations.

☞ To order this component, see "Separately-licensable components" [*Introducing SQL Anywhere Studio*, page 5].

If you specify the **https** protocol, you can optionally specify the following protocol options:

- **backlog=number-of-connectionsă** The maximum number of remote connections before MobiLink should reject new synchronization requests, causing synchronization to fail on the client side. By specifying a backlog size, you can prevent clients from waiting to synchronize when the server is busy. If you do not specify a backlog size, clients will attempt to synchronize regardless of the size of the backlog.
- **buffer_size=numberă** The maximum body size for an HTTPS message sent from MobiLink server, in bytes. Changing the option will decrease or increase the amount of memory allocated for sending HTTPS messages. The default is 65 535 bytes.
- **contd_timeout=seconds** The number of seconds the MobiLink synchronization server waits to receive the next part of a partially completed synchronization before the synchronization is abandoned. You can tune this option to free MobiLink worker threads when the wait time indicates that the client will never continue the connection. The default value is 30 seconds.
- **host=hostname** The host name or IP number on which the MobiLink synchronization server should listen. The default value is **localhost**.
- **port=portnumber** The socket port number on which the MobiLink synchronization server should listen. The port number must be a decimal number that matches the port the MobiLink synchronization server is setup to monitor. The default port is 443.
- **certificate** The path and file name of the certificate that is to be used for server authentication. This must be an RSA certificate.
- **certificate_password** An optional parameter that specifies a password for the certificate.

  ☞ For more information about security, see "MobiLink Transport-Layer Security" on page 165.

- **unknown_timeout=seconds** The number of seconds the MobiLink synchronization server waits to receive HTTP headers on a new connection before the synchronization is abandoned. You can tune this option to free MobiLink worker threads when the wait time indicates that a network failure has occurred. The default value is 30 seconds.
- **url_suffix=suffix** The suffix to add to the URL on the first line of each HTTP request. This parameter can be used to help ensure that a particular client connects to the intended server. Values must match or synchronization will not be successful.
- **version=http-version** The MobiLink synchronization server automatically detects the HTTP version used by a client. This parameter is a string specifying the default version of HTTP to use in case the server cannot detect the method used by the client. You have a choice of **1.0** or **1.1**. The default value is **1.1**.

Examples

The following command line sets the backlog size to 10 connections.

```
dbmlsrv9 -c dsn=asa90sample;uid=DBA;pwd=SQL -x http(backlog=10)
```

-za option

Function

Generates statement-based scripts that perform a simple snapshot synchronization.

Syntax

dbmlsrv9 -c "*connection-string***" -za**

Description

The following scripts are generated:

- upload_insert
- upload_update
- upload_delete
- download_cursor

This option generates active scripts; that is, they are used for the current synchronization. The scripts are also saved and will work for subsequent synchronizations using the same script version. The -za option is typically used for quick demos. To generate scripts as a starting point for writing your own scripts, the dbmlsrv9 -ze option might be more useful.

To generate scripts, you must also specify that the client sends column names. You do this when you initiate synchronization. For Adaptive Server Anywhere remotes, see “SendColumnNames (scn) extended option” [*MobiLink Clients,* page 130]. For UltraLite remotes, see “Send Column Names synchronization parameter” [*MobiLink Clients,* page 330].

The generated scripts perform one-to-one snapshot synchronization using the table and column names sent from the client. If the consolidated database has different table or column names than the remote, activating these scripts will cause an error during the synchronization.

> **Note:**
> Scripts are generated the first time that a remote synchronizes with a script version that doesn't exist. If the given script version already exists, -za has no effect. This means that you cannot use -za to generate scripts one table at a time for the same script version. Using -za, you must generate scripts for all tables and publications at once.

See also

- "-ze option" on page 220

Example

The following dbmlsrv9 command enables automatic script generation. The dbmlsync command sets the necessary SendColumnNames option.

```
dbmlsrv9 -c "dsn=YourDBDSN" -za
dbmlsync -c dsn=dsn_remote -e "SendColumnNames=ON"
```

-ze option

Function

Generates sample scripts that, if activated, perform a simple snapshot synchronization.

Syntax

dbmlsrv9 -c "*connection-string***" -ze**

Description

The following scripts are generated:

- example_upload_insert
- example_upload_update
- example_upload_delete
- example_download_cursor

You can use these scripts as starting points for creating your own scripts.

To generate scripts, you must also specify that the client sends column names. You do this when you initiate synchronization. For Adaptive Server Anywhere remotes, see "SendColumnNames (scn) extended option" [*MobiLink Clients,* page 130]. For UltraLite remotes, see "Send Column Names synchronization parameter" [*MobiLink Clients,* page 330].

The generated scripts perform one-to-one snapshot synchronization using the table and column names sent from the client. If the consolidated database has different table or column names than the remote, activating these scripts will cause an error during the synchronization.

Note:
Scripts are generated only the first time that a remote synchronizes, and only when the given script version does not exist. Otherwise, -ze has no effect.

See also

- "-za option" on page 219

-zp option

Function

Adjusts which timestamp values will be used for conflict detection purposes.

Syntax

dbmlsrv9 -c "*connection-string***" -zp**

Description

In the event of a timestamp conflict between the consolidated and remote database, this option allows timestamp values with a precision higher than the lowest precision to be used for conflict detection purposes. The option is useful when timestamps in the consolidated database are more precise than in the remote, as updated timestamps on the remote can cause conflicts in the next synchronization. The option allows MobiLink to ignore these conflicts. When there is a precision mismatch and -zp is not used, a per synchronization and a schema sensitive per table warning are written to the log to advertise the -zp option. Another per synchronization warning is also added to advise users to adjust the timestamp precision on the remote database where possible.

-zs option

Function

Specifies a MobiLink server name for dbmlstop.

Syntax

dbmlsrv9 -c "*connection-string***" -zs** *name*

Description

The name that is specified may include letters and numbers, but no other characters.

When dbmlstop is used to shut down a MobiLink synchronization server started with the -zs option, you must specify the server name on the dbmlstop command line. For example, `dbmlstop myMLserver`. Shutdown may only be initiated from the computer where the MobiLink synchronization server is installed.

See also

- "MobiLink stop utility" on page 490

-zt option

Function

Specifies the maximum number of processors used to run the MobiLink synchronization server.

Syntax

dbmlsrv9 -c "*connection-string***" -zt** *number*

Description

This option may be required for some ODBC drivers. It also gives you fine control of processor resources.

This option can only be used on Windows operating systems. The default is the number of processors on the computer.

-zu option

Function

Controls the automatic addition of users when the authenticate_user script is undefined.

Syntax

dbmlsrv9 -c "*connection-string***" -zu**{ + | - } . . .

Description

If this is supplied as -zu+, then unrecognized MobiLink user names are added to the ml_user table on first synchronizing. If the argument is supplied as -zu-, or not supplied, unrecognized user names are prevented from synchronizing.

See also

- "Authenticating MobiLink Users" [*MobiLink Clients*, page 9]
- "MobiLink user authentication utility" on page 492
- "authenticate_user connection event" on page 336

-zw option

Function

Controls which levels of warning message to display.

Syntax

dbmlsrv9 -c "*connection-string***" -zw** *levels*

Description

MobiLink has five levels of warning messages:

Level	Description
0	Suppress all warning messages
1	Server and high ODBC level: warning messages when the MobiLink synchronization server starts
2	Synchronization and user level: warning messages when a synchronization starts
3	Schema level: warning messages when a MobiLink synchronization server is processing a client schema
4	Script and lower ODBC level: warning messages when a MobiLink synchronization server fetches, prepares, or executes scripts
5	Table or row level: warning messages when a MobiLink synchronization server performs table operations in an upload or download

To specify the level of warning messages you want reported, you can separate levels with a comma, or separate a range with two dots. For example, **-zw 1..3,5** is the same as **-zw 1,2,3,5**.

The reporting of messages has a slight impact on performance. Levels with a higher number tend to produce more messages.

If -zw is used more than once in the same command line, MobiLink recognizes only the last instance. If settings of -zw, -zwd, and -zwe conflict, MobiLink gives priority to -zwe, then -zwd, then -zw.

The default is **1,2,3,4,5**, which indicates that all levels of warning message should be displayed.

-zwd option

Function

Disables specific warning codes.

Syntax

dbmlsrv9 -c "*connection-string***" -zwd** *code*,. . .

Description

You can disable specific warning codes so that they will not be reported, even though other codes of the same level are reported.

☞ For a complete list of warning message codes, see "MobiLink Synchronization Server Warning Messages" [*ASA Error Messages,* page 509].

If -zwd is used more than once in the same command line, MobiLink accumulates the settings. If settings of -zw, -zwd, and -zwe conflict, MobiLink gives priority to -zwe, then -zwd, then -zw.

-zwe option

Function

Enables specific warning codes.

Syntax

dbmlsrv9 -c "*connection-string***" -zwe** *code*,. . .

Description

You can enable specific warning codes so that they will be reported even though you have disabled other codes of the same level using -zw.

☞ For a complete list of warning message codes, see "MobiLink Synchronization Server Warning Messages" [*ASA Error Messages,* page 509].

If -zwe is used more than once on the same command line, MobiLink accumulates the settings. If settings of -zw, -zwd, and -zwe conflict, MobiLink gives priority to -zwe, then -zwd, then -zw.

PART II

MOBILINK SCRIPTING LOGIC

This part describes how to use MobiLink scripting logic.

CHAPTER 12

Writing Synchronization Scripts

About this chapter

You control the synchronization process by writing synchronization scripts and storing them in the consolidated database.

You can write scripts in SQL, Java, or .NET. This chapter applies to all kinds of scripts, but focuses on how to write synchronization scripts in SQL.

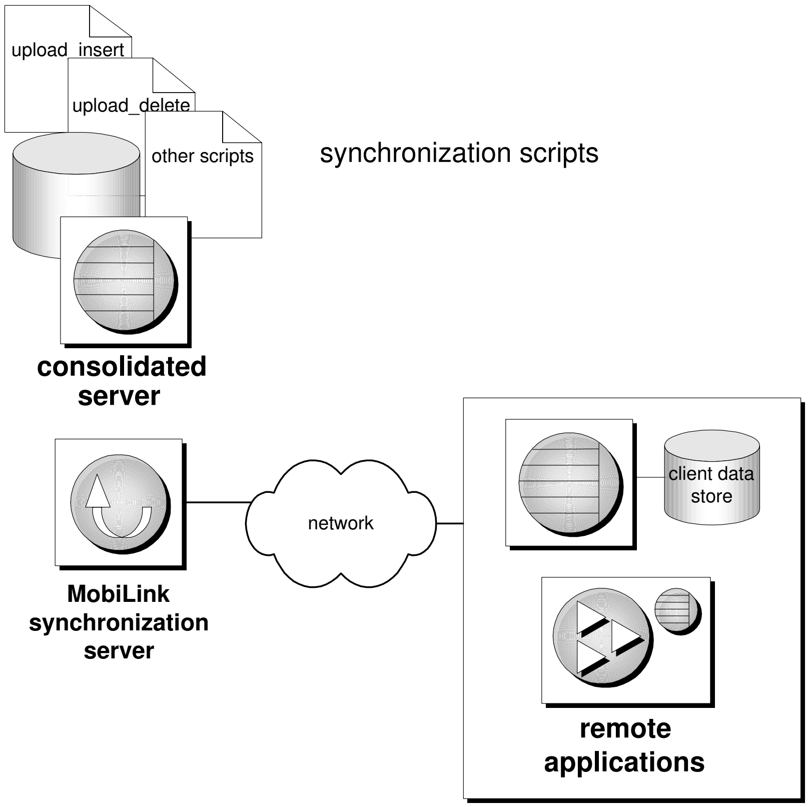
For more information about writing scripts, see

- "Synchronization Events" on page 319
- "Synchronization Techniques" on page 45
- "Writing Synchronization Scripts in Java" on page 255
- "Writing Synchronization Scripts in .NET" on page 281

Contents

Introduction to synchronization scripts

MobiLink Synchronization logic consists of scripts, which may be individual statements or stored procedure calls, stored in your consolidated database. During synchronization, the MobiLink synchronization server reads the scripts and executes them against the consolidated database. Scripts provide you with opportunities to perform tasks at various points of time during the synchronization process. You can use Sybase Central to add scripts to the consolidated database or you can use stored procedures.



The synchronization process is composed of multiple steps. A unique **event name** identifies each step. You control the synchronization process by writing scripts associated with some of these events. You write a script only when some particular action must occur at a particular event. The MobiLink synchronization server executes each script when its associated event occurs. If you do not define a script for a particular event, the MobiLink synchronization server simply proceeds to the next step.

For example, one event is begin_upload_rows. You can write a script and associate it with this event. The MobiLink synchronization server reads this script when it is first needed, and executes it during the upload phase of synchronization. If you write no script, the MobiLink synchronization server proceeds immediately to the next step, which is processing the uploaded rows.

Some scripts, called **table scripts**, are associated not only with an event, but also with a particular table in the remote database. The MobiLink synchronization server performs some tasks on a table-by-table basis; for example, downloading rows. You can have many scripts associated with the same event, but each with different application tables. Alternatively, you can define many scripts for some application tables, but none for others.

☞ For an overview of events, see "The synchronization process" on page 15.

☞ For a description of every script you can write, see "Synchronization Events" on page 319.

You can write scripts in SQL, Java, or .NET. This chapter applies to all kinds of scripts, but focuses on how to write synchronization scripts in SQL.

☞ For a description and comparison of SQL, Java, and .NET, see "Options for writing synchronization logic" on page 25.

☞ For information about writing scripts in .NET, see "Writing Synchronization Scripts in .NET" on page 281.

☞ For information about writing scripts in Java, see "Writing Synchronization Scripts in Java" on page 255.

☞ For information about how to implement synchronization scripts, see "Synchronization Techniques" on page 45.

A simple synchronization script

MobiLink provides many events that you can exploit, but it is not mandatory to provide scripts for each event. In a simple synchronization model, you may need only a few scripts.

Downloading all the rows from the table to each remote database synchronizes the ULProduct table in the CustDB sample application. In this case, no additions are permitted at the remote databases. You can implement this simple form of synchronization with a single script; in this case only one event has a script associated with it.

The MobiLink event that controls the rows to be downloaded during each synchronization is named the download_cursor event. Cursor scripts must contain SELECT statements. The MobiLink synchronization server uses these queries to define a cursor. In the case of a download_cursor script, the cursor selects the rows to be downloaded to one particular table in the remote database.

In the CustDB sample application, there is a single download_cursor script for the ULProduct table in the sample application, which consists of the following query.

```
SELECT prod_id, price, prod_name
FROM ULProduct
```

This query generates a result set. The rows that make up this result set are downloaded to the client. In this case, all the rows of the table are downloaded.

The MobiLink synchronization server knows to send the rows to the ULProduct application table because this script is associated with both the download_cursor event and ULProduct table by the way it is stored in the consolidated database. Sybase Central allows you to make these associations.

Note
In this example, the query selects data from a consolidated table also named ULProduct. The names need not match. You could, instead, download data to the ULProduct application table from any table, or any combination of tables, in the consolidated database by rewriting the query.

You can write more complicated synchronization scripts. For example, you could write a script that downloads only recently modified rows, or one that provides different information to each remote database.

Generating scripts automatically

You can use the dbmlsrv9 -za option to generate default synchronization scripts. The synchronization scripts perform a snapshot synchronization of your consolidated database with your remote database using table and column names that are sent from the client.

To use this feature with Adaptive Server Anywhere clients, set the SendColumnNames extended option to ON to cause dbmlsync to send the column names with the upload header. To use this feature with UltraLite clients, set the send_column_names parameter to ul_true.

When you use -za, scripts are generated the first time that a remote synchronizes with a script version that doesn’t exist. If the given script version already exists, -za has no effect. This means that you cannot use -za to generate scripts one table at a time for the same script version. Using -za, you must generate scripts for all tables and publications at once.

☞ For more information, see "-za option" on page 219.

Example

Start the MobiLink synchronization server using the -za switch. For example, type:

```
dbmlsrv9 -c "dsn=YourDBDSN" -za
```

Run dbmlsync and set the SendColumnNames extended option to ON. For example, type:

```
dbmlsync -c dsn=dsn_remote -e "SendColumnNames=ON"
```

Scripts are generated for all tables specified in the publication. On synchronization, these automatically-generated scripts control the upload and download of data to and from your client and consolidated databases. The following table describes these scripts for the emp table.

Script name	Script contents
upload_insert	`INSERT INTO emp (emp_id, emp_name) VALUES (?,?)`
upload_update	`UPDATE emp SET emp_name=? WHERE emp_id=?`
upload_delete	`DELETE FROM emp WHERE emp_id=?`
download_cursor	`SELECT emp_id, emp_name FROM emp`

Generating example scripts

You can use the dbmlrv9 -ze option to generate example synchronization scripts. The example synchronization scripts are capable of performing a snapshot synchronization of your consolidated database with your remote database using the table and column names sent from the client, but they are not enabled. If the consolidated database has different table or column names, then activating these scripts causes an error during the synchronization.

To use this feature with Adaptive Server Anywhere clients, set the SendColumnNames extended option to ON to cause dbmlsync to send the column names with the upload header. To use this feature with UltraLite clients, set the send_column_names parameter to ul_true.

The -ze option generates the example scripts example_upload_insert, example_upload_update, example_upload_delete, and example_download_cursor.

☞ For more information, see "-ze option" on page 220.

Example

The following example generates scripts for an Adaptive Server Anywhere remote database.

At a command prompt, type:

```
dbmlsrv9 -c "dsn=YourDBDSN" -ze
```

At a command prompt, type:

```
dbmlsync -c dsn=dsn_remote -e "SendColumnNames=ON"
```

In the example above, example scripts are generated for all tables specified in the synchronization definition. The scripts exist for each table specified in the synchronization definition. The following table lists these scripts for the emp table.

Script name	Script
example_upload_insert	`INSERT INTO emp (emp_id,emp_name) VALUES (?,?)`
example_upload_update	`UPDATE emp SET emp_name=? WHERE emp_id=?`
example_upload_delete	`DELETE FROM emp WHERE emp_id=?`
example_download_cursor	`SELECT emp_id, emp_name FROM emp`

The example scripts select and upload all records from any table in the synchronization subscription that meet the conditions specified in the statement. So, for example, the upload_insert script for emp inserts all records from emp. The example scripts are generated for each table in the remote database specified in the synchronization subscription. The MobiLink synchronization server generates complete scripts needed for a snapshot synchronization. The scripts are added right after the synchronization description is processed. The synchronization is aborted after scripts are generated.

Example scripts for UltraLite

When you build an UltraLite application, the UltraLite generator automatically inserts an example_download_cursor script into the UltraLite reference database. The action of the example download script is to download all rows of a corresponding table that exists in the remote database. The script specifies the select list in the correct order.

The example script is inserted into the ml_scripts table, but they are not used unless you insert an entry in the ml_table_script table that associates them with the download_cursor event.

Minimally, the example scripts for download cursors provide the order of columns expected by the remote database.

Scripts and the synchronization process

Each script corresponds to a particular event in the synchronization process. You write a script only when some action must occur. All unnecessary events can be left undefined.

The two principal parts of the process are the processing of uploaded information and the preparation of rows for downloading.

The MobiLink synchronization server reads and prepares each script once, when it is first needed. The script is then executed whenever the event is invoked.

The sequence of events

☞ For information about the full sequence of MobiLink events, see "Overview of MobiLink events" on page 322.

☞ For the details of upload stream processing, see "Writing scripts to upload rows" on page 244.

☞ For the details of download stream processing, see "Writing scripts to download rows" on page 246.

Notes

- ♦ MobiLink technology allows multiple clients to synchronize concurrently. In this case, each client uses a separate connection to the consolidated database.
- ♦ The begin_connection and end_connection events are independent of any one synchronization as one connection can handle many synchronization requests. These scripts have no parameters. These are examples of connection-level scripts.
- ♦ Some events are invoked only once for each synchronization and have a single parameter. This parameter is the user name, which uniquely identifies the MobiLink client that is synchronizing. These are also examples of connection-level scripts.
- ♦ Some events are invoked once for each table being synchronized. Scripts associated with these events are called table-level scripts. They provide two parameters. The first is the user name supplied in the call to the synchronization function, and the second is the name of the table in the remote database being synchronized.

  While each table can have its own table scripts, you can also write table-level scripts that are shared by several tables.
- ♦ Some events, such as begin_synchronization, occur at both the connection level and the table level. You can supply both connection and table scripts for these events.

- The COMMIT statements illustrate how the synchronization process is broken up into distinct transactions.
- Errors are a separate event that can occur at any point within the synchronization process. Errors are handled using the following script.

  ```
  handle_error( error_code, error_message, user_name, table_
           name )
  ```

☞ For reference material, including detailed information about each script and its parameters, see "Synchronization Events" on page 319.

Script types

Synchronization scripts can apply to the entire connection or to specific tables.

- **connection scripts** These scripts perform actions that are connection-specific or synchronization-specific and that are independent of any one remote table. These scripts are used in conjunction with other scripts when implementing more complex synchronization schemes.
- **table scripts** These scripts perform actions specific to one synchronization and one particular remote table. These scripts are used in conjunction with other scripts when implementing more complex synchronization schemes.

Connection scripts

Connection-level scripts control high level events that are not associated with a particular table. Use these events to perform global tasks that are required during every synchronization.

Connection scripts control actions centered on connecting and disconnecting, as well as actions at synchronization-level events such as beginning and ending the upload or download process. Some connection scripts have related table scripts. These connection scripts are always invoked regardless of the tables being synchronized.

You only need to write a connection-level script when some action must occur at a particular event. You may need to create scripts for only a few events. The default action at any event is for the MobiLink synchronization server to carry out no actions. Some simple synchronization schemes need no connection scripts.

Table scripts

Table scripts allow actions at specific events relating to the synchronization of a specific table, such as the start or end of uploading rows, resolving conflicts, or selecting rows to download.

The synchronization scripts for a given table can refer to any table, or combination of tables, in the consolidated database. You can use this feature to fill a particular remote table with data stored in one or more consolidated tables, or to store data uploaded from a single remote table into multiple tables in the consolidated database.

Table names need not match

The names of tables in the remote databases need not match the names of the tables in the consolidated database. The MobiLink synchronization server determines which scripts are associated with a table by looking up the remote table name in the ml_table system table.

Script parameters

Most synchronization scripts receive parameters from the MobiLink synchronization server. You can use these parameters in your scripts by placing question marks in the script.

The following are some common parameters used in scripts.

- **last_download_timestamp** The last download timestamp is the value obtained from the consolidated database during the last successful synchronization immediately prior to the download phase. If the current MobiLink user has never synchronized, or has never synchronized successfully, this value is set to 1900-01-01.

  ☞ For more information, see "Timestamp-based synchronization" on page 48.

- **ml_username** The value of this parameter is the string that uniquely identifies a MobiLink client. Each client must identify itself by this name when initiating synchronization with a MobiLink synchronization server. This parameter is available within most connection-level scripts, all table-level scripts, and some cursor scripts.

  The user name can be used to partition tables among remote databases.

- **table** This parameter identifies a table in the remote database. The consolidated database may or may not contain a table with the same name. Only table scripts use this parameter.

To use parameters, place a single question mark in your SQL script for each parameter. Some parameters are optional. The MobiLink synchronization server replaces each question mark with the value of a parameter. It substitutes values in the order the parameters appear in the script definition.

☞ For reference material, including detailed information about each script and its parameters, see "Synchronization Events" on page 319.

Script versions

Scripts are organized into groups called **script versions**. By specifying a particular version, MobiLink clients can select which set of synchronization scripts will be used to process the upload stream and prepare the download stream.

☞ For information about how to add a script version to the consolidated database, see "Adding a script version" on page 240.

Application of script versions

Script versions allow you to organize your scripts into sets, which are run under different circumstances. This ability provides flexibility and is especially useful in the following circumstances.

- ♦ **customization** Using a different set of scripts to process information from different types of remote users. For example, you could write a different set of scripts for use when managers synchronize their databases than would be used for other people in the organization. Although you could achieve the same functionality with one set of scripts, these scripts would be more complicated.
- ♦ **upgrading applications** When you wish to upgrade a database application, new scripts may be needed because the new version of your application may handle data differently. New scripts are almost always necessary when the remote database changes. It is usually impossible to upgrade all users simultaneously. MobiLink clients can request that a new set of scripts be used during synchronization. Since both old and new scripts can coexist on the server, all users can synchronize no matter which version of your application they are using.
- ♦ **multiple applications** A single MobiLink synchronization server may need to synchronize two entirely different applications. For example, some employees may use a sales application, whereas others require an application designed for inventory control. When two applications require different sets of data, you can create two versions of the synchronization scripts, one version for each application.
- ♦ **set properties for the script version** You can set properties for your script version that can be referenced from classes in .NET or Java synchronization logic. For more information, see "ml_add_property" on page 486.

Assigning version names

A script version name is a string. You specify this name when you add a script to the consolidated database. For example, if you add your scripts with the ml_add_connection_script and the ml_add_table_script stored procedures, the script version name is the first parameter. Alternatively, if you add your scripts using Sybase Central, you are prompted for the version name.

You cannot use the following names for script versions: **ml_sis_1** or **ml_qa_1**. These names are used internally by MobiLink. In addition, it is recommended that your script versions do not start with **ml_**.

The default script version

Whenever a remote site fails to supply a script version, the MobiLink synchronization server uses the first version defined in the ml_script_version table. If no script version has been defined, the synchronization fails.

Adding a script version

All scripts are associated with a script version. You must add a version name to your consolidated database before you can add any connection scripts.

☞ For more information, see "Script versions" on page 239.

❖ To add a script version to a database (Sybase Central)

1. From Sybase Central, right-click MobiLink Synchronization and connect to the consolidated database.
2. Open the Versions folder.
3. Double-click Add Version and follow the instructions in the wizard.

❖ To remove a script version from a database (Sybase Central)

1. From Sybase Central, right-click MobiLink Synchronization and connect to the consolidated database.
2. Open the Versions folder.
3. Right-click the version name and select Delete.
4. The Confirm Delete dialog appears. Click Yes.

❖ To add a script version to a database (stored procedures)

1. You can add a script version in the same operation as adding a connection script or table script.

☞ For more information, see "Stored procedures to add or delete scripts" on page 480.

Adding and deleting scripts in your consolidated database

When you have created scripts, you must add them to MobiLink system tables in the consolidated database. To do this, you can use stored procedures or Sybase Central wizards.

☞ For information about the MobiLink system tables, see "MobiLink System Tables" on page 501.

Note: SQL scripts are stored in the consolidated database. In the case of Java and .NET scripts, the location of the script is stored in the consolidated database. However, the method for adding/deleting a script or script location is similar.

Adding or deleting scripts

You can add synchronization scripts using Sybase Central wizards. The procedure is different for connection scripts and table scripts. Table scripts correspond to tables in the remote database, so before you can add a table script, you must add the name of the remote database table to the consolidated database.

If you are using Sybase Central, you must add a synchronization version to the database before you can add individual scripts. For more information, see "Adding a script version" on page 240.

❖ To add or delete a connection script (Sybase Central)

1. From Sybase Central, right-click MobiLink Synchronization and connect to the consolidated database.

2. Open Connection Scripts.

3. To add a connection script, double-click Add Connection Script and follow the instructions in the wizard.

   or

   To delete a connection script, right-click the script name and select Delete. The Confirm Delete dialog appears. Click Yes.

❖ To add or delete a remote table in the list of synchronized tables (Sybase Central)

1. From Sybase Central, right-click MobiLink Synchronization and connect to the consolidated database.

2. Open Synchronized Tables.

3. To add a remote table to the list of synchronized tables, double-click Add Synchronized Table. Enter the name of a table at the remote database for which you are going to write synchronization scripts. The wizard provides a shortcut if the consolidated database has a table with a matching name.

   or

   To delete a remote table from the list of synchronized tables, right-click the table name and select Delete. The Confirm Delete dialog appears. Click Yes.

❖ To add or delete a table script in a database (Sybase Central)

1. From Sybase Central, right-click MobiLink Synchronization and connect to the consolidated database.

2. Open Synchronized Tables.

3. Select the table for which you wish to add a script.

4. To add a table script, double-click Add Table Script and follow the instructions in the wizard.

   or

   To delete a table script, right-click the script name and select Delete. The Confirm Delete dialog appears. Click Yes.

❖ To add or delete all types of scripts (stored procedures)

1. You can add scripts to a consolidated database or delete scripts from a consolidated database using stored procedures that are installed along with the MobiLink system tables when you create your consolidated database.

   ☞ For a description of the stored procedures that you can use to add or delete scripts, see "Stored procedures to add or delete scripts" on page 480.

Direct inserts of scripts

In most cases, it is recommended that you use stored procedures or Sybase Central to insert scripts into the system tables. However, in some rare cases you may need to use an INSERT statement to directly insert the scripts. For example, older versions of some DBMSs may have length limitations that make it difficult to use stored procedures.

☞ For a complete description of the MobiLink system tables, see "MobiLink System Tables" on page 501.

The format of the INSERT statements that are required to directly insert scripts can be found in the source code for the ml_add_connection_script and ml_add_table_script stored procedures. The source code for these stored procedures is located in the MobiLink setup scripts. There is a different setup script for each supported RDBMS. The setup scripts are:

Consolidated database	Setup file
Adaptive Server Anywhere	*scripts\syncasa.sql*
Oracle	*MobiLink\setup\syncora.sql*
IBM DB2 UDB	*MobiLink\setup\syncdb2long.sql*
Microsoft SQL Server	*MobiLink\setup\syncmss.sql*
Adaptive Server Enterprise version 12.5 and later	*MobiLink\setup\syncase125.sql*
Adaptive Server Enterprise prior to version 12.5	*MobiLink\setup\syncase.sql*

Writing scripts to upload rows

To upload information contained in your remote database to your consolidated database, you define upload scripts. You write separate scripts to handle rows that are updated, inserted, or deleted at the remote database. A simple implementation would carry out corresponding actions (update, insert, delete) at the consolidated database.

☞ The MobiLink synchronization server uploads data in a single transaction. For a description of the upload process, see "Events during upload" on page 328.

☞ For techniques for uploading rows in .NET synchronization logic, see "Uploading rows" on page 295.

Notes

- The begin_upload and end_upload scripts for each remote table hold logic that is independent of the individual rows being updated.
- The upload stream consists of single row inserts, updates, and deletes. These actions are typically performed using upload_insert, upload_update and upload_delete scripts.
- To prepare the upload for Adaptive Server Anywhere clients, the dbmlsync utility requires access to all transaction logs written since the last successful synchronization. For more information, see "Transaction log files" [*MobiLink Clients*, page 79].

Writing upload_insert scripts

The MobiLink synchronization server uses this event during processing of the upload stream to handle rows inserted into the remote database. The following INSERT statement shows how you use the upload_insert statement.

```
INSERT INTO emp (emp_id,emp_name)
VALUES (?,?)
```

☞ For more information, see "upload_insert table event" on page 463.

Writing upload_update scripts

The MobiLink synchronization server uses this event during processing of the upload stream to handle rows updated at the remote database. The following UPDATE statement illustrates use of the upload_update statement.

```
UPDATE emp
SET emp_name=?
WHERE emp_id=?
```

For more information, see "upload_update table event" on page 475.

Writing upload_delete scripts

The MobiLink synchronization server uses this event during processing of the upload stream to handle rows deleted from the remote database. The following statement shows how to use the upload_delete statement.

```
DELETE FROM emp
WHERE emp_id=?
```

For more information, see "upload_delete table event" on page 459.

Writing upload_fetch scripts

The upload_fetch script is a SELECT statement that defines a cursor in the consolidated database table. This cursor is used to compare the old values of updated rows, as received from the remote database, against the value in the consolidated database. In this way, the upload_fetch script identifies conflicts when updates are being processed.

Given a synchronized table defined as:

```
CREATE TABLE uf_example (
   pk1 integer NOT NULL,
   pk2    integer NOT NULL,
   val    varchar(200),
   PRIMARY KEY( pk1, pk2 ))
```

Then one possible upload_fetch script for this table is:

```
SELECT pk1, pk2, val
FROM uf_example
WHERE pk1 = ? and pk2 = ?
```

☞ For more information, see "upload_fetch table event" on page 461.

The MobiLink synchronization server requires the WHERE clause of the query in the upload_fetch script to identify exactly one row in the consolidated database to be checked for conflicts.

Writing scripts to download rows

There are two scripts that can be used for processing each table during the download transaction. These are the download_cursor script, which carries out inserts and updates, and the download_delete_cursor script, which carries out deletes.

These scripts are either SELECT statements or calls to procedures that return result sets. The MobiLink synchronization server downloads the result set of the script to the remote database. The MobiLink client automatically inserts or updates rows based on the download_cursor script result set, and deletes rows based on the download_delete_cursor event.

☞ For more information about using stored procedures, see "Downloading a result set from a stored procedure call" on page 78.

The MobiLink synchronization server downloads data in a single transaction. For a description of the download process, see "Events during download" on page 332.

Notes

♦ Like the upload stream, the download stream starts and ends with connection events. Other events are table-level events.

♦ If you change the SendDownloadAck setting to ON, if no confirmation of the download is received from the client, the entire download transaction is rolled back in the consolidated database. (By default, SendDownloadAck is set to OFF.)

  ☞ For more information, see "SendDownloadACK (sa) extended option" [*MobiLink Clients*, page 131] or "Send Download Acknowledgement synchronization parameter" [*MobiLink Clients*, page 331].

♦ The begin_download and end_download scripts for each remote table hold logic that is independent of the individual rows being updated.

♦ The download stream does not distinguish between inserts and updates. The script associated with the download_cursor event is a SELECT statement that defines the rows to be downloaded. The client detects whether the row exists or not and carries out the appropriate insert or update operation.

♦ At the end of the download processing, the client automatically deletes rows if necessary to avoid referential integrity violations.

  ☞ For more information, see "Referential integrity and synchronization" on page 22.

Writing download_cursor scripts

You write download_cursor scripts to download information from the consolidated database to your remote database. You must write one of these scripts for each table in the remote database for which you want to download changes. You can use other scripts to customize the download process, but no others are necessary.

- Each download_cursor script must contain a SELECT statement or a call to a procedure that contains a SELECT statement. The MobiLink synchronization server uses this statement to define a cursor in the consolidated database.
- The script must select all columns that correspond to the columns in the corresponding table in the remote database. The columns in the consolidated database can have different names than the corresponding columns in the remote database, but they must be of compatible types.
- The columns must be selected in the order that the corresponding columns are defined in the remote database. This order is identical to the order of the columns in the reference database.

Example

The following script could serve as a download_cursor script for a remote table that holds employee information. The MobiLink synchronization server would use this SQL statement to define the download cursor. This script downloads information about all the employees.

```
SELECT emp_id, emp_fname, emp_lname
FROM employee
```

The MobiLink synchronization server passes specific parameters to some scripts. To use these parameters, you include a question mark in your SQL statement. The MobiLink synchronization server substitutes the value of the parameter before executing the statement against the consolidated database. The following script shows how you can use these parameters:

```
call ml_add_table_script( ’Lab’, ’ULOrder’, ’download_cursor’,
’SELECT o.order_id, o.cust_id, o.prod_id, o.emp_id, o.disc,
  o.quant, o.notes, o.status
FROM ULOrder o
WHERE o.last_modified >= ?
AND o.emp_name = ?’ )
```

In this example, the MobiLink synchronization server replaces the question mark with the value of the parameter to the download_cursor script.

Notes

- Row values can be selected from a single table or from a join of multiple tables.

♦ The script itself need not include the name of the remote table. The remote table need not have the same name as the table in the consolidated database. The name of the remote table is identified by an entry in the *ml_table* table. In Sybase Central, the remote tables are listed together with their scripts.

♦ The rows in the remote table must contain the values of *emp_id*, *emp_fname*, and *emp_lname*. The remote columns must be in that order, although they can have different names. The columns in the remote database are in the same order as those in the reference database.

UltraLite tip
The example scripts list the columns in the order that they are defined in the reference database. Inspect the example_download_cursor and example_upload_cursor scripts to see the column order.

♦ All cursor scripts must select the columns in the same order as the columns are defined in the remote database. Where column names or table structure is different in the consolidated database, columns should be selected in the correct order for the remote database, or equivalently, the reference database. Columns are assigned to columns in the remote database based on their order in the SELECT statement.

♦ When you build an UltraLite application, the UltraLite generator creates a sample download script for each table in your UltraLite application. It inserts these sample scripts into your reference database. The example scripts assume that the consolidated database contains the same tables as your application. You must modify the sample scripts if your consolidated database differs in design, but these scripts provide a starting point.

See also

♦ "download_cursor table event" on page 371
♦ "Partitioning rows among remote databases" on page 52
♦ "Writing download_delete_cursor scripts" on page 248

Writing download_delete_cursor scripts

You write download_delete_cursor scripts to delete rows from your remote database. You must write one of these scripts for each table in the remote database from which you want to delete rows during synchronization.

You cannot just delete rows from the consolidated database and have them disappear from remote databases. You need to keep track of the primary keys for deleted rows, so that you can select those primary keys with your download_delete_cursor. There are two common techniques for achieving this:

- **Logical deletes** Do not physically delete the row in the consolidated database. Instead, have a status column that keeps track of whether rows are valid. This simplifies the download_delete_cursor. However, the download_cursor and other applications may need to be modified to recognize and use the status column. If you have a last modified column that holds the time of deletion, and if you also keep track of the last download time for each remote, then you can physically delete the row once all remote download times are newer than the time of deletion.
- **Shadow table** For each table for which you want to track deletes, create a shadow table with two columns, one holding the primary key for the table, and the other holding a timestamp. Create a trigger that inserts the primary key and timestamp into the shadow table whenever a row is deleted. Your download_delete_cursor can then select from this shadow table. As with logical deletes, you can delete the row from the shadow table once all remote databases have downloaded it.

The MobiLink synchronization server deletes rows in the remote database by selecting primary key values from the consolidated database and passing those values to the remote database. If the values match those of a primary key in the remote database, then that row is deleted.

- Each download_delete_cursor script must contain a SELECT statement or a call to a stored procedure that returns a result set. The MobiLink synchronization server uses this statement to define a cursor in the consolidated database.
- This statement must select all the columns that correspond to the primary key columns in the table in the remote database. The columns in the consolidated database can have different names than the corresponding columns in the remote database, but they must be of compatible types.
- The values must be selected in the same order as the corresponding columns are defined in the remote database. That order is the order of the columns in the CREATE TABLE statement used to make the table, not the order they appear in the statement that defines the primary key.

While each download_delete_cursor script must select all the column values present in the primary key of the corresponding remote table, it may also select all the other columns. This feature is present only for compatibility with older clients. Selecting the additional columns is less efficient, as the database engine must retrieve more data. Unless the client is of an old design, the MobiLink synchronization server discards the extra values immediately. The extra values are downloaded only to older clients.

Deleting all the rows in a table

When MobiLink detects a download_delete_cursor with a row that contains all NULLs, it deletes all the data in the remote table. The number of NULLs in the download_delete_cursor can be the number of primary key columns or the total number of columns in the table.

For example, the following download_delete_cursor SQL script deletes every row in a table in which there are two primary key columns. This example works for Adaptive Server Anywhere, Adaptive Server Enterprise, and Microsoft SQL Server databases.

```
SELECT NULL, NULL
```

In IBM DB2 and Oracle consolidated databases, you must specify a dummy table to select NULL. For IBM DB2 7.1, you can use the following syntax:

```
SELECT NULL FROM SYSIBM.SYSDUMMY1
```

For Oracle consolidated databases, you can use the following syntax:

```
SELECT NULL FROM DUAL
```

Examples

The following example is a download_delete_cursor script for a remote table that holds employee information. The MobiLink synchronization server uses this SQL statement to define the delete cursor. This script deletes information about all employees who are both in the consolidated and remote databases at the time the script is executed.

```
SELECT emp_id
FROM employee
```

The download_delete_cursor accepts the parameters last_download and ml_username. The following script shows how you can use each parameter to narrow your selection.

```
SELECT order_id
FROM ULOrder
WHERE last_modified >= ?
   AND status = 'Approved'
   AND user_name = ?
```

Note that these examples could prove inefficient in an organization with many employees. You can make the delete process more efficient by selecting only rows that could be present in the remote database. For example, you could limit the number of rows by selecting only those people who have recently been assigned a new manager. Another strategy is to allow the client application to delete the rows itself. This method is possible only when a rule identifies the unneeded rows. For example, rows might contain a timestamp that indicates an expiry date. Before you delete the rows, use the STOP SYNCHRONIZATION DELETE statement to stop these deletes being uploaded during the next synchronization. Be sure to execute START SYNCHRONIZATION DELETE immediately afterwards if you want other deletes to be synchronized in the normal fashion.

See also

☞ You can use the referential integrity checking built into all MobiLink clients to delete rows in a particularly efficient manner. For details, see "Referential integrity and synchronization" on page 22.

☞ For more information about using download_delete_cursor scripts, see

- "download_delete_cursor table event" on page 375
- "Temporarily stopping synchronization of deletes" [*MobiLink Clients*, page 87]
- "STOP SYNCHRONIZATION DELETE statement [MobiLink]" [*ASA SQL Reference*, page 637]
- "Handling deletes" on page 73
- "download_cursor table event" on page 371
- "Partitioning rows among remote databases" on page 52
- "Snapshot synchronization" on page 50

Writing scripts to handle errors

An error in a synchronization script occurs when an operation in the script fails while the MobiLink synchronization server is executing it. The DBMS returns a SQLCODE to the MobiLink synchronization server indicating the nature of the error. Each consolidated database DBMS has its own set of SQLCODE values.

When an error occurs, the MobiLink synchronization server invokes the handle_error event. You should provide a connection script associated with this event to handle errors. The MobiLink synchronization server passes several parameters to this script to provide information about the nature and context of the error, and requires an output value to tell it how to respond to the error.

Error handling actions

Some actions you may wish to take in an error-handling script are:

- Log the error in a separate table.
- Instruct the MobiLink synchronization server whether to ignore the error and continue, or rollback the synchronization, or rollback the synchronization and shut down the MobiLink synchronization server.
- Send an e-mail message.

☞ For more information, see "handle_error connection event" on page 415.

Reporting errors

Since errors can disrupt the natural progression of the synchronization process, it can be difficult to create a log of errors and their resolutions. The report_error script provides a means of accomplishing this task. The MobiLink synchronization server executes this script whenever an error occurs. If the handle_error script is defined, it is executed immediately prior to the reporting script.

The parameters to the report_error script are identical to those of the handle_error script, except that the report_error script can not modify the action code. Since the value of the action code is that returned by the handle_error script, this script can be used to debug error-handling problems.

This script typically consists of an insert statement, which records the values, perhaps with other data, such as the time or date, in a table for later reference. To ensure that this data is not lost, the MobiLink synchronization server always runs this script in a separate transaction and automatically commits the changes as soon as this script completes.

☞ For more information, see "report_error connection event" on page 438.

Example

The following report_error script, which consists of a single insert statement, adds the script parameters into a table, along with the current date and time. The script does not commit this change because the MobiLink synchronization server always does so automatically.

```
INSERT INTO errors
VALUES( CURRENT DATE, ?, ? ,?, ?, ? );
```

Handling multiple errors on a single SQL statement

ODBC allows multiple errors per SQL statement, and some DBMSs make use of this feature. Microsoft SQL Server, for example, can have two errors for a single statement. The first is the actual error, and the second is usually an informational message telling you why the current statement has been terminated.

When a single SQL statement causes multiple errors, the handle_error script is invoked once per error. The MobiLink synchronization server uses the most severe action code (that is, the numerically greatest) to determine the action to take. The same applies to the handle_error script.

If the handle_error script itself causes a SQL error, then the default action code (3000) is assumed.

Testing script syntax

As you develop your synchronization scripts, you can use Sybase Central to test for syntax errors in your scripts.

Testing the scripts is done without any remote site in place. No data is added to the database or downloaded from the database during testing. The validity of the synchronized data itself is not tested.

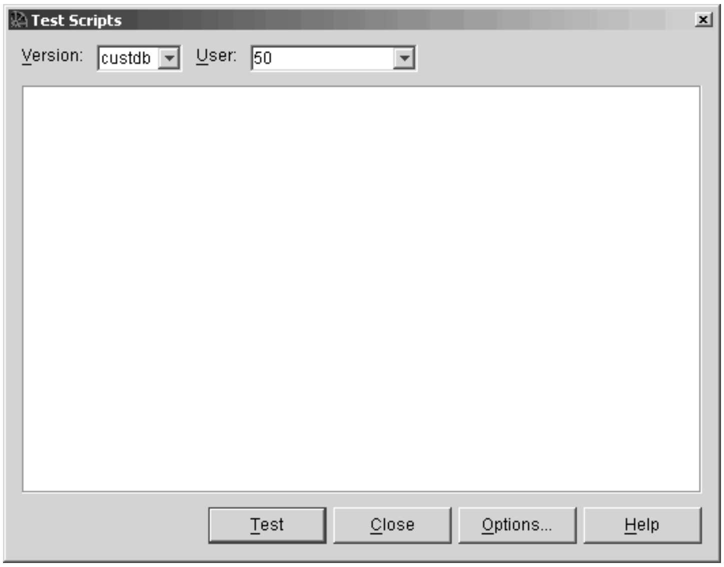
❖ To test your synchronization scripts

1. Start Sybase Central and connect to a database from MobiLink Synchronization.

2. In the left pane, click the database name.

3. Right-click the Synchronized Tables folder or the Connection scripts folder and select Test Scripts from the popup menu.

   The Test Scripts window appears:



4. Click Test. If prompted, enter parameters. The results of the test are displayed in the window.

   The test results include a listing of which scripts are executed, in which order. They also include a listing of any syntax errors or data type errors found during the test.

CHAPTER 13

Writing Synchronization Scripts in Java

About this chapter

You control the actions of the MobiLink synchronization server by writing synchronization scripts. You can implement these scripts in SQL, .NET or Java. This chapter describes how to implement synchronization scripts in Java.

☞ For a tutorial using Java synchronization scripts, see "Tutorial: Java Synchronization Logic With Adaptive Server Anywhere " [*MobiLink Tutorials*, page 51].

☞ For a description and comparison of SQL, Java, and .NET, see "Options for writing synchronization logic" on page 25.

☞ For information about writing scripts, see "Writing Synchronization Scripts" on page 227.

☞ For information about writing scripts in .NET, see "Writing Synchronization Scripts in .NET" on page 281.

Contents

Introduction

MobiLink synchronization scripts can be written in Java. Java synchronization logic can function just as SQL logic functions: the MobiLink synchronization server can make calls to Java methods on the occurrence of MobiLink events just as it accesses SQL scripts on the occurrence of MobiLink events. A SQL string may be returned as a Java method to MobiLink.

This section tells you how to set up, develop, and run Java synchronization logic. It includes a sample application and the MobiLink Java API.

☞ For a tutorial using Java synchronization scripts, see "Tutorial: Java Synchronization Logic With Adaptive Server Anywhere " [*MobiLink Tutorials,* page 51].

Setting up Java synchronization logic

When you install SQL Anywhere Studio, the installer automatically sets the location of the MobiLink Java API classes. When you start the MobiLink synchronization server, it automatically includes these classes in your classpath. The MobiLink Java API classes are installed to *java\mlscript.jar* in your SQL Anywhere Studio installation directory. MobiLink uses the ASANYSH9 environment variable to determine the shared component path.

☞ For more information, see "ASANYSH9 environment variable" [*ASA Database Administration Guide*, page 277].

❖ To implement synchronization scripts in Java

1. Create your own class or classes. Write a method for each required synchronization script. These methods must be public. The class must be public in the package.

   ☞ For more information about methods, see "Methods" on page 261.

   Each class with non-static methods should have a public constructor. The MobiLink synchronization server automatically instantiates each class the first time a method in that class is called.

   ☞ For more information about constructors, see "Constructors" on page 260.

2. When compiling the class, you must include the JAR file *java\mlscript.jar*.

   For example,

   ```
   javac MyClass.java -classpath %ASANY9%\java\mlscript.jar
   ```

3. In your consolidated database, specify the name of the package, class, and method to call for each synchronization script. One class is permitted per script version.

   The easiest way to add this information to the MobiLink system tables is to use the ml_add_java_connection_script stored procedure or the ml_add_java_table_script stored procedure.

   For example, the following SQL statement, when run in an Adaptive Server Anywhere database, specifies that for the script version ver1, myPackage.myClass.myMethod should be run whenever the authenticate_user connection-level event occurs. The method that is specified must be the fully qualified name of a public Java method, and the name is case sensitive.

   ```
   call ml_add_java_connection_script( 'ver1',
   'authenicate_user', 'myPackage.myClass.myMethod' )
   ```

☞ For more information about adding scripts, see:

- ♦ "Stored procedures to add or delete scripts" on page 480
- ♦ "ml_add_java_connection_script" on page 483
- ♦ "ml_add_java_table_script" on page 484

4. Instruct the MobiLink server to load classes. A vital part of setting up Java synchronization logic is to tell the virtual machine where to look for Java classes. There are two ways to do this:

   - ♦ Use the dbmlsrv9 -sl java -cp option to specify a set of directories or jar files in which to search for classes. For example, at the command line, enter:

     ```
     dbmlsrv9 -c "dsn=consolidated1" -sl java (-cp
              %classpath%;c:\local\Java\myclasses.jar)
     ```

     The MobiLink synchronization server automatically appends the location of the MobiLink Java API classes (java\mlscript.jar) to the set of directories or jar files. The -sl java option also forces the Java VM to load on server startup.

     ☞ For more information about the available Java options, see "-sl java option" on page 209.

   - ♦ Explicitly set the classpath. To set the classpath for user-defined classes, use a statement such as the following:

     ```
     SET classpath=%classpath%;c:\local\Java\myclasses.jar
     ```

     If your system classpath includes your Java synchronization logic classes, you do not need to make changes to your MobiLink synchronization server command line.

     You can use the -sl java option to force the Java virtual machine to load at server startup. Otherwise, the Java virtual machine is started when the first Java method is executed.

     ☞ For more information about the available Java options, see "-sl java option" on page 209.

5. On UNIX, if you want to load a specific JRE, you should set the LD_LIBRARY_PATH (LIBPATH on AIX, SHLIB_PATH on HP-UX) to include the directory containing the JRE. The directory must be listed before any of the Adaptive Server Anywhere install directories.

Writing Java synchronization logic

Writing Java synchronization logic is no different in complexity from writing any other Java code. What is required from you is knowledge of MobiLink events, some knowledge of Java, and knowledge of the MobiLink Java API. The following sections help you write useful synchronization logic.

Java synchronization logic can be used to maintain state information, and implement logic around the upload and download events. For example, a begin_synchronization script written in Java could store the MobiLink user name in a variable. Scripts called later in the synchronization process can access this variable. Also, you can use Java to access rows in the consolidated database, before or after they are committed.

Using Java reduces dependence on the consolidated database. Behavior is affected less by upgrading the consolidated database to a new version or switching to a different database-management system.

☞ For a complete description of the API, see “MobiLink Java API Reference” on page 273.

Class instances

The MobiLink synchronization server instantiates your classes at the connection level. When an event is reached for which you have written a non-static Java method, the MobiLink synchronization server automatically constructs the class, if it has not already done so on the present connection.

☞ For more information, see “Constructors” on page 287.

All methods directly associated with a connection-level or table-level event for one script version *must belong to the same class.*

For each database connection, once a class has been instantiated, the class persists until that connection is closed. Thus, the same instance may well be used for multiple consecutive synchronization sessions. Information present in public or private variables will thus persist across synchronizations that occur on the same connection unless explicitly cleared.

You can also use static classes or variables. In this case, the values are available across all connections.

The MobiLink synchronization server automatically deletes your class instances only when the connection to the consolidated database is closed.

Transactions

The normal rules regarding transactions apply to Java methods. The start and duration of database transactions is critical to the synchronization process. Transactions must be started and ended only by the MobiLink synchronization server. Explicitly committing or rolling back transactions on the synchronization connection within a Java method violates the integrity of the synchronization process and can cause errors.

These rules apply only to the database connections created by the MobiLink synchronization server and, in particular, to SQL statements returned by methods. If your classes create other database connections, you are free to manage them as you please.

SQL-Java data types

The following table shows SQL data types and the corresponding Java data types.

SQL data type	Corresponding Java data type
VARCHAR	java.lang.String
CHAR	java.lang.String
INTEGER	Int or Integer
BINARY	byte[ ]
TIMESTAMP	java.sql.Timestamp
INOUT INTEGER	ianywhere.ml.script.InOutInteger
INOUT VARCHAR	ianywhere.ml.script.InOutString
INOUT CHAR	ianywhere.ml.script.InOutString
INOUT BYTEARRAY	ianywhere.ml.script.InOutByteArray

The MobiLink synchronization server automatically adds this package to your classpath if it is not already present.

However, when you compile your class you need to add the path of java\mlscript.jar, from your SQL Anywhere Studio installation directory.

Constructors

The constructor of your class may have one of two possible signatures.

```
public MyScriptClass (
ianywhere.ml.script.DBConnectionContext sc )
```

or

```
public MyScriptClass ( )
```

The synchronization context passed to you is for the connection through which the MobiLink synchronization server is synchronizing the current user.

The getConnection method of the DBConnectionContext class returns the same database connection that MobiLink is using to synchronize the present user. You can execute statements on this connection, but you must not commit or roll back the transaction. The MobiLink synchronization server manages the transactions.

The MobiLink synchronization server prefers to use constructors with the first signature. It only uses the non-argument constructor if a constructor with the first signature is not present.

☞ For more information, see "DBConnectionContext interface" on page 273.

Methods

In general, you implement one method for each synchronization event. These methods must be public. If they are private, the MobiLink synchronization server cannot use them and will fail to recognize that they exist.

The names of the methods are not important, as long as the names match the names specified in the ml_script table in the consolidated database. In the examples included in the documentation, however, the method names are the same as those of the MobiLink events because this naming convention makes the Java code easier to read.

The signature of your method should match the signature of the script for that event, except that you can truncate the parameter list if you do not need the values of parameters at the end of the list. Indeed, you should accept only the parameters you need, because overhead is associated with passing the parameters.

You cannot, however, overload the methods. Only one method prototype per class may appear in the ml_script system table.

Return values

Methods called for a MobiLink upload or download must return a valid SQL language statement. The return type of these methods must be java.lang.String. No other return types are allowed.

The return type of all other scripts must either be java.lang.String or void. No other types are allowed. If the return type is a string and not null, the MobiLink synchronization server assumes that the string contains a valid

SQL statement and executes this statement in the consolidated database as it would an ordinary SQL-language synchronization script. If a method ordinarily returns a string but does not wish to execute a SQL statement against the database upon its return, it can return null.

Debugging Java classes

MobiLink provides various information and facilities that you may find helpful when debugging your Java code. This section describes where you can find this information and exploit these capabilities.

Information in the MobiLink synchronization server's log file

The MobiLink synchronization server writes various related information to its output log file. The synchronization server log file contains the following information:

♦ The Java Runtime Environment. You can use the -jrepath option to request a particular JRE when you start the MobiLink synchronization server. The default is the path installed with Adaptive Server Anywhere 9.

♦ The path of the standard Java classes loaded. If you did not specify these explicitly, the MobiLink synchronization server automatically adds them to your classpath before invoking the Java virtual machine.

♦ The fully specified names of the specific methods invoked. You can use this information to verify that the MobiLink synchronization server is invoking the correct methods.

♦ Any output written in a Java method to java.lang.System.out or java.lang.System.err is redirected to the MobiLink synchronization server log file.

♦ The dbmlsrv9 command line option -verbose can be used.

  ☞ For more information, see "-sl java option" on page 209.

Using a Java debugger

You can debug your Java classes using a standard Java debugger. Specify the necessary parameters using the -sl java option on the dbmlsrv9 command line.

☞ For more information, see "-sl java option" on page 209.

Specifying a debugger causes the Java virtual machine to pause and wait for a connection from a Java debugger.

Printing information from Java

Alternatively, you may choose to add statements to your Java methods that print information to the MobiLink output log, using java.lang.System.err or java.lang.System.out. Doing so can help you track the progress and behavior of your classes.

Performance tip
Printing information in this manner is a useful monitoring tool, but is not recommended in a production scenario.

The same technique can be exploited to log arbitrary synchronization information or collect statistical information on how your scripts are used.

Writing your own test driver

You may wish to write your own driver to exercise your Java classes. This approach can be helpful because it isolates the actions of your Java methods from the rest of the MobiLink system.

Handling MobiLink server errors in Java

When scanning the log is not sufficient, you can monitor your applications programmatically. For example, you can send messages of a certain type in an email.

You can write methods that are passed a class representing every error or warning message that is printed to the log. This may help you monitor and audit a MobiLink synchronization server.

The following code installs a LogListener for all warning messages, and writes the information to a file.

```
class TestLogListener implements LogListener {
        FileOutputStream    _out_file;

        public TestLogListener(   FileOutputStream    out_file )
        {
            _out_file       = out_file;
        }
        public void messageLogged(  ServerContext   sc,
                                    LogMessage      msg )
        {
            String  type;
            String  user;
            try {
                 if(msg.getType() == LogMessage.ERROR) {
                    type = "ERROR";
                } else if(msg.getType() == LogMessage.WARNING)
         {
                    type = "WARNING";
                } else {
                    type = "UNKNOWN!!!";
                }
                user = msg.getUser();
                if( user == null ) {
                    user = "NULL";
                }
                _out_file.write(
                        ("Caught msg type=" + type +
                                " user=" + user +
                                " text=" +msg.getText() +
                                "\n").getBytes() );
                _out_file.flush();
            } catch( Exception e ) {
              // print some error output to the MobiLink log
                e.printStackTrace();
            }
        }
    }
// This line of code will register TestLogListener to receive
// warning messages.  Call this code from anywhere that has
// access to the ServerContext such as a class constructor or
// synchronization script.  ServerContext serv_context;
serv_context.addWarningListener(
      new MyLogListener( ll_out_file ));
```

See also

addErrorListener, removeErrorListener, addWarningListener, removeWarningListener in "ServerContext interface" on page 275

"LogListener interface" on page 275

"LogMessage class" on page 275

User-defined start classes

You can define start classes that are loaded automatically when the server is started. The purpose of this feature is to allow you to write Java code that executes at the time the MobiLink server starts the JVM—before the first synchronization. This means you can create connections or cache data before a user synchronization request.

You do this with the DMLStartClasses option of the dbmlsrv9 -sl java option. For example, the following is part of a dbmlsrv9 command line. It causes mycl1 and mycl2 to be loaded as start classes.

```
-sl java(-DMLStartClasses=com.test.mycl1,com.test.mycl2)
```

Classes are loaded in the order in which they are listed. If the same class is listed more than once, more than one instance is created.

All start classes must be public and must have a public constructor that either accepts no arguments or accepts one argument of type ianywhere.ml.script.ServerContext.

The names of loaded start classes are output to the MobiLink log with the message “Loaded JAVA start class: *classname*”.

☞ For more information about Java virtual machine options, see “-sl java option” on page 209.

☞ To see the start classes that are constructed at server start time, see “getStartClassInstances” on page 276.

Example

Following is a template start class. It starts a daemon thread that processes events and creates a database connection. (Not all start classes will need to create a thread but if a thread is spawned it should be a daemon thread.)

```
import ianywhere.ml.script.*;
import java.sql.*;

public class StartTemplate extends
   Thread implements ShutdownListener {
//=================================================
  ServerContext   _sc;
  Connection      _conn;
  boolean         _exit_loop;

  public StartTemplate( ServerContext sc )
  //===============================================
                      throws SQLException
  {
    // perform setup first so that an exception will
    // cause MobiLink startup to fail
 _sc       = sc;
    // create a connection for use later
    _conn      = _sc.makeConnection();
    _exit_loop = false;
    setDaemon( true );
    start();
  }
  public void run()
  //===============
  {
    _sc.addShutdownListener( this );
    // we can’t throw any exceptions through run()
    try {
       handlerLoop();
       _conn.close();
       _conn = null;
    } catch( Exception e ) {
       // print some error output to the MobiLink log
       e.printStackTrace();
       // we will die so we don’t need to be notified
       // of shutdown
       _sc.removeShutdownListener( this );
       // ask server to shutdown so that this fatal
       // error will be fixed
       _sc.shutdown();
    }
    // shortly after return this thread will no longer
    // exist
    return;
  }

 public void shutdownPerformed( ServerContext sc )
  //================================================
  // stop our event handler loop
  {
    try {
       // wait max 10 seconds for thread to die
  join( 10*1000);
    } catch( Exception e ) {
       // print some error output to the MobiLink log
       e.printStackTrace();
   }
  }
  private void handlerLoop()
  //===============throws InterruptedException
  {
      while( !_exit_loop ) {
        // Handle events in this loop. Sleep not
        // needed, we should block on event queue.
        sleep( 1*1000 );
    }
  }
}
```

Java synchronization example

Java synchronization logic works with MobiLink and common Java classes to provide you with flexibility in deploying applications using MobiLink synchronization server. The following section introduces you to this extended range of functionality using a simple example.

Note:
This section provides a simple example to illustrate basic Java synchronization logic. Typically, reasons for using a custom user authentication mechanism include integration with existing DBMS user authentication schemes, or supplying custom features, such as minimum password length or password expiry. MobiLink also has a built-in mechanism.

☞ For more information about MobiLink authentication, see "Choosing a user authentication mechanism" [*MobiLink Clients,* page 13].

Introduction

This section describes a working example of Java synchronization logic. Before you try to use this class or write your own class, use the following checklist to ensure you have all the pieces in place before compiling the class.

- ♦ Plan your desired functionality using, for example, pseudocode.
- ♦ Create a map of database tables and columns.
- ♦ Set up the consolidated database by ensuring you have specified in the MobiLink system tables the language type and location of the Java synchronization methods.

  ☞ For more information see "Setting up Java synchronization logic" on page 257.
- ♦ Create a list of associated Java classes that are called during the running of your Java class.
- ♦ Have a location for your Java classes that is in the classpath for MobiLink synchronization server.

Plan

The Java synchronization logic for this example points to the associated Java files and classes that contain functionality needed for the example to work. It will show you how to create a class CustEmpScripts. It shows you how to set up a synchronization context for the connection. Finally, the example provides Java methods to

- ♦ Authenticate a MobiLink user

- Perform download and upload operations using cursors for each database table.

Schema

The tables to be synchronized are emp and cust. The emp table has three columns called emp_id, emp_name and manager. The cust table has three columns called cust_id, cust_name and emp_id. All columns in each table are synchronized. The mapping from consolidated to remote database is such that the table names and column names are identical in both databases. One additional table, an audit table, is added to the consolidated database.

Java class files

The files used in the example are included in the *Samples\MobiLink\JavaAuthentication* directory.

Create your Java synchronization script

Setup

The following sets up the Java synchronization logic. The import statements tell the Java virtual machine the location of needed files. The public class statement declares the class.

```
// use a package when you create your own script
import   ianywhere.ml.script.InOutInteger;
import   ianywhere.ml.script.DBConnectionContext;
import   ianywhere.ml.script.ServerContext;
import   java.sql.*;
public class CustEmpScripts {
/* Context for this synchronization connection.
*/
    DBConnectionContext _conn_context;
/* Same connection MobiLink uses for sync we can’t commit or
 * close this.
*/
    Connection       _sync_connection;
    Connection       _audit_connection;
/* Get a user id given the user name. On audit connection.
*/
    PreparedStatement      _get_user_id_pstmt;
/* Add record of user logins added. On audit connection.
*/
    PreparedStatement      _insert_login_pstmt;
/* Prepared statement to add a record to the audit table.
 * On audit connection.
*/
    PreparedStatement      _insert_audit_pstmt;
```

The `CustEmpScripts` constructor sets up all the prepared statements for the `authenticateUser` method. It sets up member data.

```
public CustEmpScripts( DBConnectionContext cc )
    throws SQLException
{
try
    {
    _conn_context = cc;
    _sync_connection = _conn_context.getConnection();

    ServerContext serv_context =
    _conn_context.getServerContext();
    _audit_connection = serv_context.makeConnection();

    // get the prep statements ready
    _get_user_id_pstmt =
    _audit_connection.prepareStatement(
        "select user_id from ml_user where name = ?"
        );
    _insert_login_pstmt =

   _audit_connection.prepareStatement(
        "insert into login_added( ml_user, add_time )
        " + " values( ?, { fn CONVERT( { fn NOW() },
        SQL_VARCHAR ) })"
        );
    _insert_audit_pstmt =
        _audit_connection.prepareStatement(
        "insert into login_audit( ml_user_id,
        audit_time, audit_action ) " +
        " values( ?, { fn CONVERT( { fn NOW() },
        SQL_VARCHAR ) }, ? ) "
        );
    } catch ( SQLException e ) {
        freeJDBCResources();
        throw e;
    } catch ( Error e ) {
        freeJDBCResources();
        throw e;
    }
}
```

The finalize method cleans up JDBC resources if end_connection is not called.

```
protected void finalize()
    throws SQLException, Throwable
{
    super.finalize();
    freeJDBCResources();
}
```

The freeJDBCResources method frees allocated memory and we close the audit connection. It is a housekeeping procedure.

```
private void freeJDBCResources()
        throws SQLException
    {
   if( _get_user_id_pstmt != null ) {
       _get_user_id_pstmt.close();
   }
   if( _insert_login_pstmt != null ) {
       _insert_login_pstmt.close();
   }
   if( _insert_audit_pstmt != null ) {
       _insert_audit_pstmt.close();
   }
   if( _audit_connection != null ) {
       _audit_connection.close();
   }
   _conn_context        = null;
   _sync_connection    = null;
   _audit_connection   = null;
   _get_user_id_pstmt  = null;
   _insert_login_pstmt = null;
   _insert_audit_pstmt = null;
    }
```

The `endConnection` method cleans up resources once the resources are not needed. This is also a housekeeping procedure.

```
public void endConnection()
    throws SQLException
{
    freeJDBCResources();
 }
```

The authenticateUser method below approves all user logins and logs user information to database tables. If the user is not in the ml_user table they are logged to login_added. If the user id is found in ml_user then they are logged to login_audit. In a real system we would not ignore the user_password but in order to keep this sample simple we approve all users. The procedure throws SQLException if any of the database operations we perform fail with an exception

```
public void authenticateUser( InOutInteger auth_status,
        String user_name )
    throws SQLException
{
        boolean new_user;
        int user_id;

   // get ml_user id
    _get_user_id_pstmt.setString( 1, user_name );
    ResultSet user_id_rs =
    _get_user_id_pstmt.executeQuery();
    new_user = !user_id_rs.next();
    if( !new_user ) {
        user_id = user_id_rs.getInt(1);
    } else {
        user_id = 0;
    }

   user_id_rs.close();
    user_id_rs = null;
    // in this tutorial always allow the login
    auth_status.setValue( 1000 );
    if( new_user ) {
        _insert_login_pstmt.setString( 1, user_name );
        _insert_login_pstmt.executeUpdate();
        java.lang.System.out.println( "user: " +
            user_name + " added. " );
    } else {
        _insert_audit_pstmt.setInt( 1, user_id );
        _insert_audit_pstmt.setString( 2, "LOGIN ALLOWED" );
        _insert_audit_pstmt.executeUpdate();
    }
    _audit_connection.commit();
    return;
}
```

The following methods use SQL code to act as cursors on the database tables. Since these are cursor scripts, they must return a SQL string.

```
public static String empUploadInsertStmt()
{
    return( "INSERT INTO emp(
       emp_id, emp_name) VALUES( ?, ?) " );
}
public static String empUploadDeleteStmt()
{
    return( "DELETE FROM emp WHERE emp_id = ?" );
}
public static String empUploadUpdateStmt()
{
    return( "UPDATE emp SET emp_name = ?
            WHERE emp_id = ? " );
}
```

```
public static String empDownloadCursor()
{
    return( "SELECT emp_id, emp_name FROM emp" );
}
public static String custUploadInsertStmt()
{
    return( "INSERT INTO cust(
        cust_id, emp_id, cust_name)
        VALUES ( ?, ?, ? ) " );
}
public static String custUploadDeleteStmt()
{
    return( "DELETE FROM cust WHERE cust_id = ? " );
}
public static String custUploadUpdateStmt()
{
    return( "UPDATE cust
            SET emp_id = ?, cust_name = ?
            WHERE cust_id = ? " );
}
public static String custDownloadCursor()
{
    return( "SELECT cust_id, emp_id, cust_name
             FROM cust" );
    }
}
```

This code would be compiled using the command

```
javac -cp %asany9%\java\mlscript.jar CustEmpScripts.jar
```

and we could run the MobiLink synchronization server with the location of CustEmpScripts.class in the classpath. Following is a partial command line:

```
dbmlsrv9 ... -sl java (-cp <class_location>)
```

MobiLink Java API Reference

This section explains the MobiLink Java interfaces and classes, and their associated methods and constructors.

DBConnectionContext interface

Interface for obtaining and accessing information about the current database connection. This is passed to the constructor of classes containing scripts. If context is required for a background thread or beyond the lifetime of a connection, use a ServerContext.

Caution
A DBConnectionContext instance should not be used outside the thread that calls into your Java code.

getConnection method

public java.sql.Connection **getConnection( )**
throws java.sql.SQLException

Returns the existing connection java.sql.Connection as a JDBC connection. The connection is the same connection that MobiLink uses to execute SQL scripts.

This connection must not be committed, closed or altered in any way that would affect the MobiLink server use of this connection. The connection returned is only valid for the lifetime of the underlying MobiLink connection. Do not use the connection after the end_connection event has been called for that connection.

If an error occurs binding the existing connection as a JDBC connection then it throws java.sql.SQLException

If a server connection with full access is required, use ServerContext.makeConnection().

getProperties method

public java.util.Properties **getProperties( )**

Returns the properties for this connection, based on this connection's script version. Properties are stored in the ml_property table.

For more information, see "ml_property" on page 511 and "ml_add_property" on page 486.

getServerContext method

public ServerContext **getServerContext( )**

Returns the ServerContext for this MobiLink server.

getVersion method

public java.util.Properties **getVersion( )**

Returns the name of the script version.

For more information, see “ml_property” on page 511 and “ml_add_property” on page 486.

InOutByteArray interface

Passed into methods to enable the functionality of an in/out parameter passed to a SQL script.

getValue method

public byte[ ] **getValue( )**

Returns the value of this byte array parameter.

setValue method

public void **setValue(** byte[ ] *new_value* **)**

Sets the value of this byte array parameter. There is one parameter, *new_value*, which is the value this byte array should take.

InOutInteger interface

Passed into methods to enable the functionality of an in/out parameter passed to a SQL script.

getValue method

public int **getValue( )**

Returns the value of this integer parameter.

setValue method

public void **setValue(** int *new_value* **)**

Sets the value of this integer parameter. There is one parameter, *new_value*, which is the value this integer should take.

InOutString interface

Passed into methods to enable the functionality of an in/out parameter passed to a SQL script.

getValue method

public java.lang.String **getValue( )**

Returns the value of this string parameter.

setValue method

public void **setValue(** int *new_value* **)**

Sets the value of this integer parameter. There is one parameter, *new_value*, which is the value this string should take.

LogListener interface

The listener interface for catching messages that are printed to the log.

messageLogged method

public void **messageLogged(**
  ServerContext *sc*
  LogMessage *message* **)**

Invoked when a message is printed to the log. There are two parameters: *sc*, which is the context for the server that is printing the message; and *message*, which is the LogMessage that has been sent to the MobiLink log.

LogMessage class

Holds the data associated with a log message.

Extends java.lang.Object

Constants

int **ERROR**

int **WARNING**

getType method

public int **getType( )**

Accessor for this message type.

Returns the type of this message, which can be either ERROR or WARNING.

getUser method

public java.lang.String **getUser( )**

Accessor for this message user. If the message has no user, then the user is NULL.

Returns the user associated with this message.

getText method

public java.lang.String **getText( )**

Accessor for the message text.

Returns the main text of this message.

ServerContext interface

An instantiation of all the context that is present for the duration of the MobiLink server. This context can be held as static data and used in a background thread. It is valid for the duration of the Java virtual machine invoked by MobiLink.

To access a ServerContext instance, use the DBConnectionContext.getServerContext method.

addShutdownListener

public void **addShutdownListener(** ShutdownListener *sl* **)**

Adds the specified ShutdownListener that is to receive notification before the server context is destroyed. On shutdown, the method ShutdownListener.shutdownPerformed (ianywhere.ml.script.ServerContext) is called. There is one parameter, *sl*, which specifies that the ShutdownListener is to be notified on shutdown.

removeShutdownListener

public void **removeShutdownListener(** ShutdownListener *sl* **)**

Removes the specified ShutdownListener from the list of listeners that are to receive notification before the server context is destroyed. There is one parameter, *sl*, which specifies the listener that will no longer be notified on shutdown.

shutdown

public void **shutdown( )**

Forces the server to shut down.

getStartClassInstances

public java.lang.Object[ ] **getStartClassInstances( )**

Gets an array of the start classes that were constructed at server start time. The array length is zero if there are no start classes.

☞ For more information about user-defined start classes, see "User-defined start classes" on page 264.

Following is an example of getStartClassInstances( ):

```
Object objs[] = sc.getStartClassInstances();
int I;
for( I=0; i<objs.length; I+=1 ) {
   if( objs[i] instanceof MyClass ) {
       //use class
   }
}
```

getProperties

public java.util.Properties **getProperties(**
  java.lang.String *component_name*
  java.lang.String *prop_set_name* **)**

Returns the set of properties associated with the script version. These are stored in the MobiLink system table ml_property.

☞ For more information, see "ml_property" on page 511 and "ml_add_property" on page 486.

getPropertiesByVersion

public java.util.Properties **getPropertiesByVersion(** java.lang.String *script_version* **)**

Returns the set of properties associated with the script version. These are stored in the MobiLink system table ml_property. The script version is stored in the prop_set_name column when the component_name is ScriptVersion.

☞ For more information, see "ml_property" on page 511 and "ml_add_property" on page 486.

getPropertySetNames

public java.util.Properties **getPropertySetNames(** java.lang.String *component_name* **)**

Returns the list of property set names for a given component. These are stored in the MobiLink system table ml_property.

☞ For more information, see "ml_property" on page 511 and "ml_add_property" on page 486.

makeConnection method

public java.sql.Connection **makeConnection( )** throws java.sql.SQLException

Creates a new server connection. If an error occurs when opening a new connection, the method throws java.sql.SQLException.

addErrorListener method

public void **addErrorListener(** LogListener *ll* **)**

Adds the specified LogListener to receive a notification when an error is printed.

ll is the LogListener that is to be notified.

The following method will be called: LogListener.messageLogged(ianywhere.ml.script.ServerContext, ianywhere.ml.script.LogMessage).

removeError Listener method

public void **removeErrorListener(** LogListener *ll* **)**

Removes the specified LogListener from the list of listeners that are to receive a notification when an error is printed.

ll is the LogListener that is no longer to be notified.

addWarningListener method

public void **addWarningListener(** LogListener *ll* **)**

Adds the specified LogListener to receive a notification when a warning is printed.

ll is the LogListener that is to be notified.

The following method will be called:

LogListener.messageLogged(ianywhere.ml.script.ServerContext, ianywhere.ml.script.LogMessage).

removeWarningListener method

public void **removeWarningListener(** LogListener *ll* **)**

Removes the specified LogListener from the list of listeners that are to receive a notification when a warning is printed.

ll is the LogListener that is no longer to be notified.

ServerException class

Thrown to indicate that there is an error condition that makes any further synchronization on the server impossible. Throwing this exception causes the MobiLink server to shut down.

ServerException constructors

public **ServerException( )**

Constructs a ServerException with no detail message.

public **ServerException(** java.lang.String *s* **)**

Constructs a ServerException with a specified detail message. There is one parameter, *s*, which specifies the detailed message.

ShutdownListener interface

The listener interface for catching server shutdowns. Use this interface to ensure that all resources, threads, connections, and so on are cleaned up before the server exits.

shutdownPerformed method

public void **shutdownPerformed(** ServerContext *sc***)**

Invoked before the ServerContext is destroyed due to server shutdown. There is one parameter, sc, which is the context for the server that is being shut down.

SynchronizationException class

Thrown to indicate that there is an error condition that makes the completion of the current synchronization impossible. Throwing this exception will force the MobiLink server to rollback.

SynchronizationException constructors

public **SynchronizationException( )**

Constructs a SynchronizationException with no detail message.

public **SynchronizationException(** java.lang.String *s* **)**

Constructs a SynchronizationException with the specified detail message. There is one parameter, s, which specifies a detail message.

CHAPTER 14

Writing Synchronization Scripts in .NET

About this chapter

You control the actions of the MobiLink synchronization server by writing synchronization scripts. You can implement these scripts in SQL, Java, or .NET. This chapter describes how to implement synchronization scripts in .NET.

☞ For a tutorial using .NET synchronization scripts, see "Tutorial: .NET Synchronization Logic With Adaptive Server Anywhere" [*MobiLink Tutorials*, page 65].

☞ For information about writing scripts, see "Writing Synchronization Scripts" on page 227.

☞ For a description and comparison of SQL, Java, and .NET, see "Options for writing synchronization logic" on page 25.

☞ For information about writing scripts in Java, see "Writing Synchronization Scripts in Java" on page 255.

Contents

Introduction

Microsoft .NET is a platform for building, deploying, and running Web services and applications.

MobiLink supports Visual Studio .NET programming languages for writing synchronization scripts. To write MobiLink scripts in .NET, you can use any language that lets you create valid .NET assemblies. In particular, the following languages are tested and documented:

- ♦ C#
- ♦ Visual Basic .NET
- ♦ C++

.NET synchronization logic can function just as Java logic functions: the MobiLink synchronization server can make calls to .NET methods on the occurrence of MobiLink events. A SQL string may be returned to MobiLink.

This section tells you how to set up, develop, and run .NET synchronization logic for C#, Visual Basic .NET, and C++. It includes a sample application and the MobiLink .NET API Reference.

☞ For a tutorial using .NET synchronization scripts, see "Tutorial: .NET Synchronization Logic With Adaptive Server Anywhere" [*MobiLink Tutorials*, page 65].

Setting up .NET synchronization logic

The most important part of implementing synchronization scripts in .NET is telling MobiLink where to find the packages, classes, and methods that are contained in your assemblies.

❖ To implement synchronization scripts in .NET

1. Create your own class or classes. Write a method for each required synchronization event. These methods must be public.

   ☞ For more information about methods, see “Methods” on page 288.

   Each class with non-static methods should have a public constructor. The MobiLink synchronization server automatically instantiates each class the first time a method in that class is called for a connection.

   ☞ For more information about constructors, see “Constructors” on page 287.

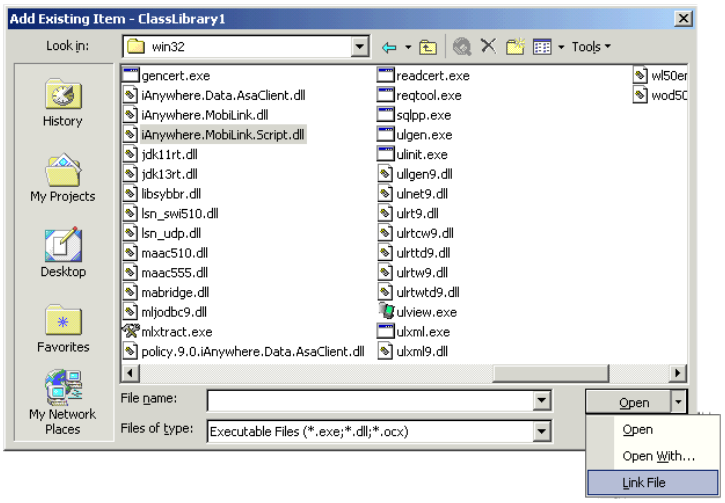
2. Create one or more assemblies. While compiling, reference *iAnywhere.MobiLink.Script.dll* which contains a repository of MobiLink API classes to utilize in your own .NET methods. *iAnywhere.MobiLink.Script.dll* is located in *win32* subdirectory of your SQL Anywhere installation.

   ☞ For more information about the MobiLink .NET API, see “MobiLink .NET API Reference” on page 303.

   You can compile your class on the command line, or using Visual Studio .NET or another .NET development environment.

   ♦ To Compile from Visual Studio .NET:

   a. From the VS.NET Project menu, choose Add Existing Item...
   b. Locate *iAnywhere.MobiLink.Script.dll*; from the Open dropdown list choose Link File:



Caution:
For Visual Studio .NET, always use the Link File method shown above. Do not use the Add Reference option to reference *iAnywhere.MobiLink.Script.dll*. The Add Reference option duplicates *iAnywhere.MobiLink.Script.dll* in the same physical directory as your class assembly, creating problems for the MobiLink synchronization server.

c. Use the Build menu to build your assembly.

♦ To compile from the command line:

You can reference *iAnywhere.MobiLink.Script.dll* from the command line as follows:

```
csc /out:c:\out.dll /target:library /reference:c:\
        iAnywhere.MobiLink.Script.dll sync_v1.cs
```

The above example assumes *iAnywhere.MobiLink.Script.dll* has been copied to *c:\*.

☞ For a complete description of the API, see "MobiLink .NET API Reference" on page 303.

3. In the MobiLink system tables in your consolidated database, specify the name of the package, class, and method to call for each synchronization script. One class is permitted per script version.

   The easiest way to add this information to the MobiLink system tables is to use the ml_add_dnet_connection_script stored procedure or the ml_add_dnet_table_script stored procedure.

   For example, the following statement, when run in an Adaptive Server Anywhere database, specifies that myNamespace.myClass.myMethod should be run whenever the authenticate_user connection-level event occurs.

```
call ml_add_dnet_connection_script( 'version1',
'authenicate_user', 'myNamespace.myClass.myMethod' )
```

Note:
The fully qualified method name is case sensitive.

As a result of this procedure call, the script_language column of the ml_script system table must contain the word **dnet**. The string in the script column, which contains a statement for scripts implemented in SQL, must instead contain the qualified name of a public .NET method.

☞ For more information, see "ml_add_dnet_connection_script" on page 482 and "ml_add_dnet_table_script" on page 483.

You can also add this information using Sybase Central.

4. Instruct the MobiLink server to load assemblies and start the CLR. You tell MobiLink where to locate these assemblies using options on the dbmlsrv9 command line. There are two options to choose from:
   - **Use -sl dnet (-MLAutoLoadPath)** This sets the given path to the application base directory and loads all the private assemblies within it. You should use this option in most cases. For example, to load all assemblies located in *c:\*, enter:

     ```
     dbmlsrv9 -c "dsn=consolidated1" -sl dnet(-
               MLAutoLoadPath=c:\)
     ```

     When you use the -MLAutoLoadPath option you cannot specify a domain when entering the fully qualified method name for the event script.

     ☞ For more information about loading assemblies, see "Loading assemblies" on page 297. For more information about the dbmlsrv9 option -sl dnet, see "-sl dnet option" on page 207.
   - **Use -sl dnet (-MLDomConfigFile)** This option requires a configuration file that contains domain and assembly settings. You should use this option when you have shared assemblies, when you don't want to load all the assemblies in a directory, or when for some other reason you need to use a configuration file.

     ☞ For more information about loading shared assemblies, see "Loading assemblies" on page 297. For more information about the dbmlsrv9 option -sl dnet, see "-sl dnet option" on page 207.

   Note:
   You can use the -MLAutoLoadPath option or the -MLDomConfigFile option, but not both.

Writing .NET synchronization logic

Writing .NET synchronization logic is no different in complexity from writing any other .NET code. What is required from you is knowledge of MobiLink events and familiarity with the MobiLink .NET API. The following sections help you write useful synchronization logic.

☞ For a complete description of the API, see “MobiLink .NET API Reference” on page 303.

.NET synchronization logic can be used to maintain state information, and implement logic around the upload and download events. For example, a begin_synchronization script written in .NET could store the MobiLink user name in a variable. Scripts called later in the synchronization process can access this variable. Also, you can use .NET to access rows in the consolidated database, before or after they are committed.

Using .NET also reduces dependence on the consolidated database. Behavior is affected less by upgrading the consolidated database to a new version or switching to a different database-management system.

Class instances

The MobiLink synchronization server instantiates your classes at the connection level. When an event is reached for which you have written a non-static .NET method, the MobiLink synchronization server automatically constructs the class, if it has not already done so on the present database connection.

☞ For more information, see “Constructors” on page 287.

All methods directly associated with a connection-level or table-level event for one script version *must belong to the same class.*

For each database connection, once a class has been instantiated, the class persists until that connection is closed. Thus, the same instance may well be used for multiple consecutive synchronization sessions. Information present in public or private variables will thus persist across synchronizations that occur on the same connection unless explicitly cleared.

You can also use static classes or variables. In this case, the values are available across all connections in the same domain.

The MobiLink synchronization server automatically deletes your class instances only when the connection to the consolidated database is closed.

Transactions

The normal rules regarding transactions apply to .NET methods. The start and duration of database transactions is critical to the synchronization process. Transactions must be started and ended only by the MobiLink synchronization server. Explicitly committing or rolling back transactions on the synchronization connection within a .NET method violates the integrity of the synchronization process and can cause errors.

These rules apply only to the database connections created by the MobiLink synchronization server and, in particular, to SQL statements returned by methods. If your classes create other database connections, you are free to manage them as you please.

SQL-.NET data types

The following table shows SQL data types and the corresponding .NET data types for MobiLink script parameters.

SQL data type	Corresponding .NET data type
VARCHAR	string
CHAR	string
INTEGER	int
BINARY	byte [ ]
TIMESTAMP	DateTime
INOUT INTEGER	ref int
INOUT VARCHAR	ref string
INOUT CHAR	ref string
INOUT BYTEARRAY	ref byte [ ]

Constructors

The constructor of your class can have one of two possible signatures.

```
public ExampleClass(
        iAnywhere.MobiLink.Script.DBConnectionContext cc )
```

or

```
public ExampleClass()
```

The synchronization context passed to you is for the connection through which the MobiLink synchronization server is synchronizing the current user.

The getConnection method of the DBConnectionContext class returns the same database connection that MobiLink is using to synchronize the present user. You can execute statements on this connection, but you must not commit or roll back the transaction. The MobiLink synchronization server manages the transactions.

The MobiLink synchronization server prefers to use constructors with the first signature. It only uses the void constructor if a constructor with the first signature is not present.

☞ For more information about the DBConnectionContext class, see "DBConnectionContext interface" on page 305.

Methods

In general, you implement one method for each synchronization event. These methods must be public. If they are private, the MobiLink synchronization server cannot use them and will fail to recognize that they exist.

The names of the methods are not important, as long as the names match the names specified in the ml_script table in the consolidated database. In the examples included in the documentation, however, the method names are the same as those of the MobiLink events as this naming convention makes the .NET code easier to read.

The signature of your method should match the signature of the script for that event, except that you can truncate the parameter list if you do not need the values of parameters at the end of the list. Indeed, you should accept only the parameters you need, because overhead is associated with passing the parameters.

You cannot, however, overload the methods. Only one method prototype per class may appear in the ml_script system table.

Return values

Methods called for a MobiLink upload or download must return a valid SQL language statement. The return type of these methods must be String. No other return types are allowed.

The return type of all other scripts must either be string or void. No other types are allowed. If the return type is a string and not null, the MobiLink synchronization server assumes that the string contains a valid SQL statement and executes this statement in the consolidated database as it would an ordinary SQL-language synchronization script. If a method ordinarily returns a string but does not wish to execute a SQL statement against the database upon its return, it can return null.

User-defined start classes

You can define start classes that are loaded automatically when the server is started. The purpose of this feature is to allow you to write .NET code that executes at the time the MobiLink server starts the CLR—before the first synchronization. This means you can create connections or cache data before a user synchronization request.

You do this with the MLStartClasses option of the dbmlsrv9 -sl dnet option. For example, the following is part of a dbmlsrv9 command line. It causes mycl1 and mycl2 to be loaded as start classes.

```
-sl dnet(-MLStartClasses=com.test.mycl1,com.test.mycl2)
```

Classes are loaded in the order in which they are listed. If the same class is listed more than once, more than one instance is created.

All start classes must be public and must have a public constructor that either accepts no arguments or accepts one argument of type MobiLink.Script.ServerContext.

The names of loaded start classes are output to the MobiLink log with the message “Loaded .NET start class: *classname*”.

☞ For more information about .NET CLR, see “-sl dnet option” on page 207.

☞ To see the start classes that are constructed at server start time, see “GetStartClassInstances method” on page 312.

Example

Following is a template start class. It starts a daemon thread that processes events and creates a database connection. (Not all start classes will need to create a thread but if a thread is spawned it should be a daemon thread.)

```
using System;
using System.IO;
using System.Threading;
using iAnywhere.MobiLink.Script;

namespace TestScripts
{
    public class MyStartClass {
   ServerContext     _sc;
   bool              _exit_loop;
   Thread            _thread;
   OdbcConnection    _conn;
```

```
public MyStartClass( ServerContext sc )
//=====================================
{
    // perform setup first so that an exception will
    // cause MobiLink startup to fail
    _sc          = sc;
    // create connection for use later
    _conn        = _sc.makeConnection();
    _exit_loop   = false;
    _thread      = new Thread( new ThreadStart( run ) ) ;
    _thread.IsBackground = true;

    _thread.Start();
}

public void run()
//===============
{
    ShutdownCallback callback = new ShutdownCallback(
      shutdownPerformed );

    _sc.ShutdownListener += callback;
    // we can't throw any exceptions through run()
    try {
   handlerLoop();
   _conn.close();
   _conn = null;
    } catch( Exception e ) {
   // print some error output to the MobiLink log
   Console.Error.Write( e.ToString() );
   // we will die so we don't need to be notified of
   // shutdown
   _sc.ShutdownListener -= callback;
   // ask server to shutdown so that this fatal error will
   // be fixed
   _sc.Shutdown();
    }
    // shortly after return this thread will no longer
    // exist
    return;
}

public void shutdownPerformed( ServerContext sc )
//===============================================
// stop our event handler loop
{
    try {
       _exit_loop = true;
       // wait max 10 seconds for thread to die
       _thread.Join( 10*1000 );
    } catch( Exception e ) {
       // print some error output to the MobiLink log
       Console.Error.Write( e.ToString() );
    }
}
```

```
        private void handlerLoop()
        //========================
        {
            while( !_exit_loop ) {
              // handle events in this loop
              Thread.Sleep( 1*1000 );
            }
        }
         }
     }
```

Printing information from .NET

You may choose to add statements to your .NET methods that print information to the MobiLink log using System.Console. Doing so can help you track the progress and behavior of your classes.

Performance tip
Printing information in this manner is a useful monitoring tool, but is not recommended in a production scenario.

The same technique can be exploited to log arbitrary synchronization information or collect statistical information on how your scripts are used.

Handling MobiLink server errors with .NET

When scanning the log is not sufficient, you can monitor your applications programmatically. For example, you can send messages of a certain type in an email.

You can write methods that are passed a class representing every error or warning message that is printed to the log. This may help you monitor and audit a MobiLink synchronization server.

The following code installs a Listener for all error messages and prints the information to a StreamWriter.

```
class TestLogListener {
    public TestLogListener( StreamWriter output_file  )
    {
        _output_file    = output_file;
    }
    public void errCallback( ServerContext sc, LogMessage lm )
    {
        string type;
        string user;
        if( lm.Type==LogMessage.MessageType.ERROR ) {
            type = "ERROR";
        } else if( lm.Type==LogMessage.MessageType.WARNING ) {
            type = "WARNING";
        } else {
            type = "INVALID TYPE!!";
        }
        if( lm.User == null ) {
            user = "null";
        } else {
            user = lm.User;
        }
        _output_file.WriteLine( "Caught msg type=" + type +
                                        " user=" + user +
                                        " text=" + lm.Text );
        _output_file.Flush();
    }
    StreamWriter    _output_file;
}
// Two lines that registers the TestLogListener Call this code
// from anywhere that has access to the ServerContext such as
// a class constructor or synchronization script.
// ServerContext serv_context;
TestLogListener etll = new TestLogListener(log_listener_file);
serv_context.ErrorListener += new LogCallback(etll.errCallback);
```

See also

"LogCallback delegate" on page 311

ErrorListener and WarningListener in "ServerContext interface" on page 311

"LogMessage class" on page 311

"MessageType enumeration" on page 311

Debugging .NET synchronization logic

The following procedure describes one way you can debug your .NET scripts using Microsoft Visual Studio .NET.

❖ To debug .NET scripts

1. Compile your code with debugging information turned on:
   - ♦ On the csc command line, set the flag **/debug+**

     or
   - ♦ Use Microsoft Visual Studio .NET settings to set debug output
     - From the File menu choose Build ➤ Configuration Manager. From the Active Solution Configuration drop down list, select Debug.
     - Build your assembly.

2. Close running instances of Visual Studio .NET that contain your source files.

   In the next step, you start a new Visual Studio .NET instance to debug the MobiLink synchronization server and your .NET synchronization scripts.

3. Start Visual Studio .NET using a command line option to debug the MobiLink synchronization server.
   - ♦ At a command prompt, navigate to the *Common7\IDE* subdirectory of your Visual Studio .NET installation.
   - ♦ Start devenv (the Visual Studio .NET IDE) using the /debugexe option.

     For example, type the following command line to debug the MobiLink synchronization server. Remember to specify dbmlsrv9 options, including the connection string and the option to load .NET assemblies.

     ```
     devenv /debugexe %asany9%\win32\dbmlsrv9.exe -c ...
     ```

     This causes Visual Studio .NET to start and dbmlsrv9.exe to appear in the Solution Explorer window.

4. Set up Microsoft Visual Studio for debugging .NET code:
   - ♦ In the Visual Studio Solution Explorer window, right-click dbmlsrv9.exe and choose Properties.
   - ♦ Change **Debugger Type** from **Auto** to **Mixed** or **Managed Only**.

     This ensures Visual Studio .NET will only debug your .NET synchronization scripts.

5. Open the associated .NET source files and set break points.

   Note: Open the source files individually in the dbmlsrv9 solution. Do not open the original solution or project file.

6. Start MobiLink from the Debug menu or using F5.

   If prompted, save *dbmlsrv9.sln* in an appropriate location.

   If the dialog No Symbolic Information appears, click OK to debug anyway. You are debugging the managed .NET synchronization scripts that MobiLink calls, not the MobiLink synchronization server itself.

7. Perform a synchronization that causes the code with a breakpoint to be executed by MobiLink.

.NET synchronization techniques

This section describes techniques you can use to tackle common .NET synchronization tasks.

Uploading rows

You can use the upload_insert, upload_update, and upload_delete events to handle rows uploaded to the consolidated database.

For more information about these events, see:

- "upload_insert table event" on page 463
- "upload_update table event" on page 475
- "upload_delete table event" on page 459

The variable number of rows passed to these events are not passed as parameters to the corresponding .NET functions. For special processing on these rows you can insert them into a temporary table and handle the result set using a subsequent event.

❖ To handle rows uploaded to the consolidated database

1. Insert the uploaded values into a temporary table.

   For example, use the following function for the upload_insert event:

   ```
   public static string UploadInsert(/* no parameters */)
    {
      return("INSERT INTO tempULCustomer(cust_id, cust_name)
            values (?,?)");
    }
   ```

2. Use a subsequent script to obtain the values from the temporary table and process the rows.

   Register the following function for the end_upload event:

```
public void EndUpload(string user, string table)
{
   // declare variables to store your row values
   string _custid;
   string _custname;

   // create the command
   DBCommand stmt = currConn.CreateCommand();
   stmt.CommandText = "select * from tempULCustomer";

  // store the result set in a DBRowReader instance
   DBRowReader rset = stmt.ExecuteReader();

   // traverse the result set
   object[] rvalues = rset.NextRow();

   while( rvalues != null )
   {
      _custid = (string)rvalues[0];
      _custname = (string)rvalues[1];

     //... your logic to process the rows

      rvalues = rset.NextRow();
   }

   rset.Close();
   stmt.Close();
}
```

Loading Shared Assemblies

This section details options to load .NET assemblies and details the process to load shared assemblies.

Loading assemblies

A .NET assembly is a package of types, metadata, and executable code. In .NET applications, all code must be in an assembly. Assembly files have the extension *.dll* or *.exe*.

There are two types of assemblies:

- ♦ **Private assemblies** A private assembly is a file in the file system.
- ♦ **Shared assemblies** A shared assembly is an assembly that is installed in the global assembly cache.

Before MobiLink can load a class and call a method of that class, it must locate the assembly that contains the class. MobiLink only needs to locate the assembly that it calls directly. The assembly can then call any other assemblies it needs.

For example, MobiLink calls MyAssembly, and MyAssembly calls UtilityAssembly and NetworkingUtilsAssembly. In this situation, MobiLink only needs to be configured to find MyAssembly.

MobiLink provides two ways to load assemblies:

- ♦ **Use -sl dnet (-MLAutoLoadPath)** This option only works with private assemblies. It sets the path to the application base directory and loads all the assemblies within it. This option is simpler to use and it is expected that it will be sufficient in most cases.

  When you use the -MLAutoLoadPath option you cannot specify a domain when entering the fully qualified method name for the event script.

  When you specify a path and directory with -MLAutoLoadPath, MobiLink does the following:

  - sets this path as the application base path
  - loads all classes in all files ending with *.dll* or *.exe* in the directory that you specified
  - creates one application domain and loads into that domain all user classes that do not have a domain specified

  Assemblies in the global assembly cache cannot be called directly with this option. To call these shared assemblies, use -MLDomConfigFile.

♦ **Use -sl dnet (-MLDomConfigFile)** This option works with both private and shared assemblies. It requires a configuration file that contains domain and assembly settings. You should use this option when you have shared assemblies, when you don’t want to load all the assemblies in the application base path, or when for some other reason you need to use a configuration file.

  With this option, MobiLink reads the settings in the specified domain configuration file. A domain configuration file contains configuration settings for one or more .NET domains. If there is more than one domain represented in the file, the first one that is specified is used as the default domain. (The default domain is used when scripts do not have a domain specified.)

  When loading assemblies, MobiLink tries to load the assembly first as private, and then attempts to load the assembly from the global assembly cache. Private assemblies must be located in the application base directory. Shared assemblies are loaded from the global assembly cache.

With the -MLDomConfigFile option, only assemblies that are specified in the domain configuration file can be called directly from event scripts.

Sample domain configuration file

A sample domain configuration file called *mlDomConfig.xml* is installed with MobiLink. You can write your own file from scratch, or edit the sample to suit your needs. The sample file is located in the SQL Anywhere Studio path, in

MobiLink\setup\dnet\mlDomConfig.xml

Following is the content of the sample domain configuration file *mlDomConfig.xml*:

```
<?xml version="1.0" encoding="utf-8"?>
<config xmlns="iAnywhere.MobiLink.mlDomConfig"
         xsi:schemaLocation='iAnywhere.MobiLink.mlDomConfig
         mlDomConfig.xsd'
         xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance' >
   <domain>
      <name>SampleDomain1</name>
      <appBase>C:\scriptsDir</appBase>
      <configFile></configFile>
      <assembly name="Assembly1" />
      <assembly name="Assembly2" />
   </domain>
   <domain>
      <name>SampleDomain2</name>
      <appBase>\Dom2assembly</appBase>
      <configFile>\Dom2assembly\
         AssemblyRedirects.config</configFile>
      <assembly name="Assembly3" />
      <assembly name="Assembly4" />
   </domain>
</config>
```

Following is an explanation of the contents of *mlDomConfig.xml*:

♦ **name** is the domain name, used when specifying the domain in an event script. An event script with the format `"DomainName:Namespace.Class.Method"` would require a domain called DomainName be in the domain configuration file.

  You must specify at least one domain name.

♦ **appBase** is the directory that the domain should use as its application base directory. All private assemblies are loaded by the .NET CLR based on this directory. You must specify appBase.

♦ **configFile** is the .NET application configuration file that should be used for the domain. This can be left blank. It is usually used to modify the default assembly binding and loading behavior. Refer to your .NET documentation for more information about application configuration files.

♦ **assembly** is the name of an assembly that MobiLink should load and search when resolving type references in event scripts. You must specify at least one assembly. If an assembly is used in more than one domain, it must be specified as an assembly in each domain. If the assembly is private, it must be in the application base directory for the domain.

☞ For more information about the dbmlsrv9 option -sl dnet, see "-sl dnet option" on page 207.

.NET synchronization example

This example modifies an existing application to describe how to use .NET synchronization logic to handle the authenticate_user event. It creates a C# script for authenticate_user called *AuthUser.cs*. This script looks up the user's password in a table called user_pwd_table and authenticates the user based on that password.

Note:
This section provides a simple example to illustrate basic .NET synchronization logic. Typically, reasons for using a custom user authentication mechanism include integration with existing DBMS user authentication schemes, or supplying custom features, such as minimum password length or password expiry. MobiLink also has a built-in default mechanism.

☞ For more information about MobiLink authentication, see "Choosing a user authentication mechanism" [*MobiLink Clients*, page 13].

First, add the table user_pwd_table to the database. Execute the following in Interactive SQL:

```
CREATE TABLE user_pwd_table (
  user_name  varchar(128) PRIMARY KEY NOT NULL,
  pwd        varchar(128)
)
```

Next, add a user and password to the table:

```
INSERT INTO user_pwd_table VALUES( 'user1', 'myPwd' )
```

Create a directory for your .NET assembly. For example:

```
mkdir c:\mlexample
```

Create a file called *AuthUser.cs* with the following contents:

☞ For more information, see "authenticate_user connection event" on page 336.

```
using System;
using iAnywhere.MobiLink.Script;

namespace MLExample
{

  public class AuthClass
  {
     private DBConnection  _conn;
  /// AuthClass constructor.

public AuthClass( DBConnectionContext cc )
{
   _conn   = cc.GetConnection();
}
/// The DoAuthenticate method handles the 'authenticate_user'
 /// event.
  /// Note: This method does not handle password changes for
/// advanced authorization status codes.

public void DoAuthenticate(
          ref int authStatus,
          string user,
          string pwd,
          string newPwd )
{
    DBCommand   pwd_command = _conn.CreateCommand();
    pwd_command.CommandText = "select pwd from user_pwd_table"
                           + " where user_name = ? ";
    pwd_command.Prepare();
      // add a parameter for the user name
    DBParameter user_param = new DBParameter();
    user_param.DbType = SQLType.SQL_CHAR;
    // we need to set the size for SQL_VARCHAR
    user_param.Size   = (uint)user.Length;
    user_param.Value  = user;
    pwd_command.Parameters.Add( user_param );
   // fetch the password for this user.
    DBRowReader   rr  = pwd_command.ExecuteReader();
    object[]   pwd_row  = rr.NextRow();
    if( pwd_row == null ) {
    // user is unknown
    authStatus  = 4000;
    } else {
    if( ((string)pwd_row[0]) == pwd ) {
         // password matched
         authStatus = 1000;
    } else {
         // password did not match
         authStatus = 4000;
    }
    }
    pwd_command.Close();
    rr.Close();
    return;
}
}
}
```

The MLExample.AuthClass.DoAuthenticate method handles the authenticate_user event. It accepts the user name and password and returns an authorization status code indicating the success or failure of the validation.

Compile the file *AuthUser.cs*. You can do this on the command line or in Visual Studio .NET.

For example, the following command line will compile *AuthUser.cs* and generate an Assembly named *example.dll* in *c:\mlexample*. Substitute your install directory for *asany9*.

```
csc /out:c:\mlexample\example.dll /target:library /reference:\
         asany9\win32\iAnywhere.MobiLink.Script.dll AuthUser.cs
```

Register .NET code for the authenticate_user event. The method you need to execute (DoAuthenticate) is in the namespace MLExample and class AuthClass. Execute the following SQL:

```
call ml_add_dnet_connection_script( 'ex_version', 'authenticate_
         user', 'MLExample.AuthClass.DoAuthenticate' )
COMMIT
```

Next, run the MobiLink synchronization server with the following option. This option causes MobiLink to load all assemblies in *c:\myexample*:

```
-sl dnet ( -MLAutoLoadPath=c:\mlexample )
```

Now, when a user synchronizes with the version ex_version, they are authenticated with the password from the table user_pwd_table.

MobiLink .NET API Reference

This section explains the MobiLink .NET interfaces and classes, and their associated methods, properties, and constructors. To use these classes, reference the assembly \win32\iAnywhere.MobiLink.Script.dll in your SQL Anywhere Studio installation directory.

This section focuses on C#, but there are equivalents in VB.NET and C++.

DBCommand interface

public interface **DBCommand**
Member of **iAnywhere.MobiLink.Script**

Represents a SQL statement or database command. DBCommand can represent an update or query.

For example, the following C# code uses the DBCommand interface to execute two queries:

```
DBCommand stmt = conn.CreateCommand();

stmt.CommandText = "select t1a1, t1a2 from table1 ";

DBRowReader rs = stmt.ExecuteReader();
printResultSet( rs );
rs.Close();
stmt.CommandText = "select t2a1 from table2 ";

rs = stmt.ExecuteReader();
printResultSet( rs );
rs.Close();
stmt.Close();
```

The following C# example uses DBCommand to execute an update with parameters:

```
DBCommand cstmt = conn.CreateCommand();

cstmt.CommandText = "call myProc( ?,?,? )";

cstmt.Prepare();

DBParameter param = new DBParameter();
param.DbType          = SQLType.SQL_CHAR;
param.Value           = "10000";
cstmt.Parameters.Add( param );
param                 = new DBParameter();
param.DbType          = SQLType.SQL_INTEGER;
param.Value           = 20000;
cstmt.Parameters.Add( param );
```

```
param                  = new DBParameter();
param.DbType           = SQLType.SQL_DECIMAL;
param.Precision        = 5;
param.Value            = new Decimal( 30000 );
cstmt.Parameters.Add( param );
// Execute update
DBRowReader rset = cstmt.ExecuteNonQuery();
cstmt.Close();
```

Prepare method

public void **Prepare( )**

Prepare the SQL statement stored in CommandText for execution.

ExecuteNonQuery() method

public int **ExecuteNonQuery( )**

Execute a non-query statement. Returns the number of rows in the database affected by the SQL statement.

ExecuteReader() method

public DBRowReader **ExecuteReader( )**

Execute a query statement returning the result set. Returns a DBRowReader for retrieving results returned by the SQL statement.

Close() method

public void **Close( )**

Close the current SQL statement or command.

CommandText property

public string **CommandText**

The value is the SQL statement to be executed.

DBParameterCollection Parameters property

public **DBParameterCollection Parameters**

Gets the iAnywhere.MobiLink.Script.DBParameterCollection for this DBCommand.

DBConnection interface

public interface **DBConnection**
Member of **iAnywhere.MobiLink.Script**

Represents a MobiLink ODBC connection.

This interface allows user-written synchronization logic to access an ODBC connection created by MobiLink.

Commit() method

public void **Commit( )**

Commit the current transaction.

Rollback() method

public void **Rollback( )**

Roll back the current transaction.

Close() method

public void **Close( )**

Close the current connection.

CreateCommand() method

public DBCommand **CreateCommand( )**

Create a SQL statement or command on this connection. Returns the newly generated DBCommand.

DBConnectionContext interface

public interface **DBConnectionContext**
Member of **iAnywhere.MobiLink.Script**

Interface for obtaining and accessing information about the current database connection. This is passed to the constructor of classes containing scripts. If context is required for a background thread or beyond the lifetime of a connection, use a ServerContext.

Caution:
A DBConnectionContext instance should not be used outside the thread that calls into your .NET code.

Interface for obtaining information about the current database connection. This is passed to the constructor of classes containing scripts.

GetConnection method

public **iAnywhere.MobiLink.Script.DBConnection GetConnection( )**
Member of **iAnywhere.MobiLink.Script.DBConnectionContext**

Returns the existing connection. The connection is the same connection that MobiLink uses to execute SQL scripts.

This connection must not be committed, closed or altered in any way that would affect the MobiLink server use of the connection. The connection returned is only valid for the lifetime of the underlying MobiLink connection. Do not use the connection after the end_connection event has been called for the connection.

If a server connection with full access is required, use ServerContext.makeConnection().

GetServerContext method

public **iAnywhere.MobiLink.Script.ServerContext.GetServerContext( )**
Member of **iAnywhere.MobiLink.Script.DBConnectionContext**

Returns the ServerContext for this MobiLink server.

getProperties method

NameValueCollection **getProperties( )**

Returns the properties for this connection, based on this connection's script version. Properties are stored in the ml_property table.

For more information, see "ml_property" on page 511 and "ml_add_property" on page 486.

getVersion method

string **getVersion( )**

Returns the name of the script version.

For more information, see "ml_property" on page 511 and "ml_add_property" on page 486.

DBParameter class

public class **DBParameter**
Member of **iAnywhere.MobiLink.Script**

Represents a bound ODBC parameter.

DBParameter is required to execute commands with parameters. All parameters must be in place before the command is executed.

For example, the following C# code uses DBCommand to execute an update with parameters:

```
DBCommand cstmt = conn.CreateCommand();

  cstmt.CommandText = "call myProc( ?,?,? )";

  cstmt.Prepare();

  DBParameter param = new DBParameter();
  param.DbType           = SQLType.SQL_CHAR;
  param.Value            = "10000";
  cstmt.Parameters.Add( param );

  param                  = new DBParameter();
  param.DbType           = SQLType.SQL_INTEGER;
  param.Value            = 20000;
  cstmt.Parameters.Add( param );

  param                  = new DBParameter();
  param.DbType           = SQLType.SQL_DECIMAL;
  param.Precision        = 5;
  param.Value            = new Decimal( 30000 );
  cstmt.Parameters.Add( param );

  // Execute update
  DBRowReader rset = cstmt.ExecuteNonQuery();
  cstmt.Close();
```

dbType property

public **SQLTYPE dbType**

The value is the SQLType of this parameter.

Default: SQLType.SQL_TYPE_NULL.

Direction property

public **System.Data.ParameterDirection Direction**

The value is the Input/Output direction of this parameter.

Default: ParameterDirection.Input.

IsNullable property

public bool **IsNullable**

The value Indicates whether this parameter can be NULL.

Default: false.

ParameterName property

public string **ParameterName**

The value is the name of this parameter.

Default: null.

Precision property

public uint **Precision**

The value is the decimal precision of this parameter. Only used for SQLType.SQL_NUMERIC and SQLType.SQL_DECIMAL parameters.

Default: 0.

Scale property

public short **Scale**

The value is the resolvable digits of this parameter. Only used for SQLType.SQL_NUMERIC and SQLType.SQL_DECIMAL parameters.

Default: 0.

Size property

public uint **Size**

The value is the size in bytes of this parameter.

Default: Inferred from DbType.

Value property

public object **Value**

The value is the value of this parameter.

Default: null.

DBParameterCollection class

public class **DBParameterCollection**
inherits from **IDataParameterCollection**, **IList**, **ICollection**, **IEnumerable**
Member of **iAnywhere.MobiLink.Script**

Collection of DBParameters. When DBCommand creates a

DBParamterCollection it is empty and must be filled with appropriate parameters before the DBCommand executes.

DBParameterCollection( ) method

public **DBParameterCollection( )**

Creates an empty list of DBParameters.

Contains(string parameterName) method

public bool **Contains(string** *parameterName* **)**

Returns true if the collection contains a parameter with the specified name. Takes one parameter, *parameterName*, which is the name of the parameter.

IndexOf(string parameterName) method

public int **IndexOf(string** *parameterName* **)**

Returns index of the parameter, or -1 if there is no parameter with the given name. Takes one parameter, *parameterName*, which is the name of the parameter.

RemoveAt(string parameterName) method

public void **RemoveAt(string** *parameterName* **)**

Removes the parameter with the given name from the collection. Takes one parameter, *parameterName*, which is the name of the parameter.

Add(object value) method

public int **Add(object** *value* **)**

Adds the given parameter to the collection. Takes one parameter, *value*, which is the iAnywhere.MobiLink.Script.DBParameter to add to the collection. Returns the index of the added parameter in the collection.

Clear() method

public void **Clear( )**

Removes all parameters from the collection.

Contains(object value) method

public bool **Contains(object** *value* **)**

Returns true if this collection contains the given iAnywhere.MobiLink.Script.DBParameter. Takes one parameter, *value*, which is the iAnywhere.MobiLink.Script.DBParameter.

IndexOf(object value) method

public int **IndexOf(object** *value* **)**

Returns the index of the given iAnywhere.MobiLink.Script.DBParameter in the collection. Takes one parameter, *value*, which is the iAnywhere.MobiLink.Script.DBParameter.

Insert(int index, object value) method

public void **Insert(int** *index***, object** *value* **)**

Inserts the given iAnywhere.MobiLink.Script.DBParameter into the collection at the specified index. Takes two parameters: *value*, which is the

	iAnywhere.MobiLink.Script.DBParameter; and *index*, which is the index to insert at.
Remove(object value) method	public void **Remove(object** *value* **)** Removes the given iAnywhere.MobiLink.Script.DBParameter from the collection. Takes one parameter, *value*, which is the iAnywhere.MobiLink.Script.DBParameter.
RemoveAt(int index) method	public int **RemoveAt(int index)** Removes the iAnywhere.MobiLink.Script.DBParameter at the given index in the collection. Takes one parameter, *index*, which is the index of the iAnywhere.MobiLink.Script.DBParameter.
CopyTo(Array array, int index) method	public void **CopyTo(Array** *array* **, int** *index* **)** Copies the contents of the collection into the given array starting at the specified index. Takes two parameters: *array*, which is the array to copy the contents of the collection into; and *index*, which is the index in the array to begin copying the contents of the collection into.
GetEnumerator() method	public IEnumerator **GetEnumerator( )** Returns an enumerator for the collection.
IsFixedSize property	public bool **IsFixedSize** Returns false.
IsReadOnly property	public bool **IsReadOnly** Returns false.
Count property	public int **Count** The number of parameters in the collection.
IsSynchronized property	public bool **IsSynchronized** Returns false.
SyncRoot property	public object **SyncRoot** Object that can be used to synchronize the collection.
this[string parameterName] property	public object **this[string** *parameterName* **]** Gets or sets the iAnywhere.MobiLink.Script.DBParameter with the given name in the collection. Takes one parameter, *parameterName*, which is the name of the iAnywhere.MobiLink.Script.DBParameter to get or set.

this[int index] property

public object **this[int** *index* **]**

Gets or sets the iAnywhere.MobiLink.Script.DBParameter at the given index in the collection. Takes one parameter, *index*, which is the index of the iAnywhere.MobiLink.Script.DBParameter to get or set.

DBRowReader interface

public interface **DBRowReader**
Member of **iAnywhere.MobiLink.Script**

Represents a set of rows being read from a database. Executing the method DBCommand.executeReader( ) creates a DBRowReader.

The following example is a C# code fragment. It calls a function with the rows in the result set represented by the given DBRowReader.

```
DBCommand stmt = conn.CreateCommand();

stmt.CommandText = "select intCol, strCol from table1 ";

DBRowReader rs = stmt.ExecuteReader();
object[] values = rset.NextRow();
while( values != null ) {
handleRow( (int)values[0], (String)values[1] );
 values = rset.NextRow();
}
rset.Close();
stmt.Close();
```

NextRow( ) method

public object[ ] **NextRow( )**

Retrieves and returns the next row of values in the result set. If there are no more rows in the result set, it returns NULL.

☞ See "SQLType enumeration" on page 314.

Close( ) method

public void **Close( )**

Cleans up resources used by this MLDBRowReader. After Close( ) is called, this MLDBRowReader cannot be used again.

ColumnNames property

public string[ ] **ColumnNames**

Gets the names of all columns in the result set. The value is an array of strings corresponding to the column names in the result set.

ColumnTypes property

public SQLType[ ] **ColumnTypes**

Gets the types of all columns in the result set. The value is an array of SQLTypes corresponding to the column types in the result set.

LogCallback delegate

public delegate void **LogCallback(**
  ServerContext *sc*
  LogMessage *message*
)
Member of **iAnywhere.MobiLink.Script**

Called when the MobiLink synchronization server prints a message.

LogMessage class

public class **LogMessage** : **iAnywhere.MobiLink.Script.LogMessage**
Member of **iAnywhere.MobiLink.Script**

Contains information about a message printed to the log.

Type property

public LogMessage.MessageType **Type**

The type of the log message that this instance represents.

User property

public string **User**

The user for which this message is being logged. It may be null.

Text property

public string **Text**

The main text of the message.

MessageType enumeration

public enum **MessageType**
Member of **iAnywhere.MobiLink.Script.LogMessage**

Enumeration of the possible types of LogMessage.

ERROR field

public **ERROR**

A log error message.

WARNING field

public **WARNING**

A log warning message.

ServerContext interface

public interface **ServerContext**
Member of **iAnywhere.MobiLink.Script**

An instantiation of all the context that is present for the duration of the MobiLink server. This context can be held as static data and used in a background thread. It is valid for the duration of the .NET CLR invoked by MobiLink.

To access a ServerContext instance, use the DBConnectionContext.getServerContext method.

GetStartClassInstances method

public **object[ ] GetStartClassInstances( )**
Member of **iAnywhere.MobiLink.Script.ServerContext**

Gets an array of the start classes that were constructed at server start time. The array length is zero if there are no start classes.

☞ For more information about user-defined start classes, see "User-defined start classes" on page 289.

Following is an example of getStartClassInstances():

```
void FindStartClass( ServerContext sc, string name )
 {
   object[] startClasses = sc.GetStartClassInstances();
   foreach( object obj in startClasses ) {
     if( obj is MyClass ) {
         // Execute some code.....
     }
   }
 }
```

LogCallback ErrorListener event

This event is triggered when the MobiLink synchronization server prints an error.

LogCallback WarningListener event

This event is triggered when the MobiLink synchronization server prints a warning.

MakeConnection method

public **iAnywhere.MobiLink.Script.DBConnection makeConnection( )**
Member of **iAnywhere.MobiLink.Script.ServerContext**

Creates a new server connection.

ShutDown method

public void **Shutdown( )**
Member of **iAnywhere.MobiLink.Script.ServerContext**

Forces the server to shut down.

ShutdownListener method

public event iAnywhere.MobiLink.Script.ShutdownCallback
 ShutdownListener(
 iAnwyhere.MobiLink.Script.ServerContext *sc***)**
Member of **iAnywhere.MobiLink.Script.ServerContext**

This event is triggered on shutdown. The following code is an example of how to use this event:

```
ShutdownCallback callback = new ShutdownCallback(
         shutdownHandler );
_sc.ShutdownListener += callback;
public void shutdownHandler( ServerContext sc )
//=============================================
{
_test_out_file.WriteLine( "shutdownPerformed" );
}
```

getProperties method

NameValueCollection **getProperties(**
string *component_name*
string *prop_set_name* **)**

Returns the set of properties associated with the script version. These are stored in the MobiLink system table ml_property.

☞ For more information, see "ml_property" on page 511 and "ml_add_property" on page 486.

getPropertiesByVersion method

NameValueCollection **getPropertiesByVersion(** string *script_version* **)**

Returns the set of properties associated with the script version. These are stored in the MobiLink system table ml_property. The script version is stored in the prop_set_name column when the component_name is ScriptVersion.

☞ For more information, see "ml_property" on page 511 and "ml_add_property" on page 486.

getPropertySetNames method

StringCollection **getPropertySetNames(** string *component_name* **)**

Returns the list of property set names for a given component. These are stored in the MobiLink system table ml_property.

☞ For more information, see "ml_property" on page 511 and "ml_add_property" on page 486.

ServerException class

public class **ServerException**
Member of **iAnywhere.MobiLink.Script**

Used to signal MobiLink that an error has occurred with the server and it should shut down immediately.

ServerException constructors

public **ServerException( )**
Member of **iAnywhere.MobiLink.Script.ServerException**

Constructs a ServerException with no detail message.

public **ServerException(string** *message* **)**
Member of **iAnywhere.MobiLink.Script.ServerException**

Creates a new ServerException with the given message. The parameter *message* is the message for this ServerException.

public **ServerException(string** *message*, **SystemException** *ie* **)**
Member of **iAnywhere.MobiLink.Script.ServerException**

Creates a new ServerException with the given message and containing the given inner exception that caused this one. There are two parameters: *message*, which is the message for this ServerException, and *ie*, which is the exception that caused this ServerException.

ShutdownCallback delegate

public sealed delegate **ShutdownCallback : System.MulticastDelegate**
Member of **iAnywhere.MobiLink.Script**

Called when the MobiLink synchronization server is shutting down. Implementations of this delegate can be registered with the ServerContext.ShutdownListener event to be called when the MobiLink server shuts down.

SQLType enumeration

public enum **SQLType**
Member of **iAnywhere.MobiLink.Script**

Enumeration of all possible ODBC data types.

SQL_TYPE_NULL field

public **SQL_TYPE_NULL**

Null data type.

SQL_UNKNOWN_TYPE field

public **SQL_UNKNOWN_TYPE**

Unknown data type.

SQL_CHAR field

public **SQL_CHAR**

UTF-8 character array of a set size. Has .NET type String.

SQL_NUMERIC field

public **SQL_NUMERIC**

Numeric value of set size and precision. Has .NET type Decimal.

SQL_DECIMAL field

public **SQL_DECIMAL**

Decimal number of set size and precision. Has .NET type Decimal.

SQL_INTEGER field	public **SQL_INTEGER** 32-bit integer. Has .NET type Int32.
SQL_SMALLINT field	public **SQL_SMALLINT** 16-bit integer. Has .NET type Int16.
SQL_FLOAT field	public **SQL_FLOAT** Floating point number with ODBC driver defined precision. Has .NET type Double.
SQL_REAL field	public **SQL_REAL** Single precision floating-point number. Has .NET type Single.
SQL_DOUBLE field	public **SQL_DOUBLE** Double precision floating point number. Has .NET type Double.
SQL_DATE field	public **SQL_DATE** A date. Has .NET type DateTime.
SQL_DATETIME field	public **SQL_DATETIME** A date and time. Has .NET type DateTime.
SQL_TIME field	public **SQL_TIME** A time. Has .NET type DateTime.
SQL_INTERVAL field	public **SQL_INTERVAL** An interval of time. Has .NET type TimeSpan.
SQL_TIMESTAMP field	public **SQL_TIMESTAMP** A time stamp. Has .NET type DateTime.
SQL_VARCHAR field	public **SQL_VARCHAR** A null terminated UTF-8 string with a user set maximum length. Has .NET type String.
SQL_TYPE_DATE field	public **SQL_TYPE_DATE** A date. Has .NET type DateTime.
SQL_TYPE_TIME field	public **SQL_TYPE_TIME** A time. Has .NET type DateTime.

SQL_TYPE_-TIMESTAMP field	public **SQL_TYPE_TIMESTAMP** A timestamp. Has .NET type DateTime.
SQL_DEFAULT field	public **SQL_DEFAULT** A default type. Has no type.
SQL_ARD_TYPE field	public **SQL_ARD_TYPE** An ARD object. Has no type.
SQL_BIT field	public **SQL_BIT** A single bit. Has .NET type Boolean.
SQL_TINYINT field	public **SQL_TINYINT** An 8-bit integer. Has .NET type SByte.
SQL_BIGINT field	public **SQL_BIGINT** A 64-bit integer. Has .NET type Int64.
SQL_LONGVARBINARY field	public **SQL_LONGVARBINARY** Variable length binary data with a driver dependent maximum length. Has .NET type byte[ ].
SQL_VARBINARY field	public **SQL_VARBINARY** Variable length binary data with a user specified maximum length. Has .NET type byte[ ].
SQL_BINARY field	public **SQL_BINARY** Fixed length binary data. Has .NET type byte[ ].
SQL_LONGVARCHAR field	public **SQL_LONGVARCHAR** A null-terminated UTF-8 string with a driver-dependent maximum length. Has .NET type String.
SQL_GUID field	public **SQL_GUID** A Global Unique ID (also called a UUID). Has .NET type Guid.
SQL_WCHAR field	public **SQL_WCHAR** Unicode character array of fixed size. Has .NET type String.
SQL_WVARCHAR field	public **SQL_WVARCHAR**

Null-terminated Unicode string of user-defined maximum length. Has .NET type String.

SQL_WLONGVARCHAR field

public **SQL_WLONGVARCHAR**

Null-terminated Unicode string of driver-dependent maximum length. Has .NET type String.

SynchronizationException class

public class **SynchronizationException**
Member of **iAnywhere.MobiLink.Script**

Used to signal that a synchronization exception has occurred and that the current synchronization should be rolled back and restarted.

SynchronizationException constructors

public **SynchronizationException( )**
Member of **iAnywhere.MobiLink.Script.SynchronizationException**

Constructs a SynchronizationException with no details.

public **SynchronizationException(string** *message* **)**
Member of **iAnywhere.MobiLink.Script.SynchronizationException**

Creates a new SynchronizationException with the given message. The parameter *message* is the message for this ServerException.

public **SynchronizationException(string** *message*, **SystemException** *ie* **)**
Member of **iAnywhere.MobiLink.Script.SynchronizationException**

Creates a new SynchronizationException with the given message and containing the given inner exception that caused this one. There are two parameters: *message*, which is the message for this ServerException, and *ie*, which is the exception that caused this ServerException.

CHAPTER 15

Synchronization Events

About this chapter

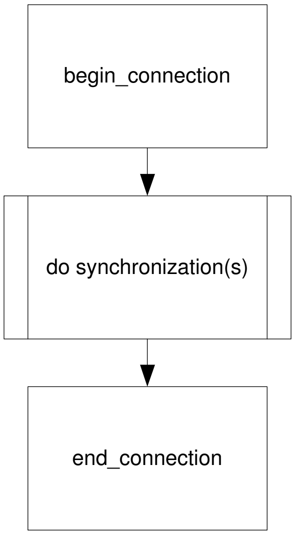
This chapter provides information about the MobiLink synchronization events and the SQL scripts, Java methods, or .NET methods that handle these events. You implement scripts to handle one or more of these events to control the actions of the MobiLink synchronization server.

☞ For information about storing scripts, see "Adding and deleting scripts in your consolidated database" on page 241.

Contents

Overview of MobiLink events

When a synchronization request occurs and MobiLink server decides that a new connection must be created, the begin_connection event is fired and synchronization starts.

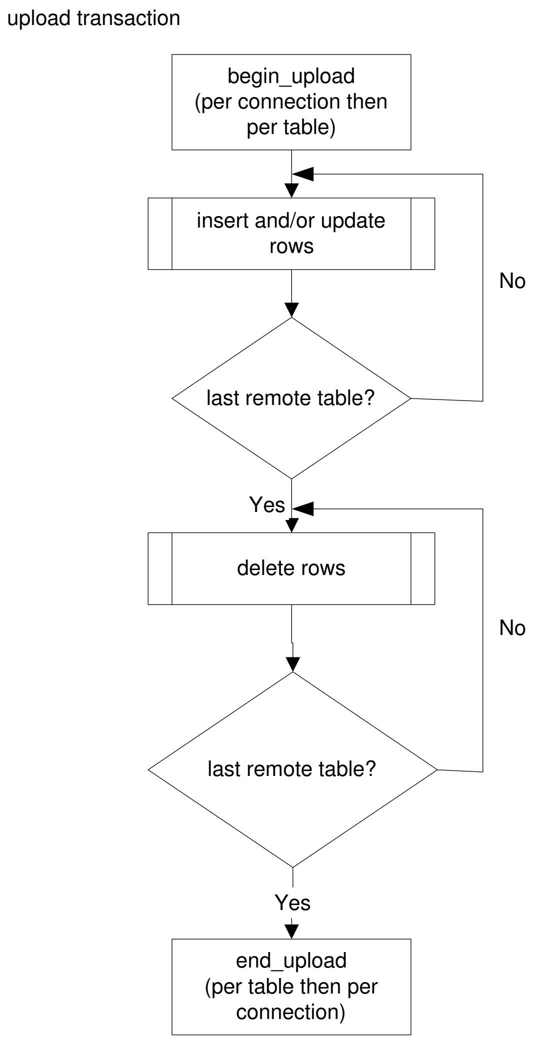


Following the synchronization, the connection is placed in a connection pool, and MobiLink again waits for a synchronization request for the current script version. Before a connection is eventually dropped from the connection pool, the end_connection event is fired. But if another synchronization request for the same version is received, then MobiLink handles the next synchronization request on the same connection. There are a number of events that affect the current synchronization.

The primary phases of a synchronization are the upload and download transactions. The events contained in the upload and download transactions are outlined below.

The upload transaction The upload transaction applies changes uploaded from a remote database.

The begin_upload event marks the beginning of the upload transaction. The upload transaction is a two-part process. First, inserts and updates are uploaded for all remote tables, and second, deletes are uploaded for all remote tables.


upload transaction
begin_upload (per connection then per table)
insert and/or update rows
last remote table?
No
Yes
delete rows
last remote table?
No
Yes
end_upload (per table then per connection)

The end_upload event marks the end of the upload transaction.

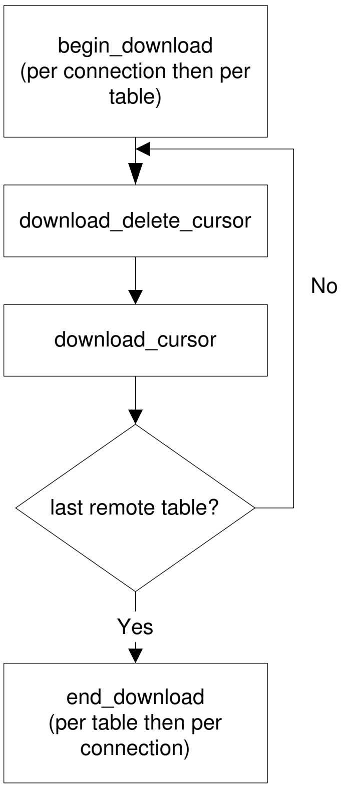
For more information about the events that happen during upload, see "Writing scripts to upload rows" on page 244.

The download transaction

The download transaction fetches rows from the consolidated database. It begins with the begin_download event.

The download transaction is a two-part process. For each table, first deletes are downloaded, and then update/insert rows (upserts) are downloaded. The end_download event ends the download transaction.

download transaction



☞ For more information about the events that happen during download, see "Writing scripts to download rows" on page 246.

The following pseudocode provides an overview of the sequence in which events, and hence the script of the same name, are invoked.

Event overview in pseudocode

The following pseudocode shows the complete MobiLink synchronization event model. This model assumes a full synchronization (not upload-only or download-only) with no errors.

Notes

- In most cases, if you have not defined a script for a given event, the default action is to do nothing.
- The begin_connection and end_connection events are **connection-level**

events. They are independent of any single synchronization and have no parameters.

- Some events are invoked once per synchronization for each table being synchronized. Scripts associated with these events are called **table-level scripts**.

  While each table can have its own table scripts, you can also write table-level scripts that are shared by several tables.

- Some events, such as begin_synchronization, occur at both the connection level and the table level. You can supply both connection and table scripts for these events.
- The COMMIT statements illustrate how the synchronization process is broken up into distinct transactions.
- A database error can occur at any point within the synchronization process. Database errors are handled using the handle_error or handle_odbc_error scripts.

Warning
There should be no implicit or explicit commit or rollback in your synchronization scripts or the procedures or triggers that are called from your synchronization scripts. COMMIT or ROLLBACK statements within scripts alter the transactional nature of the synchronization steps. If you use them, you cannot guarantee the integrity of your data in the event of a failure.

```
------------------------------------------------------
Synchronization events in pseudocode.

Legend:
- // This is a comment
- <name>
    The pseudo code for <name> is listed separately
    in a later section, under a banner:
        ------------------------
        name
        ------------------------
- VariableName <- value
    Assign the given value to the given variable name.
    Variable names are in mixed case.
- event_name
    If you have defined a script for the given event name,
    it will be invoked.
------------------------------------------------------
```

```
CONNECT to consolidated database
begin_connection_autocommit
begin_connection
COMMIT
for each synchronization request with
     the same script version {
  <synchronize>
}
end_connection
COMMIT
DISCONNECT from consolidated database

------------------------------------------------------
synchronize
------------------------------------------------------

<authenticate>
<begin_synchronization>
<upload>
<prepare_for_download>
<download>
<end_synchronization>

------------------------------------------------------
authenticate
------------------------------------------------------

Status <- 1000
UseDefaultAuthentication <- TRUE
if( authenticate_user script is defined ) {
  UseDefaultAuthentication <- FALSE
  TempStatus <- authenticate_user
  if( TempStatus > Status ) {
    Status <- TempStatus
  }
}
```

```
if( authenticate_user_hashed script is defined ) {
  UseDefaultAuthentication <- FALSE
  TempStatus <- authenticate_user_hashed
  if( TempStatus > Status ) {
    Status <- TempStatus
  }
}
if( UseDefaultAuthentication ) {
  if( the user exists in the ml_user table ) {
    if( ml_user.hashed_password column is not NULL ) {
      if( password matches ml_user.hashed_password ) {
        Status <- 1000
      } else {
        Status <- 4000
      }
    } else {
      Status <- 1000
    }
  } else if( -zu+ was on the command line ) {
    Status <- 1000
  } else {
    Status <- 4000
  }
}

if( Status <= 2000 ) {
  if( authenticate_parameters script is defined )
 {
    TempStatus <- authenticate_parameters
    if( TempStatus > Status ) {
      Status <- TempStatus
  }
}
if( Status >= 3000 ) {
  ROLLBACK
  // Abort the synchronization.
} else {
  // UserName defaults to MobiLink user name
  // sent from the remote.
  if( modify_user script is defined ) {
  UserName <- modify_user
  // The new value of UserName is later passed to
  // all scripts that expect the MobiLink user name.
  }
  COMMIT
}
```

```
------------------------------------------------------
begin_synchronization
------------------------------------------------------

begin_synchronization   // conection event
for each table being synchronized {
    begin_synchronization    // call the table level script
}
for each publication being synchronized {
  begin_publication
}
COMMIT

------------------------------------------------------
end_synchronization
------------------------------------------------------

for each publication being synchronized {
  if( begin_publication script was called ) {
    end_publication
  }
}
for each table being synchronized {
  if( begin_synchronization table script was called ) {
    end_synchronization // table event
  }
}
end_synchronization     // connection event

for each table being synchronized {
synchronization_statistics // table event
}
synchronization_statistics // connection event
for each table being synchronized {
time_statistics // table event
}
time_statistics // connection event

COMMIT
```

☞ For the details of upload stream processing, see "Events during upload" on page 328.

☞ For the details of download stream processing, see "Events during download" on page 332.

Events during upload

The following pseudocode illustrates how upload events and upload scripts are invoked.

These events take place at the upload location in the complete event model. For more information, see "Overview of MobiLink events" on page 322.

```
------------------------------------------------------
upload
------------------------------------------------------

begin_upload
  for each table being synchronized {
    begin_upload_rows
    for each uploaded INSERT or UPDATE for this table {
      if( INSERT ) {
        <upload_inserted_row>
      }
      if( UPDATE ) {
        <upload_updated_row>
      }
    }
    end_upload_rows
  }
  for each table being synchronized IN REVERSE ORDER {
    begin_upload_deletes
    for each uploaded DELETE for this table {
      <upload_deleted_row>
    }
    end_upload_deletes
  }
  end_upload

  for each table being synchronized {
    upload_statistics  // table event
  }
    upload_statistics  // connection event

  COMMIT

------------------------------------------------------
<upload_inserted_row>
------------------------------------------------------
// NOTES:
// - Only table scripts for the current table are involved.
// - Cursor-based upload scripts, like upload_cursor,
//   are deprecated, and so are not shown.

UploadUsingStatements <- (
       upload_insert script is defined
    or upload_update script is defined
    or upload_delete script is defined
    or upload_fetch script is defined
    or upload_new_row_insert script is defined
    or upload_old_row_insert script is defined )
```

```
if( UploadUsingStatements ) {
  ConflictsAreExpected <- (
       upload_new_row_insert script is defined
    or upload_old_row_insert script is defined
    or resolve_conflict script is defined )
  if( upload_insert script is defined ) {
    upload_insert
  } else if( ConflictsAreExpected
      and upload_update script is not defined
      and upload_insert script is not defined
      and upload_delete script is not defined ) {
      // Forced conflict.
      upload_new_row_insert
      resolve_conflict
  } else {
      // ignore the insert
  }
} else {
  // ignore the insert
}

------------------------------------------------------
upload_updated_row
------------------------------------------------------
// NOTES:
// - Only table scripts for the current table are involved.
// - Both the old (original) and new rows are uploaded for
//   each update.
// - Cursor-based upload scripts, like upload_cursor,
//   are deprecated, and so are not shown.

UploadUsingStatements <- (
       upload_insert script is defined
    or upload_update script is defined
    or upload_delete script is defined
    or upload_fetch script is defined
    or upload_new_row_insert script is defined
    or upload_old_row_insert script is defined )
```

```
if( UploadUsingStatements ) {
  ConflictsAreExpected <- (
       upload_new_row_insert script is defined
    or upload_old_row_insert script is defined
    or resolve_conflict script is defined )
  Conflicted <- FALSE
  if( upload_update script is defined ) {
    if( ConflictsAreExpected
      and upload_fetch script is defined ) {
      FETCH using upload_fetch INTO current_row
      if( current_row <> old row ) {
        Conflicted <- TRUE
      }
    }
    if( not Conflicted ) {
      upload_update
    }
  } else if( upload_update script is not defined
      and upload_insert script is not defined
      and upload_delete script is not defined ) {
      // Forced conflict.
      Conflicted <- TRUE
  }
  if( ConflictsAreExpected and Conflicted ) {
    upload_old_row_insert
    upload_new_row_insert
    resolve_conflict
  }
} else {
  // ignore the update
}

------------------------------------------------------
upload_deleted_row
------------------------------------------------------
// NOTES:
// - Only table scripts for the current table are involved.
// - Cursor-based upload scripts, like upload_cursor,
//   are deprecated, and so are not shown.

UploadUsingStatements <- (
       upload_insert script is defined
    or upload_update script is defined
    or upload_delete script is defined
    or upload_fetch script is defined
    or upload_new_row_insert script is defined
    or upload_old_row_insert script is defined )
```

```
if( UploadUsingStatements ) {
  ConflictsAreExpected <- (
       upload_new_row_insert script is defined
    or upload_old_row_insert script is defined
    or resolve_conflict script is defined )
  if( upload_delete is defined ) {
    upload_delete
  } else if( ConflictsAreExpected
    and upload_update script is not defined
    and upload_insert script is not defined
    and upload_delete script is not defined ) {
    // Forced conflict.
    upload_old_row_insert
    resolve_conflict
  } else {
    // ignore this delete
  }
} else {
    // ignore this delete
}
```

Events during download

The following pseudocode provides an overview of the sequence in which download events, and hence the script of the same name, are invoked.

These events take place at the download location in the complete event model provided in "Overview of MobiLink events" on page 322.

```
------------------------------------------------------
prepare_for_download
------------------------------------------------------

prepare_for_download
modify_last_download_timestamp
if( prepare_for_download script is defined
    or modify_last_download_timestamp script is defined ) {
    COMMIT
}
```

```
------------------------------------------------------
download
------------------------------------------------------

begin_download
for each table being synchronized {
  begin_download_deletes
  for each row in download_delete_cursor {
    if( all primary key columns are NULL ) {
      send TRUNCATE to remote
    } else {
      send DELETE to remote
    }
  }
  end_download_deletes
  begin_download_rows
  for each row in download_cursor {
    send INSERT ON EXISTING UPDATE to remote
  }
  end_download_rows
}
modify_next_download_timestamp
end_download
for each table being synchronized {
  download_statistics   // table event
}
  download_statistics   // connection event

COMMIT
```

Notes

♦ If an acknowledgement is expected, and if no confirmation of the downloads is received from the client, the entire download transaction is rolled back in the consolidated database.

☞ For more information, see "SendDownloadACK (sa) extended option" [*MobiLink Clients*, page 131] for Adaptive Server Anywhere remotes or "Send Download Acknowledgement synchronization parameter" [*MobiLink Clients*, page 331] for UltraLite remotes.

♦ The download stream does not distinguish between inserts and updates. The script associated with the download_cursor event is a SELECT statement that defines the rows to be downloaded. The client detects whether the row exists or not and carries out the appropriate insert or update operation.

♦ At the end of the download processing, the client automatically deletes rows that violate referential integrity.

☞ For more information, see "Referential integrity and synchronization" on page 22.

authenticate_parameters connection event

Function Receives non-table data from the remote that can be used to authenticate beyond a user ID or password. The non-table data can also be used to arbitrarily customize each synchronization.

Parameters In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Item	Parameter	Description
1	auth_status	INTEGER. This is an INOUT parameter.
2	ml_username	VARCHAR(128).
3...	remote_parameters (one or more)	VARCHAR(128).

Description You can send strings or parameters in the form of strings from both Adaptive Server Anywhere and UltraLite remotes. This allows you to have authentication beyond a user ID and password. It also means that you can customize your synchronization based on the value of parameters, and do this in a pre-synchronization phase, during authentication.

For UltraLite remote databases, pass the parameters using the num_auth_parms and auth_parms fields in the ul_synch_info struct. num_auth_parms is a count of the number of parameters, from 0 to 255. auth_parms is a pointer to a list of strings. To prevent the strings being viewed as plain text, the strings are sent in the same way as passwords. If num_auth_parms is 0, set auth_parms to NULL.

Following is an example of passing parameters in UltraLite:

```
ul_char * Params[ 3 ] = { UL_TEXT( "parm1" ), UL_TEXT( "parm2"
         ), UL_TEXT( "parm3" ) };

...
info.num_auth_parms = 3;
info.auth_parms = Params;
```

For Adaptive Server Anywhere remote databases, you pass parameters using the dbmlsync -ap option, in a comma-separated list. For example,

```
dbmlsync -ap "parm1,parm2,parm3"
```

The MobiLink synchronization server executes this event upon starting each synchronization. It is executed before, and in the same transaction as, the begin_synchronization event.

You can use this event to replace the built-in MobiLink authentication mechanism with a custom mechanism. You may want to call into the authentication mechanism of your DBMS, or you may wish to implement features not present in the MobiLink built-in mechanism.

The number of remote parameters must match the number expected or an error will result. For example, if three parameters are sent, the event must expect five, because there is auth_status and ml_username plus the three parameters. An error will also occur if parameters are sent from the client and there is no event.

If the authenticate_user or authenticate_user_hashed scripts are invoked and return an error, this event is not called.

SQL scripts for the authenticate_parameters event must be implemented as stored procedures.

See also

- "Authenticating MobiLink Users" [*MobiLink Clients,* page 9]
- "Custom user authentication" [*MobiLink Clients,* page 21]
- "authenticate_user connection event" on page 336
- "authenticate_user_hashed connection event" on page 340
- "begin_synchronization connection event" on page 359
- "-ap option" [*MobiLink Clients,* page 100]
- "Authentication Parameters synchronization parameter" [*MobiLink Clients,* page 316]

authenticate_user connection event

Function

Implements custom user authentication.

Parameters

In the following table, the description indicates the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Event parameters are optional only if no subsequent parameters are specified. For example, you must use parameter 1 if you want to use parameter 2.

Item	Parameter	Description
1	auth_status	INTEGER. This is an INOUT parameter.
2	ml_username	VARCHAR(128).
3	user_password	VARCHAR(128).
4	user_new_password	VARCHAR(128).

Default action

Use MobiLink built-in user authentication mechanism.

Description

The MobiLink synchronization server executes this event upon starting each synchronization. It is executed in a transaction before the begin_synchronization transaction.

You can use this event to replace the built-in MobiLink authentication mechanism with a custom mechanism. You may want to call into the authentication mechanism of your DBMS, or you may wish to implement features not present in the MobiLink built-in mechanism, such as password expiry or a minimum password length.

The parameters used in an authenticate_user event are as follows:

1. **auth_status** This required parameter is an INOUT parameter: a parameter that provides a value to the script, and could be given a new value by the script. The auth_status parameter indicates the overall success of the authentication, and can be set to one of the following values:

Returned Value	**Auth_status**	**Description**
V <= 1999	1000	Authentication succeeded.
1999 < V <= 2999	2000	Authentication succeeded: password expiring soon.
2999 < V <= 3999	3000	Authentication failed: password expired.
3999 < V <= 4999	4000	Authentication failed.
4999 < V <= 5999	5000	Authentication failed as user is already synchronizing.
5999 < V	4000	If the returned value is greater than 5999, MobiLink interprets it as a returned value of 4000.

2. **ml_username** This optional parameter is the MobiLink user name.

3. **user_password** This optional parameter indicates the password for authentication purposes. If the user does not supply a password, this is NULL.

4. **user_new_password** This optional parameter indicates a new password. If the user does not change their password, this is NULL.

SQL scripts for the authenticate_user event must be implemented as stored procedures.

When the two authentication scripts are both defined, and both scripts return different auth_status codes, the higher value is used.

The authenticate_user script is executed in a transaction along with all authentication scripts. This transaction always commits.

There are predefined scripts that you can use for the authenticate_user event to simplify authentication using LDAP, IMAP and POP3 servers.

For more information, see “Authenticating to external servers” [*MobiLink Clients,* page 22].

See also

- “Authenticating MobiLink Users” [*MobiLink Clients,* page 9]
- “Custom user authentication” [*MobiLink Clients,* page 21]
- “Authenticating to external servers” [*MobiLink Clients,* page 22]
- “authenticate_user_hashed connection event” on page 340
- “authenticate_parameters connection event” on page 334
- “modify_user connection event” on page 429
- “begin_synchronization connection event” on page 359

SQL example

A typical authenticate_user script is a call to a stored procedure. The order of the parameters in the call must match the order above. The following example uses ml_add_connection_script to assign the event to a stored procedure called my_auth.

```
call ml_add_connection_script(
   'ver1', 'authenticate_user', 'call my_auth ( ?, ?, ?, ? )'
)
```

The following Adaptive Server Anywhere stored procedure uses only the user name to authenticate—it has no password check. The procedure ensures only that the supplied user name is one of the employee IDs listed in the ULEmployee table.

```
CREATE PROCEDURE my_auth(in @user_name varchar(128))
begin
  if exists
  ( select * from ulemployee
    where emp_id = @user_name )
  then
    message 'OK' type info to client;
      return 1000;
    else
    message 'Not OK' type info to client;
      return 4000;
  end if
end
```

Java example

The following stored procedure call registers a Java method called authenticateUser as the script for the authenticate_user event when synchronizing the script version ver1. This syntax is for Adaptive Server Anywhere consolidated databases.

```
call ml_add_java_connection_script(
   'ver1', 'authenticate_user',
   'ExamplePackage.ExampleClass.authenticateUser'
)
```

Following is the sample Java method authenticateUser. It calls Java functions that check and, if needed, change the user's password.

```
public String authenticateUser
( ianywhere.ml.script.InOutInteger authStatus,
  String user, String pwd, String newPwd )
  throws java.sql.SQLException
{  // in a real authenticate_user handler, we would
   // handle more auth code states
   _curUser = user;
   if( checkPwd( user, pwd ) )
   {  // auth successful
      if( newPwd != null )
      {  // pwd is being changed
         if( changePwd( user, pwd, newPwd ) )
         {  // auth ok and pwd change ok. Use custom code
            authStatus.setValue( 1001 );
         }
         else {  // authorization ok but password
                 // change failed. Use custom code.
                 java.lang.System.err.println( "user: "
                 + user + " pwd change failed!" );
                 authStatus.setValue( 1002 ); } }
        else {  authStatus.setValue( 1000 ); } }
        else {  // auth failed
           authStatus.setValue( 4000 ); }
   return( null ); }
```

.NET example

The following stored procedure call registers a .NET method called AuthUser as the script for the authenticate_user connection event when synchronizing the script version ver1. This syntax is for Adaptive Server Anywhere consolidated databases.

```
call ml_add_dnet_connection_script(
   'ver1', 'authenticate_user',
   'TestScripts.Test.AuthUser'
)
```

Following is the C# signature for the method AuthUser.

```
public void AuthUser( ref int authStatus, string user, string
         pwd, string newPwd )
```

☞ For a more detailed example of an authenticate_user script written in C# in .NET, see ".NET synchronization example" on page 300.

authenticate_user_hashed connection event

Function

Implements a custom user authentication mechanism.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Event parameters are optional only if no subsequent parameters are specified. For example, you must use parameter 1 if you want to use parameter 2.

Item	Parameter	Description
1	auth_status	INTEGER. This is an INOUT parameter.
2	ml_username	VARCHAR(128).
3	hashed_user_password	BINARY(20). If the user does not supply a password, this is NULL.
4	hashed_new_password	BINARY(20). If the user does not change their password, this is NULL.

Default action

Use MobiLink built-in user authentication mechanism.

Description

This event is identical to authenticate_user except for the passwords, which are in the same hashed form as those stored in the ml_user.hashed_password column. Passing the passwords in hashed form provides increased security.

A one-way hash is used. A one-way hash takes a password and converts it to a byte sequence that is (essentially) unique to each possible password. The one-way hash lets password authentication take place without having to store the actual password in the consolidated database.

When authenticate_user and authenticate_user_hashed are both defined, and both scripts return different auth_status codes, the higher value is used.

See also

- "Authenticating MobiLink Users" [*MobiLink Clients*, page 9]
- "Custom user authentication" [*MobiLink Clients*, page 21]
- "authenticate_user connection event" on page 336
- "authenticate_parameters connection event" on page 334

SQL example

A typical authenticate_user_hashed script is a call to a stored procedure. The order of the parameters in the call must match the order above. The following example calls ml_add_connection_script to assign the event to a stored procedure called my_auth.

```
call ml_add_connection_script(
   'ver1', 'authenticate_user_hashed', 'call my_auth ( ?, ?, ?)'
)
```

The following Adaptive Server Anywhere stored procedure uses both the user name and password to authenticate. The procedure ensures only that the supplied user name is one of the employee IDs listed in the ULEmployee table. The procedure assumes that the Employee table has a binary(20) column called hashed_pwd.

```
CREATE PROCEDURE my_auth(
  inout @auth_status integer,
  in @user_name varchar(128),
  in @hpwd binary(20) )
begin
  if exists
  ( select * from ulemployee
    where emp_id = @user_name
      and hashed_pwd = @hpwd )
  then
    message 'OK' type info to client;
    return 1000;
  else
    message 'Not OK' type info to client;
    return 4000;
  end if
end
```

Java example

The following stored procedure call registers a Java method called authUserHashed as the script for the authenticate_user_hashed event when synchronizing the script version ver1.

```
call ml_add_java_connection_script(
   'ver1', 'authenticate_user_hashed',
   'ExamplePackage.ExampleClass.authUserHashed')
```

Following is the sample Java method authUserHashed. It calls Java functions that check and, if needed, change the user's password.

```
public String authUserHashed(
   ianywhere.ml.script.InOutInteger authStatus,
   String user, byte pwd[], byte newPwd[] )
  throws java.sql.SQLException
{  // in a real authenticate_user_hashed handler, we
   // would handle more auth code states
   _curUser = user;
   if( checkPwdHashed( user, pwd ) ) {
   // auth successful
     if( newPwd != null )
     {  // pwd is being changed
        if( changePwdHashed( user, pwd, newPwd ) )
        {  // auth ok and pwd change ok use custom code
           authStatus.setValue( 1001 ); }
        else
        { // auth ok but pwd change failed.
          // Use custom code
          java.lang.System.err.println( "user: " + user
            + " pwd change failed!" );
          authStatus.setValue( 1002 ); } }
   else {  authStatus.setValue( 1000 ); } }
   else {  // auth failed
        authStatus.setValue( 4000 ); }
   return( null ); }
```

.NET example

The following stored procedure call registers a .NET method called AuthUserHashed as the script for the authenticate_user_hashed connection event when synchronizing the script version ver1. This syntax is for Adaptive Server Anywhere consolidated databases.

```
call ml_add_dnet_connection_script(
   'ver1',
   'authenticate_user_hashed',
   'TestScripts.Test.AuthUserHashed'
)
```

Following is the C# signature for the call AuthUserHashed.

```
public void AuthUserHashed(
   ref int authStatus,
   string user,
   byte[] pwd,
   byte[] newPwd )
```

begin_connection connection event

Function	Invoked at the time the MobiLink synchronization server connects to the consolidated database server.
Parameters	None.
Default action	None.

Description

The MobiLink synchronization server executes this event upon opening each worker-thread connection to the consolidated database server. The MobiLink synchronization opens connections on demand as synchronization requests come in. When an application forms or reforms a connection with the MobiLink synchronization server, the MobiLink synchronization server temporarily allocates one connection with the database server for the duration of that synchronization.

See also

- “end_connection connection event” on page 384.

SQL example

The following SQL script works with an Adaptive Server Anywhere consolidated database. Two variables are created, one for the last_download timestamp, and one for employee ID.

```
call ml_add_connection_script(
    ’custdb’,
    ’begin_connection’,
    ’create variable @LastDownload timestamp;
    create variable @EmployeeID integer;’)
```

Java example

Note: This script is not generally used in Java, because instead of database variables you would use member variables in this class instance, and prepare the members in the constructor.

The following stored procedure call registers a Java method called beginConnection as the script for the begin_connection event when synchronizing the script version ver1. This syntax is for Adaptive Server Anywhere consolidated databases.

```
call ml_add_java_connection_script(
    ’ver1’,
    ’begin_connection’,
    ’ExamplePackage.ExampleClass.beginConnection’ )
```

Following is the sample Java method beginConnection. This returns SQL that will create a connection level variable.

```
public String beginConnection()
{ return("create variable @LastDownload timestamp;" ); }
```

.NET example

The following stored procedure call registers a .NET method called

BeginConnection as the script for the begin_connection connection event when synchronizing the script version ver1. This syntax is for Adaptive Server Anywhere consolidated databases.

```
call ml_add_dnet_connection_script(
   'ver1',
   'begin_connection',
   'TestScripts.Test.BeginConnection'
)
```

Following is the signature for the call BeginConnection.

```
public string BeginConnection()
{ return("create variable @LastDownload timestamp;" ); }
```

begin_connection_autocommit connection event

Function	Turns on autocommit.
Parameters	None.
Default action	Autocommit is off.

Description

When the MobiLink synchronization server connects to the consolidated database, it turns off autocommit so that it can roll back the upload and download streams if an error occurs.

However, if you are using an Adaptive Server Enterprise consolidated database, you cannot perform DDL functions such as creating temporary tables unless autocommit is on. If you are using an Adaptive Server Enterprise consolidated database, run your DDL commands in the begin_connection_autocommit event. When the event is finished, autocommit is turned off.

Begin_connection_autocommit scripts must be written so that they are repeatable. This is because if an error or deadlock occurs, the MobiLink synchronization server needs to be able to retry the script (since it can't roll it back).

begin_download connection event

Function

Processes any statements just before the MobiLink synchronization server commences preparing the download data stream.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Event parameters are optional only if no subsequent parameters are specified. You must use parameter 1 if you want to use parameter 2.

Item	Parameter	Description
1	last_download	TIMESTAMP
2	ml_username	VARCHAR(128)

Default action

None.

Description

The MobiLink synchronization server executes this event as the first step in the processing of downloaded information. Download information is processed in a single transaction. The execution of this event is the first action in this transaction.

The last_download timestamp is the value obtained from the consolidated database during the last successful synchronization immediately prior to the download phase. If the current user has never synchronized successfully, this value is set to 1900-01-01.

See also

- "end_download connection event" on page 386

SQL example

The following example calls ml_add_connection_script to assign the event to a stored procedure called SetDownloadParameters.

```
call ml_add_connection_script (
   'Lab',
   'begin_download',
   'CALL SetDownloadParameters( ?, ? )' )
```

Java example

The following stored procedure call registers a Java method called beginDownloadConnection as the script for the begin_download connection event when synchronizing the script version ver1.

```
call ml_add_java_connection_script(
   'example_ver',
   'begin_download',
   'ExamplePackage.ExampleClass.beginDownloadConnection' )
```

Following is the sample Java method beginDownloadConnection. It calls a

Java function (prepDeleteTables) that will prepare the delete tables using a JDBC synchronization that was set earlier.

```
public String beginDownloadConnection(
    Timestamp ts, String user )
  throws java.sql.SQLException
{  prepDeleteTables ( _syncConn, ts, user );
   return ( null ); }
```

.NET example

The following stored procedure call registers a .NET method called BeginDownload as the script for the begin_download connection event when synchronizing the script version ver1.

```
call ml_add_dnet_connection_script(
  'ver1',
  'begin_download',
  'TestScripts.Test.BeginDownload'
)
```

Following is the sample .NET method BeginDownload. It calls a .NET function (prepDeleteTables) that will prepare the delete tables using a JDBC synchronization that was set earlier.

```
public void BeginDownload(
  DateTime timestamp,
  string user )
{  prepDeleteTables ( _syncConn, ts, user );
   return ( null ); }
```

begin_download table event

Function

Processes statements related to a specific table just before preparing the download stream of inserts, updates, and deletions.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Event parameters are optional only if no subsequent parameters are specified. For example, you must use parameter 1 if you want to use parameter 2.

Item	Parameter	Description
1	last_download	TIMESTAMP
2	ml_username	VARCHAR(128)
3	table	VARCHAR (128)

Default action

None.

Description

The MobiLink synchronization server executes this event as the first step in preparing download information for a specific table. The download information is prepared in its own transaction. The execution of this event is the first table-specific action in the transaction.

You can have one begin_download script for each table in the remote database. The script is only invoked when that table is synchronized.

The last_download timestamp is the value obtained from the consolidated database during the last successful synchronization immediately prior to the download phase. If the current user has never synchronized successfully, this value is set to 1900-01-01.

See also

♦ "end_download table event" on page 388

SQL example

The following example can be used on an Adaptive Server Anywhere 9 database. The first piece of code calls the ml_add_table_script, and the second creates a BeginTableDownload procedure.

```
call ml_add_table_script(
   'version1',
   'Leads',
   'begin_download',
   'call BeginTableDownLoad( ?, ?, ? ) );

create procedure BeginTableDownload(
   LastDownload timestamp,
   MLUser varchar(128),
   TableName varchar(128) )
begin
  execute immediate 'update ' || TableName ||
' set last_download_check = CURRENT TIMESTAMP
  where Owner = ' ||MLUser;
end
```

Java example

The following stored procedure call registers a Java method called beginDownloadTable as the script for the begin_download table event when synchronizing the script version ver1.

```
call ml_add_java_table_script(
   'ver1',
   'table1',
   'begin_download',
   'ExamplePackage.ExampleClass.beginDownloadTable' )
```

Following is the sample Java method beginDownloadTable. It saves the name of the current table for use in a later member function call.

```
public String beginDownloadTable(
   Timestamp ts,
   String user,
   String table )
{  _curTable = table;
   return( null ); }
```

.NET example

The following stored procedure call registers a .NET method called BeginTableDownload as the script for the begin_download table event when synchronizing the script version ver1 and the table table1.

```
call ml_add_dnet_table_script(
  'ver1', 'table1', 'begin_download',
  'TestScripts.Test.BeginTableDownload'
)
```

Following is the sample .NET method BeginTableDownload. It saves the name of the current table for use in a later member function call.

```
public void BeginTableDownload(
  DateTime timestamp,
  string user,
  string table )
{  _curTable = table;
   return( null ); }
```

begin_download_deletes table event

Function

Processes statements related to a specific table just before fetching a list of rows to be deleted from the specified table in the remote database.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Event parameters are optional only if no subsequent parameters are specified. For example, you must use parameter 1 if you want to use parameter 2.

Item	Parameter	Description
1	last_download	TIMESTAMP
2	ml_username	VARCHAR (128)
3	table	VARCHAR (128)

Default action

None.

Description

This event is executed immediately before fetching a list of rows to be deleted from the named table in the remote database.

You can have one begin_download_deletes script for each table in the remote database.

The last_download timestamp is the value obtained from the consolidated database during the last successful synchronization immediately prior to the download phase. If the current user has never synchronized successfully, this value is set to 1900-01-01.

See also

- "begin_download_rows table event" on page 353
- "end_download_rows table event" on page 393

SQL example

To minimize the amount of data on remotes, you can use this event to flag data that will be deleted when the download_delete_cursor is executed. The following example flags for deletion sales leads from the remote device that are over 10 weeks old. The example can be used on an Adaptive Server Anywhere 9 database. The code calls the ml_add_table_script, and then creates a beginDownloadDeletes procedure.

```
call ml_add_table_script (
  'version1',
  'Leads',
  'begin_download_deletes',
  'call BeginDownloadDeletes (?, ?, ?)' );

create procedure BeginDownloadDeletes(
  LastDownload timestamp,
  MLUser varchar(128),
  TableName varchar(128) )
begin
  execute immediate 'update ' || TableName ||
  ' set delete_flag = 1 where
  days(creation_time, CURRENT DATE) > 70 and Owner = '
  || MLUser;
end;
```

Java example

The following stored procedure call registers a Java method called beginDownloadDeletes as the script for the begin_download_deletes table event when synchronizing the script version ver1.

```
call ml_add_java_table_script(
   'ver1',
   'table1',
   'begin_download_deletes',
   'ExamplePackage.ExampleClass.beginDownloadDeletes' )
```

Following is the sample Java method beginDownloadDeletes. It saves the name of the current table for use in a later member function call.

```
public String beginDownloadDeletes( Timestamp ts,
String user, String table )
{  _curTable = table;
   return( null ); }
```

.NET example

The following stored procedure call registers a .NET method called BeginDownloadDeletes as the script for the begin_download_deletes table event when synchronizing the script version ver1 and the table table1.

```
call ml_add_dnet_table_script(
  'ver1', 'table1', 'begin_download_deletes',
  'TestScripts.Test.BeginDownloadDeletes'
)
```

Following is the sample .NET method BeginDownloadDeletes. It saves the name of the current table for use in a later member function call.

```
public void BeginDownloadDeletes(
  DateTime timestamp,
  string user,
  string table )
{  _curTable = table;
   return( null ); }
```

begin_download_rows table event

Function

Processes statements related to a specific table just before fetching a list of rows to be inserted or updated in the specified table in the remote database.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Event parameters are optional only if no subsequent parameters are specified. For example, you must use parameter 1 if you want to use parameter 2.

Item	**Parameter**	**Description**
1	last_download	TIMESTAMP
2	ml_username	VARCHAR (128)
3	table	VARCHAR (128)

Default action

None.

Description

This event is executed immediately before fetching the stream of rows to be inserted or updated in the named table in the remote database.

You can have one begin_download_rows script for each table in the remote database.

The last_download timestamp is the value obtained from the consolidated database during the last successful synchronization immediately prior to the download phase. If the current user has never synchronized successfully, this value is set to 1900-01-01.

See also

- "begin_download_deletes table event" on page 351
- "end_download_deletes table event" on page 390

SQL example

You can use the begin_download_rows table event to flag rows that you no longer want to download for this table. The following example archives sales leads that are over seven days old.

```
call ml_add_table_script( 'version1', 'Leads',
  'begin_download_rows',
  'call BeginDownloadRows (?, ?, ?)' );

create procedure BeginDownloadRows (
  LastDownload timestamp, MLUser varchar(128),
  TableName varchar(128) )
begin
  execute immediate 'update ' || TableName ||
  ' set download_flag = 0 where
  days(creation_time, CURRENT DATE) > 7 and Owner = '
  || MLUser;
end;
```

Java example

The following stored procedure call registers a Java method called beginDownloadRows as the script for the begin_download_rows table event when synchronizing the script version ver1. This syntax is for Adaptive Server Anywhere consolidated databases.

```
call ml_add_java_table_script(
   'ver1',
   'table1',
   'begin_download_rows',
   'ExamplePackage.ExampleClass.beginDownloadRows' )
```

Following is the sample Java method beginDownloadRows. It generates an UPDATE statement using the table and user. MobiLink will execute this SQL statement.

```
public String beginDownloadRows( Timestamp ts,
String user, String table )
{  return( "update " + table + " set download_flag = 0 "
   + " where days(creation_time, CURRENT DATE) > 7 " +
   " and Owner = '" + user + "'" ); }
```

.NET example

The following stored procedure call registers a .NET method called BeginDownloadRows as the script for the begin_download_rows table event when synchronizing the script version ver1 and the table table1.

```
call ml_add_dnet_table_script(
  'ver1', 'table1', 'begin_download_rows',
  'TestScripts.Test.BeginDownloadRows'
)
```

Following is the sample .NET method BeginDownloadRows. It generates an UPDATE statement using the table and user. MobiLink will execute this SQL statement.

```
public void BeginDownloadRows(
  DateTime timestamp,
  string user,
  string table )
{  return( "update " + table + " set download_flag = 0 "
   + " where days(creation_time, CURRENT DATE) > 7 " +
   " and Owner = '" + user + "'" ); }
```

begin_publication connection event

Function

Provides useful information about the publication(s) being synchronized. This script may also be used to manage generation numbers for file-based downloads.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Item	Parameter	Description
1	generation_number	INTEGER. This is an INOUT parameter. If your deployment does not use file-based downloads, this parameter can be ignored. The default is 1.
2	ml_username	VARCHAR(128). If an UltraLite remote is synchronizing with UL_SYNC_ALL, this event is invoked once with the name 'unknown'.
3	publication_name	VARCHAR(128)
4	last_upload	TIMESTAMP. Last successful upload.
5	last_download	TIMESTAMP. Last successful download.

Default action

The default generation number is 1. If no script is defined for this event, the generation number sent to the remote will always be 1.

Description

This event lets you design synchronization logic based on the publications currently being synchronized. This event is invoked in the same transaction as the begin_synchronization event, and is invoked after the begin_synchronization event. It is invoked once per publication being synchronized.

One potential use for this event is to affect what is downloaded based on the publication used. For example, consider a table that is part of both a priority publication (PriorityPub) and a publication for all tables (AllTablesPub). A script for the begin_publication event could store the publication names in a Java class or a SQL variable or package. Download scripts could then behave differently based on whether the publication being synchronized is PriorityPub or AllTablesPub.

Generation number

The generation_number parameter is specifically for file-based downloads. The output value of the generation number is passed from the begin_synchronization script to the end_synchronization script. The meaning of the generation_number depends on whether the current synchronization is being used to create a download file, or whether the current synchronization has an upload.

The output value of the generation number is passed from the begin_publication script to the end_publication script. The meaning of the generation_number depends on whether the current synchronization is being used to create a download file, or whether the current synchronization has an upload.

In file-based downloads, generation numbers are used to force an upload before the download. The number is stored in the download file. During a synchronization that has an upload, one generation number is output for every subscription to a publication. They are sent to the remote database in the upload acknowledgement, and stored in SYSSYNC.generation_number.

See also

- ♦ "end_publication connection event" on page 395.
- ♦ "File-Based Downloads" on page 85.
- ♦ "MobiLink generation numbers" on page 93.

SQL example

You may want to record the information for each publication being synchronized. The following example calls ml_add_connection_script to assign the event to a stored procedure called RecordPubSync.

```
call ml_add_connection_script(
'version1',
'begin_publication',
'{call RecordPubSync( ?, ?, ?, ?, ? )}' );
```

Java example

The following stored procedure call registers a Java method called beginPublication as the script for the begin_publication connection event when synchronizing the script version ver1.

```
call ml_add_java_connection_script( 'ver1',
   'begin_publication',
   'ExamplePackage.ExampleClass.beginPublication' )
```

Following is the sample Java method beginPublication. It saves the name of each publication for later use.

```
public String beginPublication(
    ianywhere.ml.script.InOutInteger generation_number,
    String user,
    String pub_name,
    Timestamp last_upload,
    Timestamp last_download )
{  _publicationNames[ _numPublications++ ] = pub_name;
   return( null ); }
```

.NET example

The following stored procedure call registers a .NET method called BeginPub as the script for the begin_publication connection event when synchronizing the script version ver1.

```
call ml_add_dnet_connection_script( 'ver1',
  'begin_publication',
  'TestScripts.Test.BeginPub'
)
```

Following is the sample .NET method BeginPub. It saves the name of each publication for later use.

```
public void BeginPub(
    ref int generation_number,
    string user,
    string pub_name,
    DateTime last_upload,
    DateTime last_download )
{
    _publicationNames[ _numPublications++ ] = pub_name;
}
```

begin_synchronization connection event

Function	Processes any statements at the time an application connects to the MobiLink synchronization server in preparation for the synchronization process.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Item	Parameter	Description
1	ml_username	VARCHAR(128)

Default action

None.

Description

The MobiLink synchronization server executes this event immediately after an application preparing to synchronize has formed a connection with the MobiLink synchronization server.

This event is executed within a separate transaction before the upload transaction. It is useful for maintaining statistics.

See also

- "end_synchronization connection event" on page 398
- "begin_synchronization table event" on page 361

SQL example

You may want to store the ml_username value in a temporary table or variable if you will be referencing that value many times in subsequent scripts.

```
Call ml_add_connection_script (
  'version1',
  'begin_synchronization',
  'set @EmployeeID = ?' );
```

Java example

The following stored procedure call registers a Java method called beginSynchronizationConnection as the script for the begin_synchronization connection event when synchronizing the script version ver1.

```
call ml_add_java_connection_script( 'ver1',
  'begin_synchronization',
  'ExamplePackage.ExampleClass.beginSynchronizationConnection'
 )
```

Following is the sample Java method beginSynchronizationConnection. It saves the name of the synchronizing user for later use.

```
public String beginSynchronizationConnection(
  String user )
{  _curUser = user;
   return( null ); }
```

.NET example

The following stored procedure call registers a .NET method called BeginSync as the script for the begin_synchronization connection event when synchronizing the script version ver1.

```
call ml_add_dnet_connection_script( 'ver1',
  'begin_synchronization',
  'TestScripts.Test.BeginSync'
)
```

Following is the sample Java method BeginSync. It saves the name of the synchronizing user for later use.

```
public void BeginSync( string user )
{  _curUser = user;
   return( null ); }
```

begin_synchronization table event

Function

Processes statements related to a specific table at the time an application connects to the MobiLink synchronization server in preparation for the synchronization process.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Event parameters are optional only if no subsequent parameters are specified. You must use parameter 1 if you want to use parameter 2.

Item	Parameter	Description
1	ml_username	VARCHAR (128)
2	table	VARCHAR (128)

Default action

None.

Description

The MobiLink synchronization server executes this event after an application that is preparing to synchronize has formed a connection with the MobiLink synchronization server, and after the begin_synchronization connection-level event.

You can have one begin_synchronization script for each table in the remote database. The event is only invoked when the table is synchronized.

See also

- "end_synchronization table event" on page 400
- "begin_synchronization connection event" on page 359

SQL example

The begin_synchronization table event is used to set up the synchronization of a particular table. The following Adaptive Server Anywhere SQL procedure call registers a script that creates a temporary table for storing rows during synchronization.

```
call ml_add_table_script(
 'ver1',
 'sales_order',
 'begin_synchronization',
 'CREATE TABLE #sales_order
(
   id          integer NOT NULL default autoincrement,
   cust_id       integer NOT NULL,
   order_date       date NOT NULL,
   fin_code_id    char(2) NULL,
   region       char(7) NULL,
   sales_rep       integer NOT NULL,
   PRIMARY KEY (id),
)' )
```

Java example

The following stored procedure call registers a Java method called beginSynchronizationTable as the script for the begin_synchronization table event when synchronizing the script version ver1.

```
call ml_add_java_table_script(
 'ver1',
 'table1',
 'begin_synchronization',
 'ExamplePackage.ExampleClass.beginSynchronizationTable' )
```

Following is the sample Java method beginSynchronizationTable. It adds the current table name to a list of table names contained in this instance.

```
public String beginSynchronizationTable(String user,
String table )
{  _tableList.add( table );
   return( null ); }
```

.NET example

The following stored procedure call registers a .NET method called BeginTableSync as the script for the begin_synchronization table event when synchronizing the script version ver1 and the table table1.

```
call ml_add_dnet_table_script(
 'ver1', 'table1',
  'begin_synchronization',
  'TestScripts.Test.BeginTableSync'
             )
```

Following is the sample NET method BeginTableSync. It adds the current table name to a list of table names contained in this instance.

```
public void BeginTableSync( string user, string table )
{  _tableList.Add( table );
   return( null ); }
```

begin_upload connection event

Function

Processes any statements just before the MobiLink synchronization server commences processing the stream of uploaded inserts, updates, and deletes.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Item	Parameter	Description
1	ml_username	VARCHAR (128)

Default action

None.

Description

The MobiLink synchronization server executes this event as the first step in the processing of uploaded information. Upload information is processed in a single transaction. The execution of this event is the first action in this transaction.

See also

- "end_upload connection event" on page 402
- "begin_upload table event" on page 365

SQL example

The begin_upload connection event is used to perform whatever steps you need performed prior to uploading rows. The following Adaptive Server Anywhere SQL script creates a temporary table for storing old and new row values for conflict processing of the sales_order table.

```
call ml_add_connection_script(
 'version1',
 'begin_upload',
 'CREATE TABLE #sales_order_conflicts
(
   id          integer NOT NULL default autoincrement,
   cust_id       integer NOT NULL,
   order_date       date NOT NULL,
   fin_code_id    char(2) NULL,
   region       char(7) NULL,
   sales_rep       integer NOT NULL,
   new_value       char(1) NOT NULL,
   PRIMARY KEY (id),
)' )
```

Java example

The following stored procedure call registers a Java method called beginUploadConnection as the script for the begin_upload connection event when synchronizing the script version ver1.

```
call ml_add_java_connection_script( 'ver1', 'begin_upload',
'ExamplePackage.ExampleClass.beginUploadConnection ' )
```

Following is the sample Java method beginUploadConnection. It prints a message to the MobiLink output log. (This might be useful at development time but would slow down a production server.)

```
public String beginUploadConnection( String user )
{  java.lang.System.out.println(
     "Starting upload for user: " + user );
   return( null ); }
```

.NET example

The following stored procedure call registers a .NET method called BeginUpload as the script for the begin_upload connection event when synchronizing the script version ver1.

```
call ml_add_dnet_connection_script(
  'ver1',
  'begin_upload',
  'TestScripts.Test.BeginUpload'
)
```

Following is the C# signature for the call BeginUpload.

```
public void BeginUpload( string user )
```

The following C# example saves the current user name for use in a later event.

```
public void BeginUpload( string curUser )
{
    user = curUser;
}
```

begin_upload table event

Function

Processes statements related to a specific table just before the MobiLink synchronization server commences processing the stream of uploaded inserts, updates, and deletes.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Event parameters are optional only if no subsequent parameters are specified. You must use parameter 1 if you want to use parameter 2.

Item	Parameter	Description
1	ml_username	VARCHAR(128)
2	table	VARCHAR(128)

Default action

None.

Description

The MobiLink synchronization server executes this event as the first step in the processing of uploaded information. Upload information is processed in a separate transaction. The execution of this event is the first table-specific action in this transaction.

You can have one begin_upload script for each table in the remote database. The script is only invoked when the table is actually synchronized.

See also

- "end_upload table event" on page 404
- "begin_upload connection event" on page 363

SQL example

When uploading rows from a remote you may want to place the changes in an intermediate table and manually process changes yourself. You can populate a global temporary table in this event.

```
call ml_add_table_script(
  'version1',
  'Leads',
  'begin_upload',
  'insert into T_Leads SELECT *
FROM Leads WHERE Owner = @EmployeeID')
```

Java example

The following stored procedure call registers a Java method called beginUploadTable as the script for the begin_upload table event when synchronizing the script version ver1.

```
call ml_add_java_table_script(
  'ver1',
  'table1',
  'begin_upload',
  'ExamplePackage.ExampleClass.beginUploadTable'
 )
```

Following is the sample Java method beginUploadTable. It prints a message to the MobiLink output log. (This might be useful at development time but would slow down a production server.)

```
public String beginUploadTable( String user,
String table )
{  java.lang.System.out.println("Beginning to process upload
         for: " + table);
return( null ); }
```

.NET example

The following stored procedure call registers a .NET method called BeginTableUpload as the script for the begin_upload table event when synchronizing the script version ver1 and the table table1.

```
call ml_add_dnet_table_script(
  'ver1', 'table1', 'begin_upload',
  'TestScripts.Test.BeginTableUpload'
)
```

Following is the sample .NET method BeginTableUpload. It prints a message to the MobiLink output log. (This might be useful at development time but would slow down a production server.)

```
public void BeginTableUpload(
  string user,
  string table )
{  System.Console.WriteLine("Beginning to process upload for: "
         + table);
return( null ); }
```

begin_upload_deletes table event

Function

Processes statements related to a specific table just before uploading deleted rows from the specified table in the remote database.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Event parameters are optional only if no subsequent parameters are specified. You must use parameter 1 if you want to use parameter 2.

Item	Parameter	Description
1	ml_username	VARCHAR(128)
2	table	VARCHAR(128)

Default action

None.

Description

This event runs immediately before applying the changes that result from rows deleted in the client table named in the second parameter.

You can have one begin_upload_deletes script for each table in the remote database. The script is only invoked when the table is actually synchronized.

See also

- "end_upload_deletes table event" on page 407

SQL example

The begin_upload_deletes connection event is used to perform whatever steps you need performed after uploading inserts and updates for a particular table, but before deletes are uploaded for that table. The following Adaptive Server Anywhere SQL script creates a temporary table for storing deletes temporarily during upload:

```
call ml_add_table_script(
 'ver1',
 'sales_order',
 'begin_upload_deletes',
 'CREATE TABLE #sales_order_deletes
(
   id         integer NOT NULL default autoincrement,
   cust_id      integer NOT NULL,
   order_date      date NOT NULL,
   fin_code_id   char(2) NULL,
   region      char(7) NULL,
   sales_rep      integer NOT NULL,
   PRIMARY KEY (id),
)' )
```

Java example

The following stored procedure call registers a Java method called

beginUploadDeletes as the script for the begin_upload_deletes table event when synchronizing the script version ver1.

```
call ml_add_java_table_script( 'ver1', 'table1',
'begin_upload_deletes',
'ExamplePackage.ExampleClass. beginUploadDeletes' )
```

Following is the sample Java method beginUploadDeletes. It prints a message to the MobiLink output log. (This might be useful at development time but would slow down a production server.)

```
public String beginUploadDeletes(
   String user,
   String table )
 throws java.sql.SQLException
{  java.lang.System.out.println( "Starting upload
   deletes for table: " + table );
    return( null ); }
```

.NET example

The following stored procedure call registers a .NET method called BeginUploadDeletes as the script for the begin_upload_deletes table event when synchronizing the script version ver1 and the table table1.

```
call ml_add_dnet_table_script( 'ver1', 'table1',
  'begin_upload_deletes',
  'TestScripts.Test.BeginUploadDeletes'
)
```

Following is the sample .NET method BeginUploadDeletes. It prints a message to the MobiLink output log. (This might be useful at development time but would slow down a production server.)

```
public void BeginUploadDeletes( string user,
  string table )
{  System.Console.WriteLine( "Starting upload
   deletes for table: " + table );
    return( null ); }
```

begin_upload_rows table event

Function

Processes statements related to a specific table just before uploading inserts and updates from the specified table in the remote database.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Event parameters are optional only if no subsequent parameters are specified. You must use parameter 1 if you want to use parameter 2.

Item	Parameter	Description
1	ml_username	VARCHAR(128)
2	table	VARCHAR(128)

Default action

None.

Description

This event is run immediately prior to applying the changes that result from inserts and deletes to the client table named in the second parameter.

You can have one begin_upload_rows script for each table in the remote database. The script is only invoked when the table is actually synchronized.

See also

- "end_upload_rows table event" on page 409

SQL example

The begin_upload_rows connection event is used to perform whatever steps you need performed before uploading inserts and updates for a particular table. The following script calls a stored procedure that prepares the consolidated database for inserts and updates into the Inventory table:

```
call ml_add_table_script(
 'MyCorp 1.0',
 'Inventory',
 'begin_upload_rows',
 'call PrepareForUpserts()' )
```

Java example

The following stored procedure call registers a Java method called beginUploadRows as the script for the begin_upload_rows table event when synchronizing the script version ver1.

```
call ml_add_java_table_script(
  'ver1',
  'table1',
  'begin_upload_rows',
  'ExamplePackage.ExampleClass.beginUploadRows' )
```

Following is the sample Java method beginUploadRows. It prints a message to the MobiLink output log. (This might be useful at development time but would slow down a production server.)

```
public String beginUploadRows( String user,
String table )
  throws java.sql.SQLException
{  java.lang.System.out.println( "Starting upload rows
   for table: " + table + " and user: " + user );
   return( null ); }
```

.NET example

The following stored procedure call registers a .NET method called BeginUploadRows as the script for the begin_upload_rows table event when synchronizing the script version ver1 and the table table1.

```
call ml_add_dnet_table_script(
  'ver1', 'table1', 'begin_upload_rows',
  'TestScripts.Test.BeginUploadRows'
)
```

Following is the sample .NET method BeginUploadRows. It prints a message to the MobiLink output log. (This might be useful at development time but would slow down a production server.)

```
public void BeginUploadRows(
  string user,
  string table )
{  System.Console.WriteLine( "Starting upload rows
   for table: " + table + " and user: " + user );
   return( null ); }
```

download_cursor table event

Function

Defines a cursor to select rows that are to be downloaded and inserted or updated in the remote database.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Event parameters are optional only if no subsequent parameters are specified. You must use parameter 1 if you want to use parameter 2.

Item	Parameter	Description
1	last_download	TIMESTAMP
2	ml_username	VARCHAR(128)

Default action

None.

A default download_cursor SQL script can be generated using the MobiLink synchronization server -za option. Also, the UltraLite analyzer generates a SELECT statement based on your reference database that you can use to get started.

Description

The MobiLink synchronization server opens a read-only cursor with which to fetch a list of rows to download to the remote database. This script should contain a suitable SELECT statement.

The parameters are the last_download timestamp and the user name. You can use these values if you choose by placing question marks in your SQL statement.

You can have one download_cursor script for each table in the remote database.

The last_download timestamp is the value obtained from the consolidated database during the last successful synchronization immediately prior to the download phase. If the current user has never synchronized successfully, this value is set to 1900-01-01.

To optimize performance of the download stage of synchronization to UltraLite clients, when the range of primary key values is outside the current rows on the device, you should order the rows in the download cursor by primary key. Downloads of large reference tables, for example, can benefit from this optimization.

Each download_cursor script must contain a SELECT statement or a call to a procedure that contains a SELECT statement. The MobiLink synchronization server uses this statement to define a cursor in the consolidated database.

The script must select all columns that correspond to the columns in the corresponding table in the remote database. The columns in the consolidated database can have different names than the corresponding columns in the remote database, but they must be of compatible types.

The columns must be selected in the order that the corresponding columns are defined in the remote database. This order is identical to the order of the columns in the reference database.

Note that download_cursor allows for cascading deletes. Thus, you can delete records from a database.

For Java and .NET applications, this script must return valid SQL.

See also

- "Writing scripts to download rows" on page 246
- "Writing download_cursor scripts" on page 247
- "Partitioning rows among remote databases" on page 52
- "download_delete_cursor table event" on page 375

SQL example

The following example comes from an Oracle installation, although the statement is valid against all supported databases. The example downloads all rows that have been changed since the last time the user downloaded data, and which match the user name in the emp_name column.

```
call ml_add_table_script(
  'Lab',
  'ULOrder',
  'download_cursor',
  'SELECT order_id, cust_id, prod_id, emp_id,
      disc, quant, notes, status
       FROM ULOrder
       WHERE last_modified >= ? AND emp_name = ?' )
```

To write a download_cursor SQL script that does not use the first parameter (the last_download timestamp), but does use the second parameter (the MobiLink user name), add a dummy clause that affects no rows. For example:

```
call ml_add_table_script(
   'Lab',
   'ULOrder',
   'download_cursor',
   'SELECT order_id, cust_id, prod_id, emp_id, disc,
       quant, notes, status
        FROM ULOrder WHERE ? IS NOT NULL AND emp_name = ?' )
```

You must still use both parameters, but the first ? is a place holder that does nothing.

Java example

The following stored procedure call registers a Java method called downloadCursor as the script for the download_cursor table event when synchronizing the script version ver1.

```
call ml_add_java_table_script(
   'ver1',
   'ULCustomer',
   'download_cursor',
   'ExamplePackage.ExampleClass.downloadCursor ' )
```

Following is the sample Java method downloadCursor. It returns an SQL statement to download rows where the last_modified column is greater than or equal to the last download time.

```
public String downloadCursor(java.sql.Timestamp ts,String user )
 {
  return("SELECT cust_id, cust_name FROM ULCustomer where last_
         modified >= ' " + ts + " ' ");
 }
```

.NET example

The following stored procedure call registers a .NET method called DownloadCursor as the script for the download_cursor table event when synchronizing the script version ver1 and the table table1.

```
call ml_add_dnet_table_script(
  'ver1', 'table1', 'download_cursor',
  'TestScripts.Test.DownloadCursor'
)
```

Following is the sample .NET method DownloadCursor. It populates a temporary table with the contents of a file called *rows.txt*. It then returns a cursor that causes MobiLink to send the rows in the temporary table to the remote database. This syntax is for Adaptive Server Anywhere consolidated databases.

```
public string DownloadCursor(
   DateTime ts,
   string user )
{
  DBCommand   stmt   = curConn.CreateCommand();
  StreamReader  input  = new StreamReader( "rows.txt" );
  string        sql    = input.ReadLine();

  stmt.CommandText = "DELETE FROM dnet_dl_temp";
  stmt.ExecuteNonQuery();

  while( sql != null ){
   stmt.CommandText = "INSERT INTO dnet_dl_temp VALUES " + sql;
   stmt.ExecuteNonQuery();
   sql = input.ReadLine();
  }
  return( "SELECT * FROM dnet_dl_temp" );
}
```

download_delete_cursor table event

Function Defines a cursor to select rows that are to be deleted in the remote database.

Parameters In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Event parameters are optional only if no subsequent parameters are specified. You must use parameter 1 if you want to use parameter 2.

Item	Parameter	Description
1	last_download	TIMESTAMP
2	ml_username	VARCHAR(128)

Default action None.

Description The MobiLink synchronization server opens a read-only cursor with which to fetch a list of rows to download, and then insert or update in the remote database. This script must contain a SELECT statement that returns the primary key values of the rows to be deleted from the table in the remote database.

The parameters are the last_download timestamp and the user name. You can use these values by placing a question mark in your SQL statement.

You can have one download_delete_cursor script for each table in the remote database.

If the download_delete_cursor has NULLs for the primary key columns for one or more rows in a table, then MobiLink tells the remote to delete all the data in the table. For a complete description of this behavior, see "Deleting all the rows in a table" on page 249.

The last_download timestamp is the value obtained from the consolidated database during the last successful synchronization immediately prior to the download phase. If the current user has never synchronized successfully, this value is set to 1900-01-01.

Note that rows deleted from the consolidated database will not appear in a result set defined by a download_delete_cursor event, and so are not automatically deleted from the remote database. One technique for identifying rows to be deleted from remote databases is to add a column to the consolidated database table identifying a row as inactive.

For Java and .NET applications, this script must return valid SQL.

See also

- ♦ “download_cursor table event” on page 371
- ♦ “Writing scripts to download rows” on page 246
- ♦ “Partitioning rows among remote databases” on page 52
- ♦ “Writing download_delete_cursor scripts” on page 248

SQL example

This example is taken from the Contact sample and can be found in *Samples\MobiLink\Contact\build_consol.sql*. It deletes from the remote database any customers who have been changed since the last time this user downloaded data (`Customer.last_modified >= ?`), and either

- ♦ do not belong to the synchronizing user (`SalesRep.ml_username != ?`), or
- ♦ are marked as inactive in the consolidated database (`Customer.active = 0`).

```
call ml_add_table_script(
   'ver1',
   'table1',
   'download_delete_cursor',
'SELECT cust_id FROM Customer key join SalesRep
WHERE Customer.last_modified >= ? AND
( SalesRep.ml_username != ? OR Customer.active = 0 )')
```

Java example

The following stored procedure call registers a Java method called downloadDeleteCursor as the script for the download_delete_cursor event when synchronizing the script version ver1.

```
call ml_add_java_table_script(
   'ver1',
   'table1',
   'download_delete_cursor',
   'ExamplePackage.ExampleClass.downloadDeleteCursor' )
```

Following is the sample Java method downloadDeleteCursor. It calls a Java method that generates the SQL for the download delete cursor.

```
public String downloadDeleteCursor( Timestamp ts,
String user )
{  return( getDownloadCursor( _curUser, _curTable ) ); }
```

.NET example

The following stored procedure call registers a .NET method called DownloadDeleteCursor as the script for the download_delete_cursor table event when synchronizing the script version ver1 and the table table1.

```
call ml_add_dnet_table_script(
  'ver1',
  'table1',
  'download_delete_cursor',
  'TestScripts.Test.DownloadDeleteCursor'
)
```

Following is the sample .NET method DownloadDeleteCursor. It calls a .NET method that generates the SQL for the download delete cursor.

```
public string DownloadDeleteCursor(
  DateTime timestamp,
  string user )
{  return( getDownloadCursor( _curUser, _curTable ) ); }
```

download_statistics connection event

Function

Tracks synchronization statistics for download operations.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See “SQL-Java data types” on page 260 and “SQL-.NET data types” on page 287.

Event parameters are optional only if no subsequent parameters are specified. For example, you must use parameter 1 if you want to use parameter 2.

Item	Parameter	Description
1	ml_username	VARCHAR(128). The MobiLink user name as specified in your SYNCHRO-NIZATION USER definition.
2	warnings	INTEGER. The number of warnings issued.
3	errors	INTEGER. The number of errors, including handled errors, that occurred.
4	fetched_rows	INTEGER. The number of rows fetched by the download_cursor script.
5	deleted_rows	INTEGER. The number of rows fetched by the download_deletes script.
6	filtered_rows	INTEGER. The number of rows from (5) actually sent to the remote. This reflects download filtering of uploaded values.
7	bytes	INTEGER. The number of bytes sent to the remote as the download.

Default action

None.

Description

The download_statistics event allows you to gather, for any user, statistics on downloads. The download_statistics connection script is called just prior to the commit at the end of the download transaction.

Note:
Depending on the command line, not all warnings or errors are logged, so the warnings and errors counts may be more than the number of warnings or errors logged.

See also

- ♦ "download_statistics table event" on page 381
- ♦ "upload_statistics connection event" on page 469
- ♦ "upload_statistics table event" on page 472
- ♦ "synchronization_statistics connection event" on page 445
- ♦ "synchronization_statistics table event" on page 448
- ♦ "time_statistics connection event" on page 450
- ♦ "time_statistics table event" on page 453
- ♦ "MobiLink Monitor" on page 117

SQL example

The following example inserts synchronization statistics into a table called download_audit.

```
call ml_add_connection_script(
  'ver1', 'download_statistics',
'INSERT INTO download_audit(
  user_name, warnings, errors, deleted_rows,
  fetched_rows, download_rows, bytes )
VALUES (?,?,?,?,?,?,?)')
```

Once vital statistics are inserted into the audit table, you may use these statistics to monitor your synchronizations and make optimizations where applicable.

Java example

The following stored procedure call registers a Java method called downloadStatisticsConnection as the script for the download_statistics event when synchronizing the script version ver1.

```
call ml_add_java_connection_script(
  'ver1',
  'download_statistics',
  'ExamplePackage.ExampleClass.downloadStatisticsConnection' )
```

Following is the sample Java method downloadStatisticsConnection. It prints the number of fetched rows to the MobiLink output log. (This might be useful at development time but would slow down a production server.)

```
public String downloadStatisticsConnection(
  String user,
  int warnings,
  int errors,
  int fetchedRows,
  int deletedRows,
  int bytes )
{  java.lang.System.out.println( "download connection
   stats fetchedRows: " + fetchedRows );
   return( null ); }
```

.NET example

The following stored procedure call registers a .NET method called DownloadStats as the script for the download_statistics connection event when synchronizing the script version ver1.

```
call ml_add_dnet_connection_script(
  'ver1',
  'download_statistics',
  'TestScripts.Test.DownloadStats'
)
```

Following is the sample .NET method DownloadStats. It prints the number of fetched rows to the MobiLink output log. (This might be useful at development time but would slow down a production server.)

```
public void DownloadStats(
  string user,
  int warnings,
  int errors,
  int deletedRows,
  int fetchedRows,
  int downloadRows,
  int bytes )
{  System.Console.WriteLine( "download connection
   stats fetchedRows: " + fetchedRows );
   return( null ); }
```

download_statistics table event

Function

Tracks synchronization statistics for download operations by table.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Event parameters are optional only if no subsequent parameters are specified. For example, you must use parameter 1 if you want to use parameter 2.

Item	Parameter	Description
1	ml_username	VARCHAR(128). This is the MobiLink user name as specified in your SYNCHRONIZATION USER definition.
2	table	VARCHAR(128). The table name.
3	warnings	INTEGER. The number of warnings issued.
4	errors	INTEGER. The number of errors, including handled errors, that occurred.
5	fetched_rows	INTEGER. The number of rows fetched by the download_cursor script.
6	deleted_rows	INTEGER. The number of rows fetched by the download_deletes script.
7	filtered_rows	INTEGER. The number of rows from (6) actually sent to the remote. This reflects download filtering of uploaded values.
8	bytes	INTEGER. The number of bytes sent to the remote as the download.

Default action

None.

Description

The download_statistics event allows you to gather, for any user and table, statistics on downloads as they apply to that table. The download_statistics table script is called just prior to the commit at the end of the download transaction.

See also

- "download_statistics connection event" on page 378
- "upload_statistics connection event" on page 469

- “upload_statistics table event” on page 472
- “synchronization_statistics connection event” on page 445
- “synchronization_statistics table event” on page 448
- “time_statistics connection event” on page 450
- “time_statistics table event” on page 453
- “MobiLink Monitor” on page 117

SQL example

The following example inserts synchronization statistics into a table called download_audit. Once vital statistics are inserted into the audit table, you may use these statistics to monitor your synchronizations and make optimizations where applicable.

```
call ml_add_java_table_script(
  'ver1',
  'table1',
  'download_statistics',
'INSERT INTO download_audit (
  user_name, table, warnings, errors,
  deleted_rows, fetched_rows, download_rows, bytes)
VALUES (?,?,?,?,?,?,?,?)')
```

Java example

The following stored procedure call registers a Java method called downloadStatisticsTable as the script for the download_statistics table event when synchronizing the script version ver1.

```
call ml_add_java_table_script(
  'ver1',
  'table1',
  'download_statistics',
  'ExamplePackage.ExampleClass.downloadStatisticsTable' )
```

Following is the sample Java method downloadStatisticsTable. It prints some statistics for this table to the MobiLink output log. (This might be useful at development time but would slow down a production server.)

```
public String downloadStatisticsTable(
  String user,
  String table,
  int warnings,
  int errors,
  int fetchedRows,
  int deletedRows,
  int bytes )
{  java.lang.System.out.println( "download table stats "
   + "table: " + table + "bytes: " + bytes );
   return( null ); }
```

.NET example

The following stored procedure call registers a .NET method called DownloadTableStats as the script for the download_statistics table event when synchronizing the script version ver1 and the table table1.

```
call ml_add_dnet_table_script(
  'ver1',
  'table1',
  'download_statistics',
  'TestScripts.Test.DownloadTableStats'
)
```

Following is the sample .NET method DownloadTableStats. It prints some statistics for this table to the MobiLink output log. (This might be useful at development time but would slow down a production server.)

```
public void DownloadTableStats(
  string user,
  string table,
  int warnings,
  int errors,
  int deletedRows,
  int fetchedRows,
  int downloadRows,
  int bytes )
{  System.Console.WriteLine( "download table stats "
   + "table: " + table + "bytes: " + bytes );
   return( null ); }
```

end_connection connection event

Function

Processes any statements just before the MobiLink synchronization server closes a connection with the consolidated database server, either in preparation to shut down or when a connection is removed from the connection pool.

This script is normally used to complete any actions started by the begin_connection script and free any resources acquired by it.

Parameters

None.

Default action

None.

Description

You can use the end_connection script to perform an action of your choice just prior to closing of a connection between the MobiLink synchronization server and the consolidated database server.

See also

- "begin_connection connection event" on page 343

SQL example

The following Adaptive Server Anywhere SQL script drops a temporary table that was created by the begin_connection script. Strictly speaking, this table doesn't need to be dropped explicitly, since ASA will do this automatically when the connection is destroyed. Whether or not a temporary table needs to be dropped explicitly depends on your consolidated database type.

```
call ml_add_connection_script(
 'version 1.0',
 'end_connection',
 'drop table #sync_info' )
```

Java example

The following stored procedure call registers a Java method called endConnection as the script for the end_connection event when synchronizing the script version ver1.

```
call ml_add_java_connection_script(
   'ver1',
   'end_connection',
   'ExamplePackage.ExampleClass.endConnection' )
```

Following is the sample Java method endConnection. It prints a message to the MobiLink output log. (This might be useful at development time but would slow down a production server.)

```
public String endConnection()
{  java.lang.System.out.println( "ending connection" );
   return( null ); }
```

.NET example

The following stored procedure call registers a .NET method called

EndConnection as the script for the end_connection connection event when synchronizing the script version ver1. This syntax is for Adaptive Server Anywhere consolidated databases.

```
call ml_add_dnet_connection_script(
  'ver1',
  'end_connection',
  'TestScripts.Test.EndConnection'
)
```

Following is the sample .NET method EndConnection. It prints a message to the MobiLink output log. (This might be useful at development time but would slow down a production server.)

```
public void EndConnection()
{  System.Console.WriteLine( "ending connection" );
   return( null ); }
```

end_download connection event

Function

Processes any statements just after the MobiLink synchronization server concludes preparation of the download data.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Event parameters are optional only if no subsequent parameters are specified. You must use parameter 1 if you want to use parameter 2.

Item	Parameter	Description
1	last_download	TIMESTAMP
2	ml_username	VARCHAR(128)

Default action

None.

Description

The MobiLink synchronization server executes this script after all rows have been downloaded and, if expecting a download acknowledgement, confirmation of receipt has been received. Download information is processed in a single transaction. The execution of this script is the last non statistical action in this transaction.

The last_download timestamp is the value obtained from the consolidated database during the last successful synchronization immediately prior to the download phase. If the current user has never synchronized successfully, this value is set to 1900-01-01.

See also

- "begin_download connection event" on page 346

SQL example

The following example shows one possible use of an end_download connection script. Each row in the ULEmpCust table has an action column. The following script uses the value in this column to delete records from the remote database.

```
all ml_add_java_connection_script(
  'ver1',
  'end_download',
'DELETE FROM ULEmpCust ec
  WHERE ? IS NOT NULL
    AND ec.emp_id = ? AND action = ''D''')
```

Java example

The following stored procedure call registers a Java method called endDownloadConnection as the script for the end_download connection event when synchronizing the script version ver1.

```
call ml_add_java_connection_script(
  'ver1',
  'end_download',
  'ExamplePackage.ExampleClass.endDownloadConnection' )
```

Following is the sample Java method endDownloadConnection. Each row in the ULEmpCust table has an action column. The following script uses the value in this column to delete records from the remote database. It also uses the current MobiLink connection (saved earlier) to perform an update before the download ends. This syntax is for Adaptive Server Anywhere consolidated databases.

```
public String endDownloadConnection(
    Timestamp ts,
    String user )
  throws java.sql.SQLException
{  String del_sql =    "DELETE FROM ULEmpCust ec " +
   "WHERE ec.emp_id = '" + user + "' " +
   "AND action = 'D' ";
   execUpdate( _syncConn, del_sql );
   return( null );
}
```

.NET example

The following stored procedure call registers a .NET method called EndDownload as the script for the end_download connection event when synchronizing the script version ver1.

```
call ml_add_dnet_connection_script(
  'ver1',
  'end_download',
  'TestScripts.Test.EndDownload' )
```

Following is the sample .NET method EndDownload. Each row in the ULEmpCust table has an action column. The following script uses uses the value in this column to delete records from the remote database. It also uses the current MobiLink connection (saved earlier) to perform an update before the download ends. This syntax is for Adaptive Server Anywhere consolidated databases.

```
public void EndDownload(
  DateTime timestamp,
  string user )
{  string del_sql =    "DELETE FROM ULEmpCust ec " +
   "WHERE ec.emp_id = '" + user + "' " +
   "AND action = 'D' ";
   execUpdate( _syncConn, del_sql );
   return( null );
}
```

end_download table event

Function

Processes statements related to a specific table just after the MobiLink synchronization server concludes preparing the stream of downloaded inserts, updates, and deletes.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Event parameters are optional only if no subsequent parameters are specified. For example, you must use parameter 1 if you want to use parameter 2.

Item	Parameter	Description
1	last_download	TIMESTAMP
2	ml_username	VARCHAR(128)
3	table	VARCHAR(128)

Default action

None.

Description

The MobiLink synchronization server executes this script after all rows have been downloaded and confirmation of receipt has been received. The download information is prepared in a separate transaction. The execution of this script is the last table-specific, non-statistical action in this transaction.

You can have one end_download script for each table in the remote database.

The last_download timestamp is the value obtained from the consolidated database during the last successful synchronization immediately prior to the download phase. If the current user has never synchronized successfully, this value is set to 1900-01-01.

See also

- "begin_download table event" on page 348
- "end_download connection event" on page 386

SQL example

The end_download table event is used to perform whatever steps you need performed after downloading a particular table. The following Adaptive Server Anywhere SQL script drops a temporary table created by a prepare_for_download script to hold download rows for the sales_summary table.

```
call ml_add_table_script(
 'MyCorp 1.0',
 'sales_summary',
 'end_download',
 'drop table #sales_summary_download' )
```

Java example

The following stored procedure call registers a Java method called endDownloadTable as the script for the end_download table event when synchronizing the script version ver1..

```
call ml_add_java_table_script (
  'ver1',
  'table1',
  'end_download',
  'ExamplePackage.ExampleClass.endDownloadTable' )
```

Following is the sample Java method endDownloadTable. It resets the current table member variable.

```
public String endDownloadTable( Timestamp ts,
String user, String table )
{  _curTable = null;
   return( null ); }
```

.NET example

The following stored procedure call registers a .NET method called EndTableDownload as the script for the end_download table event when synchronizing the script version ver1 and the table table1.

```
call ml_add_dnet_table_script(
  'ver1',
  'table1',
  'end_download',
  'TestScripts.Test.EndTableDownload'
)
```

Following is the sample .NET method EndTableDownload. It resets the current table member variable.

```
public void EndTableDownload
  DateTime timestamp,
  string user,
  string table )
{  _curTable = null;
   return( null ); }
```

end_download_deletes table event

Function

Processes statements related to a specific table just after preparing a list of rows to be deleted from the specified table in the remote database.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Event parameters are optional only if no subsequent parameters are specified. For example, you must use parameter 1 if you want to use parameter 2.

Item	Parameter	Description
1	last_download	TIMESTAMP
2	ml_username	VARCHAR(128)
3	table	VARCHAR(128)

Default action

None.

Description

This script is executed immediately after preparing a list of rows to be deleted from the named table in the remote database.

You can have one end_download_deletes script for each table in the remote database.

The last_download timestamp is the value obtained from the consolidated database during the last successful synchronization immediately prior to the download phase. If the current user has never synchronized successfully, this value is set to 1900-01-01.

See also

- "begin_download_deletes table event" on page 351
- "end_download connection event" on page 386
- "begin_download_rows table event" on page 353
- "end_download_rows table event" on page 393
- "download_delete_cursor table event" on page 375

SQL example

You may want to mark a row as deleted on the remote database. The following script updates a column in the consolidated database called OnRemote. *Note:* The WHERE clause on the UPDATE matches the WHERE clause used for your download_delete_cursor event script.

```
Call ml_add_table_script(
  'version1',
  'Leads',
  'end_download_deletes',
  'UPDATE Leads SET OnRemote = 0
      WHERE LastModified >= ?
        AND Owner = ? AND DeleteFlag=1');
```

Java example

The following stored procedure call registers a Java method called endDownloadDeletes as the script for the end_download_deletes table event when synchronizing the script version ver1.

```
call ml_add_java_table_script(
  'ver1',
  'table1',
  'end_download_deletes',
  'ExamplePackage.ExampleClass.endDownloadDeletes' )
```

You may want to mark a row as deleted on the remote database. Following is the sample Java method endDownloadDeletes. It updates a column in the consolidated database called OnRemote to indicate the record no longer resides on the remote database. *Note:* The WHERE clause on the UPDATE matches the WHERE clause used for your download_delete_cursor event script.

```
public String endDownloadDeletes(
  Timestamp ts,
  String user,
  String table )
{ return( "UPDATE Leads SET OnRemote = 0
     WHERE LastModified >= ?
     AND Owner = ? AND DeleteFlag=1" ); }
```

.NET example

The following stored procedure call registers a .NET method called EndDownloadDeletes as the script for the end_download_deletes table event when synchronizing the script version ver1 and the table table1.

```
call ml_add_dnet_table_script(
  'ver1',
  'table1',
  'end_download_deletes',
  'TestScripts.Test.EndDownloadDeletes'
)
```

You may want to mark a row as deleted on the remote database. Following is the sample .NET method EndDownloadDeletes. It updates a consolidated database column called OnRemote to indicate that the record no longer resides on the remote database. The WHERE clause on the UPDATE matches the WHERE clause used for your download_delete_cursor event script.

```
public void EndDownloadDeletes(
  DateTime timestamp, string user, string table)
{ return( "UPDATE Leads SET OnRemote = 0
     WHERE LastModified >= ?
     AND Owner = ? AND DeleteFlag=1" ); }
```

end_download_rows table event

Function

Processes statements related to a specific table just after preparing a list of rows to be inserted or updated in the specified table in the remote database.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Event parameters are optional only if no subsequent parameters are specified. For example, you must use parameter 1 if you want to use parameter 2.

Item	Parameter	Description
1	last_download	TIMESTAMP
2	ml_username	VARCHAR(128)
3	table	VARCHAR(128)

Default action

None.

Description

This script is executed immediately after preparing the stream of rows to be inserted or updated in the named table in the remote database.

You can have one end_download_rows script for each table in the remote database.

The last_download timestamp is the value obtained from the consolidated database during the last successful synchronization immediately prior to the download phase. If the current user has never synchronized successfully, this value is set to 1900-01-01.

See also

- "begin_download_rows table event" on page 353
- "end_download connection event" on page 386
- "end_download_deletes table event" on page 390
- "begin_download_deletes table event" on page 351

SQL example

You may want to mark a row as successfully downloaded to the remote database. The following script updates a column in the consolidated database called OnRemote. *Note:* The WHERE clause on the UPDATE matches the WHERE clause used for your download_delete_cursor event script.

```
call ml_add_table_script(
   'version1',
   'Leads',
   'end_download_rows',
   'UPDATE Leads SET OnRemote = 1 WHERE LastModified >= ?
      AND Owner = ? AND DownloadFlag=1');
```

Java example

The following stored procedure call registers a Java method called endDownloadRows as the script for the end_download_rows table event when synchronizing the script version ver1.

```
call ml_add_java_table_script(
   'ver1',
   'table1',
   'end_download_rows',
   'ExamplePackage.ExampleClass.endDownloadRows' )
```

Following is the sample Java method endDownloadRows. It prints a message to the MobiLink output log. (This might be useful at development time but would slow down a production server.)

```
public String endDownloadRows(
  Timestamp ts,
  String user,
  String table )
{  java.lang.System.out.println( "Done downloading
   inserts and updates for table " + table );
   return( null ); }
```

.NET example

The following stored procedure call registers a .NET method called EndDownloadRows as the script for the end_download_rows table event when synchronizing the script version ver1 and the table table1.

```
call ml_add_dnet_table_script(
 'ver1', 'table1', 'end_download_rows',
  'TestScripts.Test.EndDownloadRows'
)
```

Following is the sample Java method endDownloadRows. It prints a message to the MobiLink output log. (This might be useful at development time but would slow down a production server.)

```
public void EndDownloadRows(
  DateTime timestamp,
  string user,
  string table )
{  System.Console.WriteLine( "Done downloading
   inserts and updates for table " + table );
   return( null ); }
```

end_publication connection event

Function

Provides useful information about the publication(s) being synchronized.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Item	Parameter	Description
1	generation_number	INTEGER. If your deployment does not use file-based downloads, this parameter can be ignored. The default is 1.
2	ml_username	VARCHAR(128). If an UltraLite remote is synchronizing with UL_-SYNC_ALL, this event is invoked once with the name 'unknown'.
3	publication_name	VARCHAR(128)
4	last_upload	TIMESTAMP. Last successful up-load.
5	last_download	TIMESTAMP. Last successful download.

Default action

None.

Description

This event lets you design synchronization logic based on the publications currently being synchronized. This event is invoked in the same transaction as the end_synchronization event, and is invoked before the end_synchronization event. It is invoked once per publication being synchronized.

If the current synchronization successfully applied an upload, the last_upload parameter will contain the time this latest upload was applied. If the current synchronization has a successful download acknowledgement, the last_download time will contain the time this latest download was generated. This is the same value that was passed to the download scripts as the last download timestamp.

Generation number

The generation_number parameter is specifically for file-based downloads.

The output value of the generation number is passed from the begin_publication script to the end_publication script. The meaning of the generation_number depends on whether the current synchronization is being used to create a download file, or whether the current synchronization has an upload.

In file-based downloads, generation numbers are used to force an upload before the download. The number is stored in the download file.

See also

- "begin_publication connection event" on page 356
- "File-Based Downloads" on page 85

SQL example

You may want to record the information for each publication being synchronized. The following example calls ml_add_connection_script to assign the event to a stored procedure called RecordPubEndSync.

```
call ml_add_connection_script(
   'version1',
   'end_publication',
   'call RecordPubEndSync( ?, ?, ?, ?, ? )' );
```

Java example

The following stored procedure call registers a Java method called endPublication as the script for the begin_publication connection event when synchronizing the script version ver1.

```
call ml_add_java_connection_script(
   'ver1',
   'end_publication',
   'ExamplePackage.ExampleClass.endPublication' )
```

Following is the sample Java method endPublication. It outputs a message to the MobiLink log. (This might be useful at development time but would slow down a production server.)

```
public String endPublication(
    ianywhere.ml.script.InOutInteger generation_number,
    String user,
    String pub_name,
    Timestamp last_upload,
    Timestamp last_download )
{  System.out.println(
     "Finished synchronizing publication " + pub_name );
   return( null ); }
```

.NET example

The following stored procedure call registers a .NET method called EndPub as the script for the end_publication connection event when synchronizing the script version ver1.

```
call ml_add_dnet_connection_script( 'ver1',
  'end_publication',
  'TestScripts.Test.EndPub'
)
```

Following is the sample Java method endPublication. It outputs a message to the MobiLink log. (This might be useful at development time but would slow down a production server.)

```
public void EndPub(
    ref int generation_number,
    string user,
    string pub_name,
    DateTime last_upload,
    DateTime last_download )
{
    Console.Write(
      "Finished synchronizing publication " + pub_name );
}
```

end_synchronization connection event

Function

Processes any statements at the time an application disconnects from the MobiLink synchronization server upon completion of the synchronization process.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Item	Parameter	Description
1	ml_username	VARCHAR(128)
2	sync_ok	INTEGER. This value is 1 for a successful synchronization and 0 for an unsuccessful synchronization.

Default action

None.

Description

The MobiLink synchronization server executes this script after synchronization is complete and, if expecting a download acknowledgement, the MobiLink client has returned confirmation of receipt of the download stream.

This script is executed within a separate transaction after the download transaction. It is useful for maintaining statistics.

See also

- "begin_synchronization connection event" on page 359
- "begin_synchronization table event" on page 361
- "end_synchronization table event" on page 400

SQL example

The following Adaptive Server Anywhere SQL script calls a stored procedure that records the end time of the synchronization attempt along with its success or failure status.

```
call ml_add_connection_script(
 'ver1',
 'end_synchronization',
 'call RecordEndOfSyncAttempt(?,?)' )
```

Java example

The following stored procedure call registers a Java method called endSynchronizationConnection as the script for the end_synchronization event when synchronizing the script version ver1.

```
call ml_add_java_connection_script(
  'ver1',
  'end_synchronization',
  'ExamplePackage.ExampleClass.endSynchronizationConnection'
)
```

Following is the sample Java method endSynchronizationConnection. It uses a JDBC connection to execute an update. This syntax is for Adaptive Server Anywhere consolidated databases.

```
public String endSynchronizationConnection(
 String user )
  throws java.sql.SQLException
{  execUpdate( _syncConn, "UPDATE sync_count set cnt =
   count + 1 where user_id = '" + user + "' " );
   return( null ); }
```

.NET example

The following stored procedure call registers a .NET method called EndSync as the script for the end_synchronization connection event when synchronizing the script version ver1.

```
call ml_add_dnet_connection_script(
  'ver1',
  'end_synchronization',
  'TestScripts.Test.EndSync'
)
```

Following is the sample .NET method EndSync. It updates the table sync_count. This syntax is for Adaptive Server Anywhere consolidated databases.

```
public void EndSync( string user )
{
  return("UPDATE sync_count set cnt =
   count + 1 where user_id = '" + user + "' ");
}
```

end_synchronization table event

Function

Processes statements related to a specific table at the time an application disconnects from the MobiLink synchronization server upon completion of the synchronization process.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Event parameters are optional only if no subsequent parameters are specified. For example, you must use parameter 1 if you want to use parameter 2.

Item	Parameter	Description
1	ml_username	VARCHAR(128)
2	table	VARCHAR(128)
3	sync_ok	INTEGER. This value is 1 for a successful synchronization and 0 for an unsuccessful synchronization.

Default action

None.

Description

The MobiLink synchronization server executes this script after an application has synchronized and is about to disconnect from the MobiLink synchronization server, and before the connection level script of the same name.

You can have one end_synchronization script for each table in the remote database.

See also

- "begin_synchronization table event" on page 361
- "end_synchronization connection event" on page 398
- "end_synchronization table event" on page 400

SQL example

The following Adaptive Server Anywhere SQL script drops a temporary table created by the begin_synchronization script.

```
call ml_add_table_script(
 'ver1',
 'sales_order',
 'end_synchronization',
 'drop table #sales_order' )
```

Java example

The following stored procedure call registers a Java method called

endSynchronizationTable as the script for the end_synchronization table event when synchronizing the script version ver1.

```
call ml_add_java_table_script(
   'ver1',
   'table1',
   'end_synchronization',
   'ExamplePackage.ExampleClass.endSynchronizationTable' )
```

Following is the sample Java method endSynchronizationTable. It returns SQL to drop a temporary table created by the begin_synchronization script.

```
public String endSynchronizationTable(  String user,
String table )
{  return( "drop table #sales_order" ); }
```

.NET example

The following stored procedure call registers a .NET method called EndTableSync as the script for the end_synchronization table event when synchronizing the script version ver1 and the table table1.

```
call ml_add_dnet_table_script(
  'ver1', 'table1', 'end_synchronization',
  'TestScripts.Test.EndTableSync'
)
```

Following is the sample Java method EndTableSync. It returns SQL to drop a temporary table created by the begin_synchronization script.

```
public void EndTableSync( string user, string table )
{  return( "drop table #sales_order" ); }
```

end_upload connection event

Function

Processes any statements just after the MobiLink synchronization server concludes processing uploaded inserts, updates, and deletes.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Item	Parameter	Description
1	ml_username	VARCHAR(128)

Default action

None.

Description

The MobiLink synchronization server executes this script as the last step in the processing of uploaded information. Upload information is processed in a single transaction. The execution of this script is the last action in this transaction before statistical scripts.

See also

- "begin_upload connection event" on page 363
- "end_upload table event" on page 404

Java example

The following stored procedure call registers a Java method called endUploadConnection as the script for the end_upload connection event when synchronizing the script version ver1. This syntax is for Adaptive Server Anywhere consolidated databases.

```
call ml_add_java_connection_script(
   'ver1',
   'end_upload',
   'ExamplePackage.ExampleClass.endUploadConnection' )
```

Following is the sample Java method endUploadConnection. It calls a method to perform operations on the database.

```
public String endUploadConnection( String user )
{  // clean up new and old tables
   Iterator    two_iter = _tables_with_ops.iterator();
   while( two_iter.hasNext() )
   {  TableInfo cur_table = (TableInfo)two_iter.next();
      dumpTableOps( _sync_conn, cur_table ); }
    _tables_with_ops.clear(); }
```

.NET example

The following stored procedure call registers a .NET method called EndUpload as the script for the end_upload connection event when synchronizing the script version ver1.

```
call ml_add_dnet_connection_script(
  'ver1',
  'end_upload',
  'TestScripts.Test.EndUpload'
)
```

Following is the C# signature for the call EndUpload.

```
public void EndUpload( string user )
```

end_upload table event

Function

Processes statements related to a specific table just after the MobiLink synchronization server concludes processing the stream of uploaded inserts, updates, and deletions.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Event parameters are optional only if no subsequent parameters are specified. You must use parameter 1 if you want to use parameter 2.

Item	Parameter	Description
1	ml_username	VARCHAR(128)
2	table	VARCHAR(128)

Default action

None.

Description

The MobiLink synchronization server executes this script as the last step in the processing of uploaded information. Upload information is processed in a separate transaction. The execution of this script is the last table-specific action in this transaction.

You can have one end_upload script for each table in the remote database.

See also

- "begin_upload table event" on page 365
- "end_upload connection event" on page 402

SQL example

The following statements define a stored procedure and an end_upload script. The first piece of code calls the ml_add_table_script, and the second creates the ULCustomerIDPool_maintain procedure.

```
  ml_add_table_script(
    'custdb',
    'ULCustomerIDPool',
    'end_upload',
ULCustomerIDPool_maintain( ? );)

CREATE OR REPLACE PROCEDURE ULCustomerIDPool_maintain(
  SyncUserID IN integer )
AS
  pool_count INTEGER;
  pool_max   INTEGER;
BEGIN
  -- Determine how many ids to add to the pool
```

```
SELECT COUNT(*)
    INTO pool_count
    FROM ULCustomerIDPool
    WHERE pool_emp_id = SyncUserID;
  -- Determine the current Customer id max

SELECT MAX(pool_cust_id)
    INTO pool_max
    FROM ULCustomerIDPool;
  -- Top up the pool with new ids

 WHILE pool_count < 20 LOOP
    pool_max := pool_max + 1;
      INSERT INTO ULCustomerIDPool(
          pool_cust_id, pool_emp_id )
            VALUES ( pool_max, SyncUserID );
      pool_count := pool_count + 1;
  END LOOP;
END;
```

Java example

The following stored procedure call registers a Java method called endUploadTable as the script for the end_upload table event when synchronizing the script version ver1.

```
call ml_add_java_table_script(
   'ver1',
   'table1'
   'end_upload',
   'ExamplePackage.ExampleClass.endUploadTable' )
```

Following is the sample Java method endUploadTable. It generates a delete for a table with a name related to the passing-in table name. This syntax is for Adaptive Server Anywhere consolidated databases.

```
public String endUploadTable(   String user,
String table)
{  return( "DELETE from '" + table + "_temp'" ); }
```

.NET example

The following stored procedure call registers a .NET method called EndTableUpload as the script for the end_upload table event when synchronizing the script version ver1 and the table table1.

```
call ml_add_dnet_table_script(
   'ver1',
   'table1',
   'end_upload',
   'TestScripts.Test.EndTableUpload'
)
```

Following is the C# signature for the call EndTableUpload.

```
public void EndTableUpload(
   string user, string table )
```

The following C# example moves rows inserted into a temporary table into the table passed into the script.

```
public void EndUpload( string user, string table )
{
  DBCommand stmt = curConn.CreateCommand();

  // move the uploaded rows to the destination table
  stmt.CommandText = "INSERT INTO "
    + table
    + " SELECT * FROM dnet_ul_temp";
  stmt.ExecuteNonQuery();
  stmt.Close();
}
```

end_upload_deletes table event

Function

Processes statements related to a specific table just after applying deletes uploaded from the specified table in the remote database.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Event parameters are optional only if no subsequent parameters are specified. You must use parameter 1 if you want to use parameter 2.

Item	Parameter	Description
1	ml_username	VARCHAR(128)
2	table	VARCHAR(128)

Default action

None.

Description

This script is run immediately after applying the changes that result from rows deleted in the remote table named in the second parameter.

You can have one end_upload_deletes script for each table in the remote database.

See also

- "begin_upload_deletes table event" on page 367

SQL example

You can use this event to process rows deleted during the upload stream on an intermediate table. You can compare the rows in the base table with rows in the intermediate table and decide what to do with the deleted row.

```
Call ml_add_table_script(
  'version1',
  'Leads',
  'end_uploads_deletes',
  'call EndUploadDeletesLeads()');
Create procedure EndUploadDeletesLeads ( )
Begin
  FOR names AS curs CURSOR FOR
  SELECT LeadID
    FROM Leads
    WHERE LeadID NOT IN (SELECT LeadID FROM T_Leads);
  DO
   CALL decide_what_to_do( LeadID );
  END FOR;
end
```

Java example

The following stored procedure call registers a Java method called endUploadDeletes as the script for the end_upload_deletes table event when synchronizing the script version ver1.

```
call ml_add_java_table_script(
   'ver1',
   'table1',
   'end_upload_deletes',
   'ExamplePackage.ExampleClass.endUploadDeletes' )
```

Following is the sample Java method a endUploadDeletes. It calls a Java method that manipulates the database.

```
public String endUploadDeletes( String user,
String table )
  throws java.sql.SQLException
{  processUploadedDeletes( _syncConn, table );
   return( null ); }
```

.NET example

The following stored procedure call registers a .NET method called EndUploadDeletes as the script for the end_upload_deletes table event when synchronizing the script version ver1 and the table table1.

```
call ml_add_dnet_table_script(
   'ver1',
   'table1',
   'end_upload_deletes',
   'TestScripts.Test.EndUploadDeletes'
)
```

Following is the sample .NET method a EndUploadDeletes. It calls a .NET method that manipulates the database.

```
public void EndUploadDeletes( string user, string table )
{  processUploadedDeletes( _syncConn, table );
   return( null ); }
```

end_upload_rows table event

Function

Processes statements related to a specific table just after applying uploaded inserts and updates from the specified table in the remote database.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Event parameters are optional only if no subsequent parameters are specified. You must use parameter 1 if you want to use parameter 2.

Item	Parameter	Description
1	ml_username	VARCHAR(128)
2	table	VARCHAR(128)

Default action

None.

Description

Uploaded information can require inserting or updating rows in the consolidated database. This script is run immediately after applying the changes that result from modifications to the remote table named in the second parameter.

You can have one end_upload_rows script for each table in the remote database.

See also

♦ "begin_upload_rows table event" on page 369

SQL example

You use this event to process rows deleted during the upload stream on an intermediate table. You can compare the rows in the base table with the rows in the intermediate table and decide what to do with the deleted row as the following example illustrates.

```
Call ml_add_table_script(
   'version1',
   'Leads',
   'end_uploads_deletes',
   'call EndUploadDeletesLeads()');
Create procedure EndUploadDeletesLeads ( )
Begin
  FOR names AS curs CURSOR FOR
  SELECT LeadID FROM Leads
    WHERE LeadID NOT IN (select LeadID from T_Leads);
  DO
   CALL decide_what_to_do( LeadID );
  END FOR;
end
```

Java example

The following stored procedure call registers a Java method called endUploadRows as the script for the end_upload_rows table event when synchronizing the script version ver1.

```
call ml_add_java_table_script(
   'ver1',
   'table1',
   'end_upload_rows',
   'ExamplePackage.ExampleClass.endUploadRows' )
```

Following is the sample Java method endUploadRows. It calls a Java method that manipulates the database.

```
public String endUploadRows(String user,
String table )
  throws java.sql.SQLException
{  processUploadedRows( _syncConn, table );
   return( null ); }
```

.NET example

The following stored procedure call registers a .NET method called EndUploadRows as the script for the end_upload_rows table event when synchronizing the script version ver1 and the table table1.

```
call ml_add_dnet_table_script(
  'ver1',
  'table1',
  'end_upload_rows',
  'TestScripts.Test.EndUploadRows'
)
```

Following is the sample .NET method endUploadRows. It calls a .NET method that manipulates the database.

```
public void EndUploadRows(
  string user,
  string table )
{  processUploadedRows( _syncConn, table );
   return( null ); }
```

example_upload_cursor table event

Function

Provides an event that the MobiLink synchronization server does not use during processing of the upload stream to handle rows inserted into the remote database. The event is not called.

Parameters

Item	Parameter
1	column 1
2	column 2
…	…

Description

The statement based example_upload_cursor script performs direct inserts of column values identical to those specified in the example_upload_cursor statement. The example_upload_cursor event is not called by MobiLink.

SQL example

The script is not called. If it were called, it would insert the values into a table named Customer in the consolidated database. The final column of the table identifies the Customer as active. The final column does not appear in the remote database.

```
SELECT cust_id, name, rep_id
FROM customer
WHERE cust_id=?
```

example_upload_delete table event

Function

Processes the upload stream to handle rows deleted from the remote database. The script is not called by MobiLink.

Parameters

Item	Parameter
1	column 1
2	column 2
…	…

Description

The statement based example_upload_delete script handles rows that are deleted in the remote database. The action taken at the consolidated database can be a DELETE statement, but need not be.

See also

- "upload_delete table event" on page 459

SQL example

This example marks customers who are deleted from the remote database as inactive.

```
UPDATE Customer
SET active = 0
WHERE cust_id=?
```

example_upload_insert table event

Function

Provides an event that the MobiLink synchronization server uses during processing of the upload stream to handle rows inserted into the remote database.

Parameters

Item	Parameter
1	column 1
2	column 2
…	…

Description

The statement-based example_upload_insert script performs direct inserts of column values identical to those specified in the upload_insert statement.

The example_upload_insert event is not called.

See also

- ♦ "upload_insert table event" on page 463

SQL example

The script is not called. If it were called, it would insert the values into a table named Customer in the consolidated database. The final column of the table identifies the Customer as active. The final column does not appear in the remote database.

```
INSERT INTO Customer( cust_id, name, rep_id )
VALUES ( ?, ?, ? )
```

example_upload_update table event

Function

An example event for the upload stream to handle rows updated at the remote database. The example script is not called by MobiLink but is identical in form to the upload_update event.

Parameters

Clause	**Parameters**
SET	column 1
	column 2
	. . .
WHERE	primary key 1
	primary key 2
	. . .

Description

You create an example_upload_update event script by using the option -za in the dbmlsync command line.

See also

- “upload_update table event” on page 475

SQL example

This example handles updates made to the Customer table in the remote database. The script updates the values in a table named Customer in the consolidated database. Note: The script is never called and is only an example script.

```
UPDATE Customer
SET name=?, rep_id=?
WHERE cust_id=?
```

handle_error connection event

Function

Executed whenever the MobiLink synchronization server encounters a SQL error.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Event parameters are optional only if no subsequent parameters are specified. For example, you must use parameter 1 if you want to use parameter 2.

Item	Parameter	Description
1	action_code	INTEGER. This is an IN-OUT parameter.
2	error_code	INTEGER
3	error_message	TEXT
4	ml_username	VARCHAR(128)
5	table	VARCHAR(128). If the script is not a table script, the table name is NULL.

Default action

When no handle_error script is defined or this script causes an error, the default action code is 3000: rollback the current transaction and cancel the current synchronization.

Description

The MobiLink synchronization server sends in the current action_code. Initially, this is set to 3000 for each set of errors caused by a single SQL operation. Usually, there is only one error per SQL operation, but there may be more. This handle_error script is called once per error in the set. The action code passed into the first error is 3000. Subsequent calls are passed in the action code returned by the previous call. MobiLink will use the numerically highest value returned from multiple calls.

You can modify the action code in the script, and return a value instructing MobiLink how to proceed. The action code parameter takes one of the following values:

- **1000** Skip the current row and continue processing.
- **3000** Rollback the current transaction and cancel the current synchronization. This is the default action code, and is used when no

handle_error script is defined or this script causes an error.

- **4000** Rollback the current transaction, cancel the synchronization, and shut down the MobiLink synchronization server.

SQL scripts for the handle_error event must be implemented as stored procedures.

The MobiLink synchronization server executes this script whenever it encounters an error during the synchronization process. The error codes and message allow you to identify the nature of the error. If the error happened as part of synchronization, the user name is supplied. Otherwise, this value is NULL.

If the error happened while manipulating a particular table, the table name is supplied. Otherwise, this value is NULL. The table name is the name of a table in the client application. This name may or may not have a direct counterpart in the consolidated database, depending upon the design of the synchronization system.

The action code tells the MobiLink synchronization server what to do next. Before it calls this script, the MobiLink synchronization server sets the action code to a default value, which depends upon the severity of the error. Your script may modify this value. Your script must return or set an action code.

You can return a value from the handle_error script in two ways.

- Pass the action parameter to an OUTPUT parameter of a procedure:

  ```
  CALL my_handle_error( ?, ?, ?, ?, ? )
  ```

- Set the action code via a procedure or function return value:

  ```
  ? = CALL my_handle_error( ?, ?, ?, ? )
  ```

  Most DBMSs use the RETURN statement to set the return value from a procedure or function.

The CustDB sample application contains error handlers for various database-management systems.

See also

- "report_error connection event" on page 438
- "report_odbc_error connection event" on page 440
- "handle_odbc_error connection event" on page 419

SQL example

The following example works with an Adaptive Server Anywhere consolidated database. It allows your application to ignore redundant inserts.

```
call ml_add_connection_script(
    'ver1',
    'handle_error',
   'call ULHandleError(?,?,?,?,?)' )
```

```
CREATE PROCEDURE ULHandleError(
   INOUT action integer,
   IN error_code integer,
   IN error_message varchar(1000),
   IN user_name varchar(128),
   IN table_name varchar(128) )
BEGIN
  -- -196 is SQLE_INDEX_NOT_UNIQUE
  -- -194 is SQLE_INVALID_FOREIGN_KEY
  if error_code = -196 or error_code = -194 then
      -- ignore the error and keep going
      SET action = 1000;
  else
      -- abort the synchronization
      SET action = 3000;
  end if;
END
```

Java example

The following stored procedure call registers a Java method called handleError as the script for the handle_error connection event when synchronizing the script version ver1.

```
call ml_add_java_connection_script(
    'ver1',
    'handle_error',
    'ExamplePackage.ExampleClass.handleError' )
```

Following is the sample Java method handleError. It processes an error based on the data that is passed in. It also determines the resulting error code.

```
public String handleError(
  ianywhere.ml.script.InOutInteger actionCode,
  int errorCode,
  String errorMessage,
  String    user,
  String  table )
{  int new_ac;
   if( user == null )
   {  new_ac = handleNonSyncError( errorCode,
      errorMessage ); }
   else if( table == null )
{  new_ac = handleConnectionError( errorCode,
   errorMessage, user ); }
else
{  new_ac = handleTableError( errorCode,
   errorMessage, user, table ); }
// keep the most serious action code
if( actionCode.getValue() < new_ac )
{  actionCode.setValue( new_ac ); }
return( null ); }
```

handle_odbc_error connection event

Function

Executed whenever the MobiLink synchronization server encounters an error triggered by the ODBC Driver Manager.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Event parameters are optional only if no subsequent parameters are specified. For example, you must use parameter 1 if you want to use parameter 2.

Item	Parameter	Description
1	action_code	INTEGER. This is an INOUT parameter.
2	ODBC_state	VARCHAR(5)
3	error_message	TEXT
4	ml_username	VARCHAR(128)
5	table	VARCHAR(128)

Default action

The MobiLink synchronization server selects a default action code. You can modify the action code in the script, and return a value instructing MobiLink how to proceed. The action code parameter takes one of the following values:

- **1000** Skip the current row and continue processing.
- **3000** Rollback the current transaction and cancel the current synchronization. This is the default action code, and is used when no handle_error script is defined or this script causes an error.
- **4000** Rollback the current transaction, cancel the synchronization, and shut down the MobiLink synchronization server.

Description

The MobiLink synchronization server executes this script whenever it encounters an error flagged by the ODBC Driver Manager during the synchronization process. The error codes allow you to identify the nature of the error.

The action code tells the MobiLink synchronization server what to do next. Before it calls this script, the MobiLink synchronization server sets the action code to a default value, which depends upon the severity of the error.

Your script may modify this value. Your script must return or set an action code.

The handle_odbc_error script is called after the handle_error and report_error scripts, and before the report_odbc_error script.

When only one, but not both, error-handling script is defined, the return value from that script decides error behavior. When both error-handling scripts are defined, the MobiLink synchronization server uses the numerically highest action code. If both handle_error and handle_ODBC_error are defined, MobiLink uses the numerically highest action code returned from all calls.

See also

- ♦ "handle_error connection event" on page 415
- ♦ "report_error connection event" on page 438
- ♦ "report_odbc_error connection event" on page 440

SQL example

The following example works with an Adaptive Server Anywhere consolidated database. It allows your application to ignore ODBC integrity constraint violations.

```
call ml_add_connection_script(
 'ver1',
 'handle_odbc_error',
 'call HandleODBCError( ?, ?, ?, ?, ? )'  )
CREATE PROCEDURE HandleODBCError( INOUT action integer,
IN odbc_state varchar(5), IN error_message varchar(1000),
IN user_name varchar(128), IN table_name varchar(128) )
BEGIN
  if odbc_state = '23000' then
      -- ignore the error and keep going
      SET action = 1000;
  else
      -- abort the synchronization
      SET action = 3000;
  end if;
END
```

Java example

The following stored procedure call registers a Java method called handleODBCError as the script for the handle_odbc_error event when synchronizing the script version ver1.

```
call ml_add_java_connection_script(
'ver1', 'handle_odbc_error',
         'ExamplePackage.ExampleClass.handleODBCError'
)
```

Following is the sample Java method handleODBCError. It processes an error based on the data that is passed in. It also determines the resulting error code.

```
public String handleODBCError(
  ianywhere.ml.script.InOutInteger actionCode,
  String ODBCState,
  String errorMessage,
  String user,
  String table )
{  int new_ac;
   if( user == null )
   {  new_ac = handleNonSyncError( ODBCState,
      errorMessage ); }
   else if( table == null )
   {  new_ac = handleConnectionError( ODBCState,
      errorMessage, user ); }
   else {  new_ac = handleTableError( ODBCState,
           errorMessage, user, table ); }
   // keep the most serious action code
   if( actionCode.getValue() < new_ac )
   {  actionCode.setValue( new_ac ); }
   return( null ); }
```

modify_error_message connection event

Function

The script can be used to customize the message text (error, warning, and information) that is sent to remote databases.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Event parameters are optional only if no subsequent parameters are specified. You must use parameter 1 if you want to use parameter 2.

Item	Parameter	Description
1	error_message	VARBINARY(1024). This is an INOUT parameter.
2	ml_username	VARCHAR(128)
3	error_code	INT

Default action

None.

Description

SQL scripts for the modify_error_message event must be implemented as stored procedures.

SQL example

The following example downloads everything from one day ago, regardless of whether the databases were synchronized since then.

First, create a procedure for your Adaptive Server Anywhere consolidated database:

```
CREATE PROCEDURE ModifyLastDownloadTimestamp(
  inout last_download_time TIMESTAMP,
  in user_name VARCHAR(128) )
BEGIN
  SELECT dateadd(day, -1, last_download_time )
  INTO last_download_time
END
```

Second, install the script into your Adaptive Server Anywhere consolidated database:

```
call ml_add_connection_script(
  'modify_ts_test',
  'modify_last_download_timestamp',
  'call  ModifyLastDownloadTimestamp ( ?, ? )' )
```

Java example

The following stored procedure call registers a Java method called

modifyLastDownloadTimestamp as the script for the modify_last_download_timestamp connection event when synchronizing the script version ver1.

```
call ml_add_java_connection_script(
  'ver1',
  'modify_last_download_timestamp',
  'ExamplePackage.ExampleClass.modifyLastDownloadTimestamp' )
```

Following is the sample Java method modifyLastDownloadTimestamp. It prints the current and new timestamp and modifies the timestamp that is passed in.

```
public String modifyLastDownloadTimestamp(
     Timestamp last_download_time,
     String user_name )
{  java.lang.System.out.println( "old date: " +
   last_download_time.toString() );
   last_download_time.setDate(
   last_download_time.getDate() -1 );
   java.lang.System.out.println( "new date: " +
   last_download_time.toString() );
   return( null ); }
```

modify_last_download_timestamp connection event

Function

The script can be used to modify the last_download timestamp for the current synchronization.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Event parameters are optional only if no subsequent parameters are specified. You must use parameter 1 if you want to use parameter 2.

Item	Parameter	Description
1	last_download_timestamp	TIMESTAMP. This is an INOUT parameter.
2	ml_username	VARCHAR(128)

Default action

None.

Description

Use this script when you want to modify the last download timestamp for the current synchronization. If this script is defined, the MobiLink synchronization server calls this script and uses the modified last_download timestamp as the last_download timestamp passed to the download scripts.

SQL scripts for the modify_last_download_timestamp event must be implemented as stored procedures. The MobiLink synchronization server passes in the last_download_timestamp as the first parameter to the stored procedure, and replaces the timestamp by the first value passed out by the stored procedure.

For example, if you defined the following download_cursor script, the MobiLink synchronization server would replace the ? with the value of the modified last download timestamp before executing the SELECT statement:

```
SELECT pk, c2, c3 FROM test WHERE last_modified >= ?
```

This script is executed just before the prepare_for_download script, in the same transaction.

SQL example

First, create a procedure for your Adaptive Server Anywhere consolidated database. In Oracle:

```
CREATE PROCEDURE modify_nldts(
      nldts      OUT DATE,
      ldts      IN DATE,
      user_name    IN VARCHAR )
       AS
       BEGIN
       -- N is max replication latency in consolidated cluster
      nldts := nldts - N;
       END;
```

In Adaptive Server Anywhere, Adaptive Server Enterprise, or Microsoft SQL Server:

```
CREATE PROCEDURE modify_nldts
      @nldts     DATETIME OUTPUT,
      @ldts      DATETIME,
      @t_name    VARCHAR( 128 )
       AS
       BEGIN
       -- N is max replication latency in consolidated cluster
      SELECT @nldts = @nldts - N
       END
```

Next, install the script into your consolidated database. For an Adaptive Server Anywhere consolidated database:

```
ml_add_connection_script(
  'my_version',
  'modify_last_download_timestamp',
  '{call modify_nldts(?,?,?)}' )
```

Java example

The following stored procedure call registers a Java method called modifyLastDownloadTimestamp as the script for the modify_last_download_timestamp connection event when synchronizing the script version ver1.

```
call ml_add_java_connection_script(
  'ver1',
  'modify_last_download_timestamp',
  'ExamplePackage.ExampleClass.modifyLastDownloadTimestamp' )
```

Following is the sample Java method modifyLastDownloadTimestamp. It prints the current and new timestamp and modifies the timestamp that is passed in.

```
public String modifyLastDownloadTimestamp(
     Timestamp last_download_time,
     String user_name )
{  java.lang.System.out.println( "old date: " +
   last_download_time.toString() );
   last_download_time.setDate(
   last_download_time.getDate() -1 );
   java.lang.System.out.println( "new date: " +
   last_download_time.toString() );
   return( null ); }
```

modify_next_last_download_timestamp connection event

Function

The script can be used to modify the last_download timestamp for the next synchronization.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Event parameters are optional only if no subsequent parameters are specified. For example, you must use parameter 1 if you want to use parameter 2.

Item	Parameter	Description
1	download_timestamp	TIMESTAMP. This is an INOUT parameter.
2	last_download_timestamp	TIMESTAMP
3	ml_username	VARCHAR(128)

Default action

None.

Description

Use this script when you want to modify the last download timestamp for the next synchronization. If this script is defined, the MobiLink synchronization server calls this script and sends the next last download timestamp down to the remote, which will send it as part of the next synchronization.

SQL scripts for the modify_next_last_download_timestamp event must be implemented as stored procedures. The MobiLink synchronization server passes in the download_timestamp as the first parameter to the stored procedure, and replaces the timestamp by the first value passed out by the stored procedure.

You can use this script to modify the download timestamp that the MobiLink synchronization server sends to the MobiLink client. If the client is dbmlsync, the timestamp is stored in the SYSSYNC system table.

This script is executed in the download transaction, after downloading user tables.

SQL example

The following example shows one application of this script. First, create a procedure for your Adaptive Server Anywhere consolidated database:

```
CREATE PROCEDURE ModifyNextDownloadTimestamp(
  inout download_timestamp TIMESTAMP ,
  in last_download TIMESTAMP ,
  in user_name VARCHAR(128) )
  BEGIN
    SELECT dateadd(hour, -1, download_timestamp )
    INTO download_timestamp
END
```

Second, install the script into your Adaptive Server Anywhere consolidated database:

```
call ml_add_connection_script(
  'modify_ts_test',
  'modify_next_last_download_timestamp',
  'call  ModifyNextDownloadTimestamp ( ?, ?, ? )' )
```

Java example

The following stored procedure call registers a Java method called modifyNextDownloadTimestamp as the script for the modify_next_last_download_timestamp connection event when synchronizing the script version ver1.

```
call ml_add_java_connection_script(
   'ver1',
   'modify_next_last_download_timestamp',
   'ExamplePackage.ExampleClass.modifyNextDownloadTimestamp' )
```

Following is the sample Java method modifyNextDownloadTimestamp. It sets the download timestamp back an hour.

```
public String modifyNextDownloadTimestamp(
  Timestamp download_timestamp,
  Timestamp last_download,
  String user_name )
{  download_timestamp.setHours(
   download_timestamp.getHours() -1 );
   return( null ); }
```

modify_user connection event

Function

Provide the MobiLink user name.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Item	Parameter	Description
1	ml_username	VARCHAR(128). This is an INOUT parameter.

Default action

None.

Description

The MobiLink server provides the user name as a parameter when it calls scripts; the user name is sent by the MobiLink client. In some cases, you may want to have an alternate user name. This script allows you to modify the user name used in calling MobiLink scripts.

The ml_username parameter must be long enough to hold the user name.

SQL scripts for the modify_user event must be implemented as stored procedures.

See also

- "authenticate_user connection event" on page 336
- "authenticate_user_hashed connection event" on page 340

SQL example

The following example works with an Adaptive Server Anywhere consolidated database. It maps a remote database user name to the id of the user using the device, by using a mapping table called user_device. This technique can be used when the same person has multiple remotes (such as a PDA and a laptop) requiring the same synchronization logic (based on the user's name or id).

```
call ml_add_connection_script(
 'ver1',
 'modify_user',
 'call ModifyUser( ? )'  )
CREATE PROCEDURE ModifyUser( INOUT u_name varchar(128) )
BEGIN
 select user_name
    into u_name
    from user_device
    where device_name = u_name
END
```

Java example

The following stored procedure call registers a Java method called modifyUser as the script for the modify_user connection event when synchronizing the script version ver1.

```
call ml_add_java_connection_script(
   'ver1',
   'modify_user',
   'ExamplePackage.ExampleClass.modifyUser' )
```

Following is the sample Java method modifyUser. It gets the user ID from the database and then uses it to set the user name.

```
public void ModifyUser( InOutString io_user_name )
  throws SQLException
{  Statement uid_select = curConn.createStatement();
   ResultSet uid_result = uid_select.executeQuery(
   "select rep_id from SalesRep where name = '" +

   io_user_name.getValue() + "' " );
   uid_result.next();
   io_user_name.setValue(
   java.lang.Integer.toString(uid_result.getInt( 1 ))
   uid_result.close();
   uid_select.close();
   return; }
```

.NET example

The following stored procedure call registers a .NET method called ModUser as the script for the modify_user connection event when synchronizing the script version ver1.

```
call ml_add_dnet_connection_script(
  'ver1',
  'modify_user',
  'TestScripts.Test.ModUser'
)
```

Following is the C# signature for the call ModUser.

```
public void ModUser( string user )
```

new_row_cursor table event (deprecated)

Function

Defines the insert cursor that the MobiLink synchronization server uses to insert the new values of rows that were updated in the remote database, but conflict with values presently in the consolidated database.

Use statement-based events for uploads
This script has been deprecated. Use the statement-based event upload_-new_row_insert instead. Support for the new_row_cursor event is likely to be removed from future releases.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Item	Parameter	Description
1	ml_username	VARCHAR(128)

Default action

None.

Description

When a row is updated on a remote database, the MobiLink client saves a copy of the original values. The client sends both old and new values to the MobiLink synchronization server. Also used to input an INSERT operation in forced conflict mode.

When the MobiLink synchronization server receives an updated row, it compares the original values with the present values in the consolidated database, using the upload_cursor. If the old uploaded values do not match the current value in the consolidated database, the row conflicts. Instead of updating the row, the MobiLink synchronization server inserts both old and new values into the consolidated database using the old_row_cursor and the new_row_cursor, respectively.

The MobiLink synchronization server uses a cursor to insert the new uploaded values from conflicting rows into the consolidated database. This script contains the SELECT statement used to define this cursor.

It is common practice to use temporary tables to store the old and new versions of conflicting rows. You can create these temporary tables in an earlier script.

You can have one new_row_cursor script for each table in the remote database.

Normally, the columns in the select list must match those in the client table in both order and type. However, the MobiLink synchronization server permits you to add one extra column. If you do so, the MobiLink synchronization server automatically inserts the user name into the first column, then proceeds to insert the new row values using the remaining columns, as usual.

Note
The script is ignored if any of the following scripts are defined for the same table: upload_insert, upload_update, upload_delete, upload_fetch, upload_new_row_insert, upload_old_row_insert.

See also

- "upload_new_row_insert table event" on page 465
- "Handling conflicts" on page 64
- "resolve_conflict table event" on page 442

SQL example

The following SELECT statement defines a *new_row_cursor* script suited to the CustDB sample application.

```
call ml_add_table_script(
  'ver1',
  'table1',
  'new_row_cursor',
'SELECT order_id, cust_id, prod_id, emp_id,
    disc, quant, notes, status
  FROM ULNewOrder FOR update')
```

The primary key of the ULOrder table is order_id.

The following SELECT statement could instead be used for the same client table. This variation includes the permitted one extra row. The MobiLink synchronization server automatically stores the user name in the first column.

```
SELECT user_name, order_id, cust_id, prod_id,
    emp_id, disc, quant, notes, status
  FROM ULNewOrder FOR update
```

Java example

This script must return valid SQL.

The following stored procedure call registers a Java method called newRowCursor as the script for the new_row_cursor event when synchronizing the script version ver1.

```
call ml_add_java_table_script(
  'ver1',
  'table1',
  'new_row_cursor',
  'ExamplePackage.ExampleClass.newRowCursor' )
```

Following is the sample Java method newRowCursor. It dynamically generates a new row cursor statement by calling a Java method.

```
public String newRowCursor()
{  return( getRowCursor ( _curTable ) ); }
```

old_row_cursor table event (deprecated)

Function

Defines the cursor that the MobiLink synchronization server uses to insert the old values of rows that were updated in the remote database, but that conflict with values presently in the consolidated database. The event is also used to insert the values of deleted rows when in forced conflict mode.

Use statement-based events for uploads
This script has been deprecated. Use the statement-based event upload_-old_row_insert instead. Support for the old_row_cursor event is likely to be removed from future releases.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Item	Parameter	Description
1	ml_username	VARCHAR(128)

Default action

None.

Description

When a row is updated on a remote database, the MobiLink client saves a copy of the original values. The client sends both old and new values to the MobiLink synchronization server.

When the MobiLink synchronization server receives an updated row, it compares the original values with the present values in the consolidated database, using the upload_cursor event. If the old uploaded values do not match the current values in the consolidated database, the row conflicts. Instead of updating the row, the MobiLink synchronization server inserts both old and new values into the consolidated database using the old_row_cursor event and the new_row_cursor event.

It is common practice to use temporary tables to store the old and new versions of conflicting rows. In Adaptive Server Anywhere, you can create these tables in an earlier script. Some non-ASA consolidated databases support temporary tables, but they usually differ significantly from the temporary tables offered by ASA. Consult your DBMS documentation for details. An alternative to a temporary table is a base table with an extra column for the MobiLink user name. This effectively partitions the rows of the base table between concurrent synchronizations.

The MobiLink synchronization server uses a cursor to insert the old uploaded values from conflicting rows into the consolidated database. This script contains the SELECT statement used to define this cursor.

You can have one old_row_cursor script for each table in the remote database.

Normally, the columns in the SELECT list must match those in the client table in both order and type. However, the MobiLink synchronization server permits you to add one extra column. If you do so, the MobiLink synchronization server automatically inserts the user name into the first column, then proceeds to insert the old row values using the remaining columns, as usual.

See also

- "upload_old_row_insert table event" on page 467
- "Handling conflicts" on page 64
- "resolve_conflict table event" on page 442

SQL example

The following SELECT statement defines an *old_row_cursor* script suited to the CustDB sample application for an Oracle installation. The primary key of the ULOrder table is order_id.

```
call ml_add_table_script(
  'ver1',
  'table1',
  'old_row_cursor',
'SELECT order_id, cust_id, prod_id, emp_id,
    disc, quant, notes, status
 FROM ULOldOrder')
```

The following SELECT statement could instead be used for the same client table. This variation includes the permitted one extra row. The MobiLink synchronization server automatically stores the user name in the first column.

```
SELECT user_name, order_id, cust_id, prod_id,
    emp_id, disc, quant, notes, status
 FROM ULOldOrder
```

Java example

This script must return valid SQL.

The following stored procedure call registers a Java method called oldRowCursor as the script for the old_row_cursor event when synchronizing the script version ver1.

```
call ml_add_java_table_script(
  'ver1',
  'table1',
  'old_row_cursor',
  'ExamplePackage.ExampleClass.oldRowCursor' )
```

Following is the sample Java method oldRowCursor. It dynamically generates an old row cursor statement by calling a Java method.

```
public String oldRowCursor()
{  return( getRowCursor( _curTable ) ); }
```

prepare_for_download connection event

Function

Processes any required operations between the upload and download transactions.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Event parameters are optional only if no subsequent parameters are specified. You must use parameter 1 if you want to use parameter 2.

Item	Parameter	Description
1	last_download	TIMESTAMP
2	ml_username	VARCHAR(128)

Default action

None.

Description

The MobiLink synchronization server executes this script as a separate transaction, between the upload transaction and the start of the download transaction.

The last_download timestamp is the value obtained from the consolidated database during the last successful synchronization immediately prior to the download phase. If the current user has never synchronized successfully, this value is set to 1900-01-01.

See also

- "end_upload connection event" on page 402
- "begin_download connection event" on page 346

Java example

The following stored procedure call registers a Java method called prepareForDownload as the script for the prepare_for_download event when synchronizing the script version ver1.

```
call ml_add_java_connection_script(
   'ver1',
   'prepare_for_download',
   'ExamplePackage.ExampleClass.prepareForDownload' )
```

Following is the sample Java method prepareForDownload. It calls a Java method to modify some rows in the database.

```
public String prepareForDownload( Timestamp ts,
String user )
{  adjustUploadedRows( _syncConn, user );
   return( null ); }
```

.NET example

The following stored procedure call registers a .NET method called

PrepareForDownload as the script for the prepare_for_download connection event when synchronizing the script version ver1.

```
call ml_add_dnet_connection_script( ’ver1’,
  ’prepare_for_download’,
  ’TestScripts.Test.PrepareForDownload’
)
```

Following is the C# signature for the call PrepareForDownload.

```
public void PrepareForDownload(
  DateTime timestamp,
  string user )
```

report_error connection event

Function

Allows you to log errors and to record the actions selected by the handle_error script.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Event parameters are optional only if no subsequent parameters are specified. For example, you must use parameter 1 if you want to use parameter 2.

Item	Parameter	Description
1	action_code	INTEGER. This parameter is mandatory.
2	error_code	INTEGER. This parameter is optional if none of the following parameters are specified.
3	error_message	TEXT. This parameter is optional if none of the following parameters are specified.
4	ml_username	VARCHAR(128). This parameter is optional if none of the following parameters are specified.
5	table	VARCHAR(128). This parameter is optional.

Default action

None.

Description

This script allows you to log errors and to record the actions selected by the handle_error script. This script is executed after the handle_error event, whether or not a handle_error script is defined. It is always executed in its own transaction, on a different database connection than the synchronization connection (the administrative/information connection).

The error code and error message allow you to identify the nature of the error. The action code value is returned by the last call to an error handling script for the SQL operation that caused the current error.

If the error happened as part of synchronization, the user name is supplied. Otherwise, this value is NULL.

If the error happened while manipulating a particular table, the table name is supplied. Otherwise, this value is NULL. The table name is the name of a table in the remote database. This name may or may not have a direct counterpart in the consolidated database, depending on the design of the synchronization system.

See also

- "handle_error connection event" on page 415
- "handle_odbc_error connection event" on page 419
- "report_odbc_error connection event" on page 440

SQL example

The following example works with an Adaptive Server Anywhere consolidated database. It inserts a row into a table used to record synchronization errors.

```
call ml_add_connection_script(
 'ver1',
 'report_error',
 'insert into sync_error(
   action_code,
   error_code,
   error_message,
   user_name,
   table_name )
    values( ?, ?, ?, ?, ? )'  )
```

Java example

The following stored procedure call registers a Java method called reportError as the script for the report_error connection event when synchronizing the script version ver1.

```
call ml_add_java_connection_script(
   'ver1',
   'report_error,
   'ExamplePackage.ExampleClass.reportError' )
```

Following is the sample Java method reportError. It logs the error to a table using the JDBC connection provided by MobiLink. It also sets the action code.

```
public String reportError(
ianywhere.ml.script.InOutInteger actionCode,
int errorCode, String errorMessage, String user,
String table )
  throws java.sql.SQLException
{  // insert error information in a table
   JDBCLogError( _syncConn, errorCode, errorMessage,
   user, table );
   actionCode.setValue( getActionCode( errorCode ) );
   return( null ); }
```

report_odbc_error connection event

Function

Allows you to log errors and to record the actions selected by the handle_odbc_error script.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See “SQL-Java data types” on page 260 and “SQL-.NET data types” on page 287.

Event parameters are optional only if no subsequent parameters are specified. For example, you must use parameter 1 if you want to use parameter 2.

Item	Parameter	Description
1	action_code	INTEGER. This parameter is mandatory.
2	ODBC_state	VARCHAR(5). This parameter is optional if none of the following parameters are specified.
3	error_message	TEXT. This parameter is optional if none of the following parameters are specified.
4	ml_username	VARCHAR(128). This parameter is optional if none of the following parameters are specified.
5	table	VARCHAR(128). This parameter is optional.

Default action

None.

Description

This script allows you to log errors and to record the actions selected by the handle_odbc_error script. This script is executed after the handle_odbc_error event, whether or not a handle_odbc_error script is defined. It is always executed in its own transaction, on a different database connection than the synchronization connection (the administrative/information connection).

The error code and error message allow you to identify the nature of the error. The action code value is returned by the last call to an error handling script for the SQL operation that caused the current error.

If the error happened as part of synchronization, the user name is supplied. Otherwise, this value is NULL.

If the error happened while manipulating a particular table, the table name is supplied. Otherwise, this value is NULL. The table name is the name of a table in the remote database. This name may or may not have a direct counterpart in the consolidated database, depending on the design of the synchronization system.

See also

- "handle_error connection event" on page 415
- "handle_odbc_error connection event" on page 419
- "report_error connection event" on page 438

SQL example

The following example works with an Adaptive Server Anywhere consolidated database. It inserts a row into a table used to record synchronization errors.

```
call ml_add_connection_script(
 'ver1',
 'report_odbc_error',
 'insert into sync_error(
    action_code,
    odbc_state,
    error_message,
    user_name,
    table_name )
  values( ?, ?, ?, ?, ? )'  )
```

Java example

The following stored procedure call registers a Java method called reportODBCError as the script for the report_odbc_error event when synchronizing the script version ver1.

```
call ml_add_java_connection_script(
   'ver1',
   'report_odbc_error',
   'ExamplePackage.ExampleClass.reportODBCError' )
```

Following is the sample Java method reportODBCError. It logs the error to a table using the JDBC connection provided by MobiLink. It also sets the action code.

```
public String reportODBCError(
   ianywhere.ml.script.InOutInteger actionCode,
   String ODBCState,
   String errorMessage,
   String user,
   String table )
  throws java.sql.SQLException
{  JDBCLogError( _syncConn, ODBCState, errorMessage,
   user, table );
   actionCode.setValue( getActionCode( ODBCState ) );
   return( null ); }
```

resolve_conflict table event

Function

Defines a process for resolving a conflict in a specific table.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Event parameters are optional only if no subsequent parameters are specified. You must use parameter 1 if you want to use parameter 2.

Item	Parameter	Description
1	ml_username	VARCHAR(128)
2	table	VARCHAR(128)

Default action

None.

Description

When a row is updated on a remote database, the MobiLink client saves a copy of the original values. The client sends both old and new values to the MobiLink synchronization server.

When the MobiLink synchronization server receives an updated row, it compares the original values with the present values in the consolidated database. The comparison is carried out using the upload_fetch script.

If the old uploaded values do not match the current values in the consolidated database, the row conflicts. Instead of updating the row, the MobiLink synchronization server inserts both old and new values into the consolidated database. The old and new rows are handled using the upload_old_row_insert and upload_new_row_insert scripts, respectively. If you are using cursor-based uploads the rows are handled using old_row_cursor and new_row_cursor, respectively.

Once the values have been inserted, the MobiLink synchronization server executes the resolve_conflict script. It provides the opportunity to resolve the conflict. You can implement any scheme of your choosing.

This script is executed once per conflict.

Alternatively, instead of defining the resolve_conflict script, you can resolve conflicts in a set-oriented fashion by putting conflict-resolution logic either in your end_upload_rows script or in your end_upload table script.

You can have one resolve_conflict script for each table in the remote database.

See also

- "upload_old_row_insert table event" on page 467

- "upload_new_row_insert table event" on page 465
- "upload_update table event" on page 475
- "old_row_cursor table event (deprecated)" on page 434
- "new_row_cursor table event (deprecated)" on page 431
- "end_upload_rows table event" on page 409

SQL example

The following statement defines a *resolve_conflict* script suited to the CustDB sample application for an Oracle installation. It calls a stored procedure *ULResolveOrderConflict* .

```
exec ml_add_table_script(
   'custdb', 'ULOrder', 'resolve_conflict',
   'begin ULResolveOrderConflict();
end; ')

CREATE OR REPLACE PROCEDURE ULResolveOrderConflict()
AS
  new_order_id integer;
  new_status   varchar(20);
  new_notes   varchar(50);
BEGIN
  -- approval overrides denial
  SELECT order_id, status, notes
    INTO new_order_id, new_status, new_notes
    FROM ULNewOrder
   WHERE syncuser_id = SyncUserID;
  IF new_status = 'Approved' THEN
    UPDATE ULOrder o
       SET o.status = new_status, o.notes =
new_notes
      WHERE o.order_id = new_order_id;
  END IF;
  DELETE FROM ULOldOrder;
  DELETE FROM ULNewOrder;
END;
```

Java example

The following stored procedure call registers a Java method called resolveConflict as the script for the resolve_conflict table event when synchronizing the script version ver1.

```
call ml_add_java_table_script(
   'ver1',
   'table1',
   'resolve_conflict',
   'ExamplePackage.ExampleClass.resolveConflict' )
```

Following is the sample Java method resolveConflict. It calls a Java method that will use the JDBC connection provided by MobiLink. It also sets the action code.

```
public String resolveConflict( String user,
 String table)
{  resolveRows(_syncConn, user ); }
```

synchronization_statistics connection event

Function

Tracks synchronization statistics.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Event parameters are optional only if no subsequent parameters are specified. For example, you must use parameter 1 if you want to use parameter 2.

Item	Parameter	Description
1	ml_username	VARCHAR(128)
2	warnings	INTEGER
3	errors	INTEGER
4	deadlocks	INTEGER
5	synchronized_tables	INTEGER
6	connection_retries	INTEGER

Default action

None.

Description

The synchronization_statistics event allows you to gather, for any user and connection, various statistics about the current synchronization. The synchronization_statistics connection script is called just prior to the commit at the end of the end synchronization transaction.

See also

- "download_statistics connection event" on page 378
- "download_statistics table event" on page 381
- "upload_statistics connection event" on page 469
- "upload_statistics table event" on page 472
- "synchronization_statistics table event" on page 448
- "time_statistics connection event" on page 450
- "time_statistics table event" on page 453
- "MobiLink Monitor" on page 117

SQL example

The following example inserts synchronization statistics into the sync_con_audit table.

```
call ml_add_connection_script(
  'ver1',
  'synchronization_statistics',
'INSERT INTO sync_con_audit(
  ml_user, warnings, errors,
  deadlocks, synchronized_tables,
  connection_retries)
VALUES (?,?,?,?,?,?)')
```

Once statistics are inserted into the audit table, you may use these statistics to monitor your synchronizations and make optimizations where applicable.

Java example

The following stored procedure call registers a Java method called synchronizationStatisticsConnection as the script for the synchronization_statistics connection event when synchronizing the script version ver1.

```
call ml_add_java_connection_script(
  'ver1',
  'synchronization_statistics',
  'ExamplePackage.ExampleClass.synchronizationStatisticsConnecti
        on'
)
```

Following is the sample Java method synchronizationStatisticsConnection. It logs some of the statistics to the MobiLink output log. (This might be useful at development time but would slow down a production server.)

```
public String synchronizationStatisticsConnection(
  String user, int warnings, int errors, int deadlocks,
  int synchronizedTables, int connectionRetries )
{  java.lang.System.out.println( "synch statistics
   number of deadlocks: " + deadlocks ;
   return( null ); }
```

.NET example

The following stored procedure call registers a .NET method called SyncStats as the script for the synchronization_statistics connection event when synchronizing the script version ver1.

```
call ml_add_dnet_connection_script(
  'ver1',
  'synchronization_statistics',
  'TestScripts.Test.SyncStats'
)
```

Following is the sample .NET method SyncStats. It logs some of the statistics to the MobiLink output log. (This might be useful at development time but would slow down a production server.)

```
public void SyncStats(
  string user,
  int warnings,
  int errors,
  int deadLocks,
  int syncedTables,
  int connRetries )
{  System.Console.WriteLine( "synch statistics
   number of deadlocks: " + deadlocks ;
   return( null ); }
```

synchronization_statistics table event

Function

Tracks synchronization statistics.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Event parameters are optional only if no subsequent parameters are specified. For example, you must use parameter 1 if you want to use parameter 2.

Item	Parameter	Description
1	ml_username	VARCHAR(128)
2	table	VARCHAR(128)
3	warnings	INTEGER
4	errors	INTEGER

Default action

None.

Description

The synchronization_statistics event allows you to gather, for any user and table, the number of warnings and errors that occurred during synchronization. The synchronization_statistics table script is called just prior to the commit at the end of the end synchronization transaction.

See also

- "download_statistics connection event" on page 378
- "download_statistics table event" on page 381
- "upload_statistics connection event" on page 469
- "upload_statistics table event" on page 472
- "synchronization_statistics connection event" on page 445
- "time_statistics connection event" on page 450
- "time_statistics table event" on page 453
- "MobiLink Monitor" on page 117

SQL example

The following example inserts synchronization statistics into the sync_tab_audit table.

```
call ml_add_java_table_script(
  'ver1',
  'table1',
'INSERT INTO sync_tab_audit ( ml_user, table,
warnings, errors) VALUES (?,?,?,?)')
```

Once synchronization statistics are inserted into the audit table, you may use these statistics to monitor your synchronizations and make optimizations where applicable.

Java example

The following stored procedure call registers a Java method called synchronizationStatisticsTable as the script for the synchronization_statistics table event when synchronizing the script version ver1.

```
call ml_add_java_table_script(
  'ver1',
  'table1',
  'synchronization_statistics',
  'ExamplePackage.ExampleClass.synchronizationStatisticsTable'
)
```

Following is the sample Java method synchronizationStatisticsTable. It logs some of the statistics to the MobiLink output log. (This might be useful at development time but would slow down a production server.)

```
public String synchronizationStatisticsTable(
String user, String table, int warnings, int errors )
{  java.lang.System.out.println( "synch statistics for
   table: " + table + " errors: " + errors );
   return( null ); }
```

.NET example

The following stored procedure call registers a .NET method called SyncTableStats as the script for the synchronization_statistics table event when synchronizing the script version ver1 and the table table1.

```
call ml_add_dnet_table_script(
  'ver1',
  'table1',
  'synchronization_statistics',
  'TestScripts.Test.SyncTableStats'
)
```

Following is the sample .NET method SyncTableStats. It logs some of the statistics to the MobiLink output log. (This might be useful at development time but would slow down a production server.)

```
public void SyncTableStats(
  string user,
  string table,
  int warnings,
  int errors )
{  System.Console.WriteLine( "synch statistics for
   table: " + table + " errors: " + errors );
   return( null ); }
```

time_statistics connection event

Function

Tracks time statistics by user and event.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Event parameters are optional only if no subsequent parameters are specified. For example, you must use parameter 1 if you want to use parameter 2.

Item	Parameter	Description
1	ml_username	VARCHAR(128)
2	event_name	VARCHAR(128)
3	num_calls	INTEGER
4	min_time	INTEGER. Milliseconds.
5	max_time	INTEGER. Milliseconds.
6	total_time	INTEGER. Milliseconds.

Default action

None.

Description

The time_statistics event allows you to gather time statistics for any user during synchronization. The statistics are gathered only for those events for which there is a corresponding script. The script gathers aggregate data for occasions where a single event occurs multiple times. The script can be especially useful for time comparisons across users, events and tables.

See also

- ♦ "time_statistics table event" on page 453
- ♦ "download_statistics connection event" on page 378
- ♦ "download_statistics table event" on page 381
- ♦ "upload_statistics connection event" on page 469
- ♦ "upload_statistics table event" on page 472
- ♦ "synchronization_statistics connection event" on page 445
- ♦ "synchronization_statistics table event" on page 448
- ♦ "MobiLink Monitor" on page 117

SQL example

The following example inserts statistical information into the time_statistics table.

```
call ml_add_connection_script(
  'ver1',
  'time_statistics',
'INSERT INTO time_statistics (id, ml_user, table,
     event_name, num_calls, min_time, max_time, total_time)
   VALUES (ts_id.nextval,?,?,?,?,?,?)')
```

Java example

The following stored procedure call registers a Java method called timeStatisticsConnection as the script for the time_statistics connection event when synchronizing the script version ver1.

```
call ml_add_java_connection_script(
  'ver1',
  'time_statistics',
  'ExamplePackage.ExampleClass.timeStatisticsConnection' )
```

Following is the sample Java method timeStatisticsConnection. It prints statistics for the prepare_for_download event. (This might be useful at development time but would slow down a production server.)

```
public void timeStatisticsConnection(
  String ml_username,
  String table_name,
  String event_name,
int num_calls, int min_time, int max_time,
int total_time )
{  if( event_name.equals( "prepare_for_download")
   {  java.lang.System.out.println(
      "prepare_for_download num_calls: " + num_calls +
      "total_time: " + total_time ); } }
```

.NET example

The following stored procedure call registers a .NET method called TimeStats as the script for the time_statistics connection event when synchronizing the script version ver1.

```
call ml_add_dnet_connection_script(
  'ver1',
  'time_statistics',
  'TestScripts.Test.TimeStats'
)
```

Following is the sample Java method TimeStats. It prints statistics for the prepare_for_download event. (This might be useful at development time but would slow down a production server.)

```
public void TimeStats(
  string user,
  string eventName,
  int numCalls,
  int minTime,
  int maxTime,
  int totTime )
{  if( event_name=="prepare_for_download")
   {  System.Console.WriteLine(
      "prepare_for_download num_calls: " + num_calls +
      "total_time: " + total_time ); } }
```

time_statistics table event

Function	Tracks time statistics.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Event parameters are optional only if no subsequent parameters are specified. For example, you must use parameter 1 if you want to use parameter 2.

Item	Parameter	Description
1	ml_username	VARCHAR(128)
2	table	VARCHAR(128)
3	event_name	VARCHAR(128)
4	num_calls	INTEGER
5	min_time	INTEGER. Milliseconds.
6	max_time	INTEGER. Milliseconds.
7	total_time	INTEGER. Milliseconds.

Default action

None.

Description

The time_statistics table event allows you to gather time statistics for any user and table during synchronization. The statistics are gathered only for those events for which there is a corresponding script. The script gathers aggregate data for occasions where a single event occurs multiple times. The script can be especially useful for time comparisons across users, events and tables.

See also

- "time_statistics connection event" on page 450
- "download_statistics connection event" on page 378
- "download_statistics table event" on page 381
- "upload_statistics connection event" on page 469
- "upload_statistics table event" on page 472
- "synchronization_statistics connection event" on page 445
- "synchronization_statistics table event" on page 448
- "MobiLink Monitor" on page 117

SQL example

The following example inserts statistical information into the time_statistics table.

```
INSERT INTO time_statistics(
 ml_user, table, event_name, num_calls,
 min_time, max_time, total_time)
VALUES (?,?,?,?,?,?,?)
```

Java example

The following stored procedure call registers a Java method called timeStatisticsTable as the script for the time_statistics table event when synchronizing the script version ver1.

```
call ml_add_java_table_script(
  'ver1',
  'table1',
  'time_statistics',
  'ExamplePackage.ExampleClass.timeStatisticsTable' )
```

Following is the sample Java method timeStatisticsTable. It prints statistics for the upload_old_row_insert event.

```
public void timeStatisticsConnection(
   String ml_username,
   String table_name,
   String event_name,
int num_calls, int min_time, int max_time,
int total_time )
{  if( event_name.equals( "upload_old_row_insert")
   {  java.lang.System.out.println(
      "upload_old_row_insert num_calls: " + num_calls +
      "total_time: " + total_time ); } }
```

.NET example

The following stored procedure call registers a .NET method called TimeTableStats as the script for the time_statistics table event when synchronizing the script version ver1 and the table table1.

```
call ml_add_dnet_table_script(
  'ver1',
  'table1',
  'time_statistics',
  'TestScripts.Test.TimeTableStats'
)
```

Following is the sample .NET method TimeTableStats. It prints statistics for the upload_old_row_insert event.

```
public void TimeTableStats(
  string user,
  string table,
  string eventName,
  int numCalls,
  int minTime,
  int maxTime,
  int totTime )
{  if( event_name == "upload_old_row_insert")
   {  System.Console.WriteLine(
      "upload_old_row_insert num_calls: " + num_calls +
      "total_time: " + total_time ); } }
```

upload_cursor table event (deprecated)

Function

Defines a cursor that the MobiLink synchronization server uses to insert, update, or delete rows during processing of the upload stream.

Use statement-based events for uploads
This script has been deprecated. Use the statement-based events upload_-delete, upload_insert, and upload_update instead of the upload_cursor event to process the upload stream. Support for the upload_cursor event is likely to be removed from future releases.

Parameters

Item	Parameter
1	primary key 1
2	primary key 2
…	…

Default action

None.

Description

The MobiLink synchronization server opens a cursor with which to insert, update, or delete rows in the consolidated database based on rows uploaded from a client application. This script should contain a suitable SELECT statement or call a stored procedure that contains a suitable SELECT statement.

The parameters are the values of each column included in the primary key of the corresponding client table. You must use these in a WHERE clause, so that the synchronization can identify a unique row based on these values. The type and order of the parameters is as defined in the example_upload_cursor script. This order is the same as that in the corresponding table definition in the remote database, which in turn may have been copied from your reference database.

You can have one upload_cursor script for each table in the remote database.

For Java and .NET applications, this script must return valid SQL.

See also

- "Writing scripts to upload rows" on page 244
- "upload_delete table event" on page 459
- "upload_insert table event" on page 463
- "upload_update table event" on page 475

SQL example

The following SELECT statement defines the upload cursor in the CustDB sample application.

```
call ml_add_table_script(
  'ver1',
  'table1',
  'upload_cursor',
'SELECT cust_id, cust_name
FROM ULCustomer
WHERE cust_id = ?')
```

The primary key of the ULCustomer table in the CustDB sample application is the column cust_id. If the corresponding table in the consolidated database is, instead, named Customer, then change the above statement as follows.

```
SELECT cust_id, cust_name
FROM Customer
WHERE cust_id = ?
```

Java example

The following stored procedure call registers a Java method called uploadCursor as the script for the upload_cursor event when synchronizing the script version ver1.

```
call ml_add_java_table_script(
  'ver1',
  'table1',
  'upload_cursor',
  'ExamplePackage.ExampleClass.uploadCursor' )
```

Following is the sample Java method uploadCursor. It calls getUploadCursor to dynamically generate an upload cursor.

```
public String uploadCursor()
{ return( getUploadCursor( _curTable ) ); }
```

.NET example

The following stored procedure call registers a .NET method called UploadCursor as the script for the upload_cursor event when synchronizing the script version ver1.

```
call ml_add_dnet_table_script(
  'ver1',
  'table1',
  'upload_cursor',
  'TestScripts.Test.UploadCursor' )
```

The following C# example deletes the contents of a temporary table. It then returns SQL that causes rows to be uploaded into the temporary table.

```
public string UploadCursor()
{
    DBCommand stmt = curConn.CreateCommand();
    stmt.CommandText = "DELETE FROM dnet_ul_temp";
    stmt.ExecuteNonQuery();
    stmt.Close();

    return( "SELECT * FROM dnet_ul_temp WHERE pk = ?" );
}
```

upload_delete table event

Function Provides an event that the MobiLink synchronization server uses during processing of the upload stream to handle rows deleted from the remote database.

Parameters

Item	Parameter
1	primary key 1
2	primary key 2
…	…

Default action None.

Description The statement-based upload_delete script handles rows that are deleted on the remote database. The action taken at the consolidated database can be a DELETE statement, but need not be.

You can have one upload_delete script for each table in the remote database.

For Java and .NET applications, this script must return valid SQL.

See also

- "upload_insert table event" on page 463
- "upload_update table event" on page 475

SQL example This example is taken from the Contact sample and can be found in *Samples\MobiLink\Contact\build_consol.sql*. It marks customers that are deleted from the remote database as inactive.

```
call ml_add_table_script(
  'ver1',
  'table1',
  'upload_delete',
'UPDATE Customer SET active = 0 WHERE cust_id=?')
```

Java example The following stored procedure call registers a Java method called uploadDeleteTable as the script for the upload_delete table event when synchronizing the script version ver1.

```
call ml_add_java_table_script(
  'ver1',
  'table1',
  'upload_delete',
  'ExamplePackage.ExampleClass.uploadDeleteTable' )
```

Following is the sample Java method uploadDeleteTable. It calls genUD which dynamically generates an UPLOAD statement.

```
public string uploadDeleteTable()
{  return( genUD(_curTable) ); }
```

.NET example

The following stored procedure call registers a .NET method called UploadDelete as the script for the upload_delete table event when synchronizing the script version ver1 and the table table1.

```
call ml_add_dnet_table_script(
  'ver1',
  'table1',
  'upload_delete',
  'TestScripts.Test.UploadDelete'
)
```

Following is the sample .NET method UploadDelete. It calls genUD which dynamically generates an UPLOAD statement.

```
public string UploadDelete( object pk1 )
{  return( genUD(_curTable) ); }
```

upload_fetch table event

Function

Provides an event that the MobiLink synchronization server uses to identify update conflicts during statement-based processing of the upload stream.

Parameters

Item	Parameter
1	primary key 1
2	primary key 2
…	…

Default action

None.

Description

The statement-based upload_fetch script fetches rows from a synchronized table for the purposes of conflict detection. It is a companion to the upload_update event.

The columns of the result set must match the number of columns being uploaded from the remote database for this table. If the values returned do not match the pre-image in the uploaded row, a conflict is identified.

You can have one upload_fetch script for each table in the remote database.

This script may be ignored if none of the following scripts are defined: upload_new_row_insert, upload_old_row_insert, and resolve_conflict.

See also

- "resolve_conflict table event" on page 442
- "upload_delete table event" on page 459
- "upload_insert table event" on page 463
- "upload_update table event" on page 475

SQL example

The following SQL script is taken from the Contact sample and can be found in *Samples\MobiLink\Contact\build_consol.sql* in the SQL Anywhere installation. It is used to identify conflicts that occur when rows updated in the remote database Product table are uploaded. This script selects rows from a table also named Product, but depending on your consolidated and remote database schema, the two table names may not match.

```
call ml_add_table_script(
  'ver1',
  'table1',
  'upload_fetch',
'SELECT id, name, size, quantity, unit_price
FROM Product WHERE id=?')
```

Java example

This script must return valid SQL.

The following stored procedure call registers a Java method called uploadFetchTable as the script for the upload_fetch table event when synchronizing the script version ver1.

```
call ml_add_java_table_script(
  ’ver1’,
  ’table1’,
  ’upload_fetch’,
  ’ExamplePackage.ExampleClass.uploadFetchTable’ )
```

Following is the sample Java method uploadFetchTable. It calls genUF to dynamically generate an UPLOAD statement.

```
public string uploadFetchTable()
{  return( genUF(_curTable) ); }
```

upload_insert table event

Function

Provides an event that the MobiLink synchronization server uses during processing of the upload stream to handle rows inserted into the remote database.

Parameters

Item	Parameter
1	column 1
2	column 2
…	…

Default action

None.

Description

The statement based upload_insert script performs direct inserts of column values.

You can have one upload_insert script for each table in the remote database.

For Java and .NET applications, this script must return valid SQL.

See also

- ♦ “upload_delete table event” on page 459
- ♦ “upload_update table event” on page 475
- ♦ “upload_fetch table event” on page 461

SQL example

This example handles inserts that were made on the Customer table in the remote database. The script inserts the values into a table named Customer in the consolidated database. The final column of the table identifies the Customer as active. The final column does not appear in the remote database.

```
call ml_add_table_script(
  'ver1',
  'Customer',
  'upload_insert',
'INSERT INTO Customer( cust_id, name, rep_id, active )
VALUES ( ?, ?, ?, 1 )')
```

Java example

The following stored procedure call registers a Java method called uploadInsertTable as the script for the upload_insert table event when synchronizing the script version ver1.

```
call ml_add_java_table_script(
  'ver1',
  'table1',
  'upload_insert',
  'ExamplePackage.ExampleClass.uploadInsertTable' )
```

Following is the sample Java method uploadInsertTable. It dynamically generates an UPLOAD statement. This syntax is for Adaptive Server Anywhere consolidated databases.

```
public string uploadInsertTable()
{  return("insert into" + _curTable + getCols(_curTable)
   + "values" + getQM(_curTable)); }
```

.NET example

The following stored procedure call registers a .NET method called UploadInsert as the script for the upload_insert table event when synchronizing the script version ver1 and the table table1. This syntax is for Adaptive Server Anywhere consolidated databases.

```
call ml_add_dnet_table_script(
  'ver1',
  'table1',
  'upload_insert',
  'TestScripts.Test.UploadInsert'
)
```

Following is the sample .NET method UploadInsert. It returns an UPLOAD statement for the ULCustomer table.

```
public static string UploadInsert()
 {
   return("INSERT INTO ULCustomer(cust_id,cust_name) values
         (?,?)");
 }
```

upload_new_row_insert table event

Function

Conflict resolution scripts for statement-based uploads commonly require access to the old and new values of rows uploaded from the remote database. This event allows you to handle the new, updated values of rows uploaded from the remote database.

Parameters

Item	Parameter
ml_username	VARCHAR(128). This parameter is optional.
1	column 1
2	column 2
…	…

Default action

None.

Description

When a MobiLink client sends an updated row to the MobiLink synchronization server, it includes not only the new values (the post-image), but also a copy of the old row values (the pre-image). When the pre-image does not match the current values in the consolidated database, a conflict is detected.

This event allows you to save post-image values to a table. You can use this event to assist in developing conflict resolution procedures for statement-based updates. The parameters for this event hold new row values from the remote database before the update is carried out on the corresponding consolidated database table. This event is also used to insert rows in statement-based, forced-conflict mode.

The script for this event is usually an insert statement that inserts the new row into a temporary table for use by a resolve_conflict script.

You can have one upload_new_row_insert script for each table in the remote database.

For Java and .NET applications, this script must return valid SQL.

See also

- ♦ "Handling conflicts" on page 64
- ♦ "resolve_conflict table event" on page 442
- ♦ "upload_old_row_insert table event" on page 467
- ♦ "upload_update table event" on page 475
- ♦ "Storing the user name" on page 71

SQL example

This example handles updates made on the product table in the remote database. The script inserts the new value of the row into a global temporary table named product_conflict. The final column of the table identifies the row as a new row.

```
call ml_add_table_script(
  'ver1',
  'table1',
  'upload_new_row_insert',
'INSERT INTO DBA.product_conflict(
  id, name, size, quantity, unit_price, row_type )
VALUES( ?, ?, ?, ?, ?, 'N' )')
```

Java example

The following stored procedure call registers a Java method called uploadNewRowInsertTable as the script for the upload_new_row_insert table event when synchronizing the script version ver1.

```
call ml_add_java_table_script(
  'ver1',
  'table1',
  'upload_new_row_insert',
  'ExamplePackage.ExampleClass.uploadNewRowInsertTable'
)
```

Following is the sample Java method uploadNewRowInsertTable. It dynamically generates an UPLOAD statement. This syntax is for Adaptive Server Anywhere consolidated databases.

```
public string uploadNewRowInsertTable()
{  return("insert into" + _curTable + "_new" +
   getCols(_curTable) + "values" + getQM(_curTable)); }
```

upload_old_row_insert table event

Function

Conflict resolution scripts for statement-based uploads commonly require access to the old and new values of rows uploaded from the remote database. This event allows you to handle the old values of rows uploaded from the remote database.

Parameters

Item	Parameter
ml_username	VARCHAR(128). This parameter is optional.
1	column 1
2	column 2
…	…

Default action

None.

Description

When a MobiLink client sends an updated row to the MobiLink synchronization server, it includes not only the new values (the post-image), but also a copy of the old row values (the pre-image). When the pre-image does not match the current values in the consolidated database, a conflict is detected.

This event allows you to save pre-image values to a table. You can use this event to assist in developing conflict resolution procedures for statement-based updates. The parameters for this event hold old row values from the remote database before the update is carried out on the corresponding consolidated database table. This event is also used to insert rows in statement-based, forced-conflict mode.

The script for this event is usually an insert statement that inserts the old row into a temporary table for use by a resolve_conflict script.

You can have one upload_old_row_insert script for each table in the remote database.

For Java and .NET applications, this script must return valid SQL.

See also

- ♦ "Handling conflicts" on page 64
- ♦ "resolve_conflict table event" on page 442
- ♦ "upload_new_row_insert table event" on page 465
- ♦ "upload_update table event" on page 475
- ♦ "Storing the user name" on page 71

SQL example

This example handles updates made on the product table in the remote database. The script inserts the old value of the row into a global temporary table named product_conflict. The final column of the table identifies the row as an old row.

```
call ml_add_table_script(
  ’ver1’,
  ’table1’,
  ’upload_old_row_insert’,
’insert into DBA.product_conflict(
id, name, size, quantity, unit_price, row_type )
values( ?, ?, ?, ?, ?, ’O’ )’)
```

Java example

The following stored procedure call registers a Java method called uploadOldRowInsertTable as the script for the upload_old_row_insert table event when synchronizing the script version ver1.

```
call ml_add_java_table_script(
  ’ver1’,
  ’table1’,
  ’upload_old_row_insert’,
  ’ExamplePackage.ExampleClass.uploadNewRowInsertTable’
)
```

Following is the sample Java method uploadOldRowInsertTable. It dynamically generates an UPLOAD statement.

```
public string uploadOldRowInsertTable()
{  old" + getCols(_curTable) +
   "values" + getQM(_curTable)); }
```

upload_statistics connection event

Function

Tracks synchronization statistics for upload operations.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Event parameters are optional only if no subsequent parameters are specified. For example, you must use parameter 1 if you want to use parameter 2.

Item	Parameter	Description
1	ml_username	VARCHAR(128)
2	warnings	INTEGER
3	errors	INTEGER
4	inserted_rows	INTEGER
5	deleted_rows	INTEGER
6	updated_rows	INTEGER
7	conflicted_inserts	INTEGER
8	conflicted_deletes	INTEGER
9	conflicted_updates	INTEGER
10	ignored_inserts	INTEGER
11	ignored_deletes	INTEGER
12	ignored_updates	INTEGER
13	bytes	INTEGER
14	deadlocks	INTEGER

Default action

None.

Description

The upload_statistics event allows you to gather, for any user, statistics on uploads. The upload_statistics connection script is called just prior to the commit at the end of the upload transaction.

See also

- "download_statistics connection event" on page 378
- "download_statistics table event" on page 381

- “upload_statistics table event” on page 472
- “synchronization_statistics connection event” on page 445
- “synchronization_statistics table event” on page 448
- “time_statistics connection event” on page 450
- “time_statistics table event” on page 453
- “MobiLink Monitor” on page 117

SQL example

The following example inserts synchronization statistics for upload operations into the table upload_summary_audit.

```
call ml_add_connection_script(
   ’ver1’,
   ’upload_statistics’,
’INSERT INTO upload_summary_audit (
 ml_user, warnings, errors, inserted_rows,
 deleted_rows, updated_rows, conflicted_inserts,
 conflicted_deletes, conflicted_updates,
 bytes, ignored_inserts, ignored deletes,
 ignored_updates, bytes, deadlocks)
VALUES (?,?,?,?,?,?,?,?,?,?,?,?,?,?,?)’
```

Once statistics are inserted into the audit table, you may use these statistics to monitor your synchronizations and make optimizations where applicable.

Java example

The following stored procedure call registers a Java method called uploadStatisticsConnection as the script for the upload_statistics connection event when synchronizing the script version ver1.

```
call ml_add_java_connection_script(
   ’ver1’,
   ’upload_statistics’,
   ’ExamplePackage.ExampleClass.uploadStatisticsConnection’ )
```

Following is the sample Java method uploadStatisticsConnection. It logs some statistics to the MobiLink output log. (This might be useful at development time but would slow down a production server.)

```
public String uploadStatisticsConnection(
  String user,
  int warnings,
  int errors,
  int insertedRows,
  int deletedRows,
  int updatedRows,
  int conflictedInserts,
  int conflictedDeletes,
  int conflictedUpdates,
  int ignoredInserts,
  int ignoredDeletes,
  int ignoredUpdates,
  int bytes,
  int deadlocks
)
{  java.lang.System.out.println( "updated rows: " +
   updatedRows ); }
```

.NET example

The following stored procedure call registers a .NET method called UploadStats as the script for the upload_statistics connection event when synchronizing the script version ver1.

```
call ml_add_dnet_connection_script(
  'ver1',
  'upload_statistics',
  'TestScripts.Test.UploadStats'
)
```

Following is the sample .NET method UploadStats. It logs some statistics to the MobiLink output log. (This might be useful at development time but would slow down a production server.)

```
public void UploadStats(
  string user,
  int warnings,
  int errors,
  int insertedRows,
  int deletedRows,
  int updatedRows,
  int conflictInserts,
  int conflictDeletes,
  int conflictUpdates,
  int ignoredInserts,
  int ignoredDeletes,
  int ignoredUpdates,
  int bytes,
  int deadlocks)
{  System.Console.WriteLine( "updated rows: " +
   updatedRows ); }
```

upload_statistics table event

Function

Tracks synchronization statistics for upload operations for a specific table.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 260 and "SQL-.NET data types" on page 287.

Event parameters are optional only if no subsequent parameters are specified. For example, you must use parameter 1 if you want to use parameter 2.

Item	Parameter	Description
1	ml_username	VARCHAR(128)
2	table	VARCHAR(128)
3	warnings	INTEGER
4	errors	INTEGER
5	inserted_rows	INTEGER
6	deleted_rows	INTEGER
7	updated_rows	INTEGER
8	conflicted_inserts	INTEGER
9	conflicted_deletes	INTEGER
10	conflicted_updates	INTEGER
11	ignored_inserts	INTEGER
12	ignored_deletes	INTEGER
13	ignored_updates	INTEGER
14	bytes	INTEGER
15	deadlocks	INTEGER

Default action

None.

Description

The upload_statistics event allows you to gather, for any user, vital statistics on synchronization happenings as they apply to any table. The upload_statistics table script is called just prior to the commit at the end of the upload transaction.

See also

- ♦ "download_statistics connection event" on page 378
- ♦ "upload_statistics connection event" on page 469
- ♦ "upload_statistics table event" on page 472
- ♦ "synchronization_statistics connection event" on page 445
- ♦ "synchronization_statistics table event" on page 448
- ♦ "time_statistics connection event" on page 450
- ♦ "time_statistics table event" on page 453
- ♦ "MobiLink Monitor" on page 117

SQL Example

The following example inserts a row into a table used to track upload statistics.

```
call ml_add_connection_script(
 'ver1',
 'upload_statistics',
 'INSERT INTO my_upload_statistics
    ( user_name, table_name, num_warnings, num_errors,
      inserted_rows, deleted_rows, updated_rows,
      conflicted_inserts, conflicted_deletes,
      conflicted_updates, ignored_inserts,
      ignored_deletes, ignored_updates, bytes,
      deadlocks )
    VALUES( ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ? )'  )
```

The following example works with an Oracle consolidated database.

```
INSERT INTO upload_tables_audit (
id, user_name, table, warnings, errors,
inserted_rows, deleted_rows, updated_rows,
conflicted_inserts, conflicted_deletes,
conflicted_updates, ignored_inserts, ignored_deletes,
ignored_updates, bytes, deadlocks)
VALUES ( ut_audit.nextval,
?,?,?,?,?,?,?,?,?,?,?,?,?,?,?)
```

Once statistics are inserted into the audit table, you may use these statistics to monitor your synchronizations and make optimizations where applicable.

Java example

The following stored procedure call registers a Java method called uploadStatisticsTable as the script for the upload_statistics table event when synchronizing the script version ver1.

```
call ml_add_java_table_script(
  'ver1',
  'table1',
  'upload_statistics',
  'ExamplePackage.ExampleClass.uploadStatisticsTable' )
```

Following is the sample Java method uploadStatisticsTable. It logs some statistics to the MobiLink output log. (This might be useful at development time but would slow down a production server.)

```
public String uploadStatisticsTable(
  String user,
  int warnings,
  int errors,
  int insertedRows,
  int deletedRows,
  int updatedRows,
  int conflictedInserts,
  int conflictedDeletes,
  int conflictedUpdates,
  int ignoredInserts,
  int ignoredDeletes,
  int ignoredUpdates,
  int bytes,
 int deadlocks
)
{  java.lang.System.out.println( "updated rows: " +
   updatedRows ); }
```

.NET example

The following stored procedure call registers a .NET method called UploadTableStats as the script for the upload_statistics table event when synchronizing the script version ver1 and the table table1.

```
call ml_add_dnet_table_script(
  'ver1',
  'table1',
  'upload_statistics',
  'TestScripts.Test.UploadTableStats'
)
```

Following is the sample .NET method uploadStatisticsTable. It logs some statistics to the MobiLink output log. (This might be useful at development time but would slow down a production server.)

```
public void UploadTableStats(
  string user,
  string table,
  int warnings,
  int errors,
  int insertedRows,
  int deletedRows,
  int updatedRows,
  int conflictInserts,
  int conflictDeletes,
  int conflictUpdates,
  int ignoredInserts,
  int ignoredDeletes,
  int ignoredUpdates,
  int bytes,
  int deadlocks )
{  System.Console.WriteLine( "updated rows: " +
   updatedRows ); }
```

upload_update table event

Function

Provides an event that the MobiLink synchronization server uses during processing of the upload stream to handle rows updated at the remote database.

Parameters

Clause	Parameters
SET	non-primary key column 1
	non-primary key column 2
	. . .
WHERE	primary key 1
	primary key 2
	. . .

Default action

None.

Description

The statement-based upload_update script may perform direct updates of column values as specified in the UPLOAD statement.

The WHERE clause must include all of the primary key columns that are being synchronized. The SET clause must contain all of the non-primary key columns that are being synchronized.

You use as many non-primary key columns in your SET clause as exist in the table, and MobiLink will send the correct number of column values. Similarly, in the WHERE clause, you can have any number of primary keys, but all must be specified here, and MobiLink will send the correct values. MobiLink sends these column values and primary key values in the order the columns or primary keys appear in a MobiLink report of your schema. You can use the -vh option to determine the column ordering for this table schema.

You can have one upload_update script for each table in the remote database.

For Java and .NET applications, this script must return valid SQL.

See also

- ♦ "upload_delete table event" on page 459
- ♦ "upload_fetch table event" on page 461
- ♦ "upload_insert table event" on page 463

SQL example

This example handles updates made to the Customer table in the remote database. The script updates the values in a table named Customer in the consolidated database.

```
call ml_add_table_script(
  'ver1',
  'table1',
  'upload_update',
'UPDATE Customer SET name=?, rep_id=? WHERE cust_id=?')
```

Java example

The following stored procedure call registers a Java method called uploadUpdateTable as the script for the upload_update table event when synchronizing the script version ver1.

```
call ml_add_java_table_script(
  'ver1',
  'table1',
  'upload_update',
  'ExamplePackage.ExampleClass.uploadUpdateTable' )
```

Following is the sample Java method uploadUpdateTable. It calls a method called genUU to dynamically generate an UPLOAD statement.

```
public string uploadUpdateTable()
{  return( genUU(_curTable) ); }
```

.NET example

The following stored procedure call registers a .NET method called UploadUpdate as the script for the upload_update table event when synchronizing the script version ver1 and the table table1.

```
call ml_add_dnet_table_script(
  'ver1',
  'table1',
  'upload_update',
  'TestScripts.Test.UploadUpdate'
)
```

Following is the C# signature for the call UploadUpdate.

```
public string UploadUpdate()
```

Part III

MobiLink Reference

This part contains MobiLink reference material.

CHAPTER 16

Stored Procedures

About this chapter

This chapter provides information about the MobiLink predefined stored procedures.

Contents

Stored procedures to add or delete scripts

You must add synchronization scripts to system tables in the consolidated database before you can use them. The following stored procedures add synchronization scripts to the consolidated database. They can also be used to delete scripts.

Notes

- When you add a script using a stored procedure, the script is a string. Any strings within the script need to be escaped. For Adaptive Server Anywhere, each quotation mark (‘) needs to be doubled so as not to terminate the string.
- You cannot use stored procedures to add scripts longer than 255 bytes to Adaptive Server Enterprise 11.5 or earlier. Instead, use Sybase Central or direct insertion to define longer scripts.
- IBM DB2 prior to version 6 only supports column names and other identifiers of 18 characters or less, and so the names are truncated. For example, ml_add_connection_script is shortened to ml_add_connection_.

ml_add_connection_script

Function

Use this stored procedure to add or delete SQL connection scripts in the consolidated database.

Parameters

Item	Description	Remarks
1	version name	VARCHAR(128)
2	event name	VARCHAR(128)
3	script contents	For Adaptive Server Anywhere and MS SQL Server, this is TEXT. For ASE, this is VARCHAR(16384). For ASE prior to 12.5, this is VARCHAR(255). For DB2, this is VARCHAR(4000). For Oracle, this is VARCHAR.

Description

To delete a connection script, set the script contents parameter to NULL.

When you add a script, the script is inserted into the ml_script table and the appropriate references are defined to associate the script with the event and script version that you specify. If the version name is new, it is automatically inserted into the ml_version table.

See also

- “Adding and deleting scripts in your consolidated database” on page 241
- “ml_add_table_script” on page 481

- ♦ “ml_add_dnet_connection_script” on page 482
- ♦ “ml_add_dnet_table_script” on page 483
- ♦ “ml_add_java_connection_script” on page 483
- ♦ “ml_add_java_table_script” on page 484

Example

The following statement adds a connection script associated with the begin_synchronization event to the script version custdb in an Adaptive Server Anywhere consolidated database. The script itself is the single statement that sets the @EmployeeID variable.

```
call ml_add_connection_script( ’custdb’,
   ’begin_synchronization’,
   ’set @EmployeeID = ?’ )
```

ml_add_table_script

Function

Use this stored procedure to add or delete SQL table scripts in the consolidated database.

Parameters

Item	Description	Remarks
1	version name	VARCHAR(128)
2	table name	VARCHAR(128)
3	event name	VARCHAR(128)
4	script contents	For Adaptive Server Anywhere and MS SQL Server, this is TEXT. For ASE, this is VARCHAR(16384). For ASE prior to 12.5, this is VARCHAR(255). For DB2, this is VARCHAR(4000). For Oracle, this is VARCHAR.

Description

To delete a table script, set the script contents parameter to NULL.

When you add a script, the script is inserted into the ml_script table and the appropriate references are defined to associate the script with the table, event and script version that you specify. If the version name is new, it is automatically inserted into the ml_version table.

See also

- ♦ “Adding and deleting scripts in your consolidated database” on page 241
- ♦ “ml_add_connection_script” on page 480
- ♦ “ml_add_dnet_connection_script” on page 482
- ♦ “ml_add_dnet_table_script” on page 483
- ♦ “ml_add_java_connection_script” on page 483
- ♦ “ml_add_java_table_script” on page 484

Example

The following command adds a table script associated with the upload_insert event on the Customer table.

```
call ml_add_table_script( 'default', 'Customer', 'upload_
         insert',
   'INSERT INTO Customer( cust_id, name, rep_id, active )
       VALUES ( ?, ?, ?, 1 )' )
```

ml_add_dnet_connection_script

Function

Use this stored procedure to add or delete .NET connection scripts in the consolidated database.

Parameters

Item	Description	Remarks
1	version name	VARCHAR(128)
2	event name	VARCHAR(128)
3	script contents	For Adaptive Server Anywhere and MS SQL Server, this is TEXT. For ASE, this is VARCHAR(16384). For ASE prior to 12.5, this is VARCHAR(255). For DB2, this is VARCHAR(4000). For Oracle, this is VARCHAR.

Description

To delete a connection script, set the script contents parameter to NULL.

The *script* value is a public method in a class in the MobiLink synchronization server classpath (for example, MyClass.MyMethod).

When you add a script, the method is associated with the event and script version that you specify. If the version name is new, it is automatically inserted into the ml_version table.

See also

- “Adding and deleting scripts in your consolidated database” on page 241
- “ml_add_dnet_table_script” on page 483
- “ml_add_connection_script” on page 480
- “ml_add_table_script” on page 481
- “ml_add_java_table_script” on page 484
- “Methods” on page 261

Example

The following example assigns the beginDownloadConnection method of the ExampleClass class to the begin_download event.

```
call ml_add_dnet_connection_script( 'ver1',
'begin_download',
'ExamplePackage.ExampleClass.beginDownloadConnection')
```

ml_add_dnet_table_script

Function

Use this stored procedure to add or delete .NET table scripts in the consolidated database.

Parameters

Item	Description	Remarks
1	version name	VARCHAR(128)
2	table name	VARCHAR(128)
3	event name	VARCHAR(128)
4	script contents	For Adaptive Server Anywhere and MS SQL Server, this is TEXT. For ASE, this is VARCHAR(16384). For ASE prior to 12.5, this is VARCHAR(255). For DB2, this is VARCHAR(4000). For Oracle, this is VARCHAR.

Description

To delete a connection script, set the script contents parameter to NULL.

The *script* value is a public method in a class in the MobiLink synchronization server classpath (for example, MyClass.MyMethod).

When you add a script, the method is associated with the table, event, and script version that you specify. If the version name is new, it is automatically inserted into the ml_version table.

See also

- “Adding and deleting scripts in your consolidated database” on page 241
- “ml_add_dnet_connection_script” on page 482
- “ml_add_connection_script” on page 480
- “ml_add_table_script” on page 481
- “ml_add_java_connection_script” on page 483
- “Methods” on page 261

Example

The following example assigns the empDownloadCursor method of the EgClass class to the download_cursor event for the table emp.

```
call ml_add_dnet_table_script( ’ver1’, ’emp’,
’download_cursor’,EgPackage.EgClass.empDownloadCursor’)
```

ml_add_java_connection_script

Function

Use this stored procedure to add or delete Java connection scripts in the consolidated database.

Parameters

Item	Description	Remarks
1	version name	VARCHAR(128)
2	event name	VARCHAR(128)
3	script contents	For Adaptive Server Anywhere and MS SQL Server, this is TEXT. For ASE, this is VARCHAR(16384). For ASE prior to 12.5, this is VARCHAR(255). For DB2, this is VARCHAR(4000). For Oracle, this is VARCHAR.

Description

To delete a connection script, set the script contents parameter to NULL.

The *script* value is a public method in a class in the MobiLink synchronization server classpath (for example, MyClass.MyMethod).

When you add a script, the method is associated with the event and script version that you specify. If the version name is new, it is automatically inserted into the ml_version table.

See also

- "Adding and deleting scripts in your consolidated database" on page 241
- "ml_add_connection_script" on page 480
- "ml_add_table_script" on page 481
- "ml_add_dnet_connection_script" on page 482
- "ml_add_dnet_table_script" on page 483
- "ml_add_java_table_script" on page 484
- "Methods" on page 261

Example

The following example is taken from the *Samples\MobiLink\JavaAuthentication* sample. It assigns the endConnection method of the CustEmpScripts class to the end_connection event.

```
call ml_add_java_connection_script( 'ver1',
'end_connection',
'CustEmpScripts.endConnection')
```

ml_add_java_table_script

Function

Use this stored procedure to add or delete Java table scripts in the consolidated database.

Parameters

Item	Description	Remarks
1	version name	VARCHAR(128)
2	table name	VARCHAR(128)
3	event name	VARCHAR(128)
4	script	For Adaptive Server Anywhere and MS SQL Server, this is TEXT. For ASE, this is VARCHAR(16384). For ASE prior to 12.5, this is VARCHAR(255). For DB2, this is VARCHAR(4000). For Oracle, this is VARCHAR.

Description

To delete a connection script, set the script parameter to NULL.

The *script* value is a public method in a class in the MobiLink synchronization server classpath (for example, MyClass.MyMethod).

When you add a script, the method is associated with the table, event, and script version that you specify. If the version name is new, it is automatically inserted into the ml_version table.

See also

- "Adding and deleting scripts in your consolidated database" on page 241
- "ml_add_connection_script" on page 480
- "ml_add_table_script" on page 481
- "ml_add_dnet_connection_script" on page 482
- "ml_add_dnet_table_script" on page 483
- "ml_add_java_connection_script" on page 483
- "Methods" on page 261

Example

The following example is taken from the *Samples\MobiLink\JavaAuthentication* sample. It assigns the empDownloadCursor method of the CustEmpScripts class to the download_cursor event for the table emp.

```
call ml_add_java_table_script( 'ver1', 'emp',
'download_cursor','CustEmpScripts.empDownloadCursor')
```

Stored procedures to add or delete properties

The following stored procedure adds properties to the consolidated database. It can also be used to delete properties.

ml_add_property

Function

Use this stored procedure to add or delete MobiLink properties. This stored procedure changes rows in the ml_property MobiLink system table.

Parameters

Item	Description	Remarks
1	component_name	VARCHAR(128)
2	prop_set_name	VARCHAR(128)
3	prop_name	VARCHAR(128)
4	prop_value	LONG VARCHAR

Description

♦ **component_name** You can create records for which this parameter is **ScriptVersion** or **SIS**.

To save properties by script version, set this to **ScriptVersion**.

For server-initiated synchronization properties, set this to **SIS**. For more information, see "Setting properties" [*MobiLink Server-Initiated Synchronization User's Guide*, page 15].

♦ **prop_set_name** If the component_name is **ScriptVersion**, then this is the name of the script version.

If the component_name is **SIS**, then this is the name of the Notifier, gateway, or carrier that you are setting a property for.

♦ **prop_name** This is the name of the property.

If the component_name is **ScriptVersion**, then this is a property that you define. You reference these properties using the following methods:

- from DBConnectionContext: getVersion and getProperties
- from ServerContext: getPropertiesByVersion, getProperties, and getPropertySetNames

For more information, see "MobiLink Java API Reference" on page 273 and "MobiLink .NET API Reference" on page 303.

If the component_name is **SIS**, then this is a property of the Notifier, gateway, or carrier. For a list of properties, see "MobiLink Notification Properties" [*MobiLink Server-Initiated Synchronization User's Guide*, page 55].

- **prop_value** This is the value of the property.

  To delete a property, set the prop_value parameter to NULL.

Server-initiated synchronization

For server-initiated synchronization, the ml_add_property stored procedure allows you to set properties for Notifiers, gateways, and carriers.

For example, to add the property `server=mailserver1` for an SMTP gateway called x:

```
ml_add_property( 'SIS','SMTP(x)','server','mailserver1');
```

The verbosity property applies to all Notifiers and gateways, and so you cannot specify a particular prop_set_name. To change the verbosity setting, leave the prop_set_name blank:

```
ml_add_property( 'SIS','','verbosity',2);
```

☞ For more information about setting properties, see "Setting properties" [*MobiLink Server-Initiated Synchronization User's Guide*, page 15]. For a complete list of server-initiated synchronization properties, see "MobiLink Notification Properties" [*MobiLink Server-Initiated Synchronization User's Guide*, page 55].

Script Version

For regular MobiLink synchronization, you can use this stored procedure to associate properties with a script version. In this case, set the component_name to **ScriptVersion**. You can specify whatever properties you like, and use Java and .NET classes to access them.

For example, to associate an LDAP server with a script version called MyVersion:

```
ml_add_property( 'ScriptVersion','MyVersion','ldap-
          server','MyServer' )
```

☞ For more information, see the following methods in the "MobiLink Java API Reference" on page 273 and "MobiLink .NET API Reference" on page 303:

- from DBConnectionContext: getVersion and getProperties
- from ServerContext: getPropertiesByVersion, getProperties, and getPropertySetNames

See also

"ml_property" on page 511

CHAPTER 17

Utilities

About this chapter

This chapter describes MobiLink synchronization server utilities.

☞ For information about MobiLink client utilities, see "Utilities" [*MobiLink Clients*, page 27].

☞ For information about other Adaptive Server Anywhere utilities, see "Database Administration Utilities" [*ASA Database Administration Guide*, page 493].

Contents

MobiLink stop utility

Stops the MobiLink synchronization server on the local machine.

Syntax

dbmlstop [*options*] [*server-name*]

Option	Description
@*data*	Reads options from the specified environment variable or configuration file.
-f	Forced shutdown. Use if a hard shutdown does not work.
-h	Hard shutdown. MobiLink stops all synchronizations and exits. Some remotes may report an error.
-q	Quiet mode. Suppresses the banner.
-t *time*	Soft shutdown, with a hard shutdown done after the specified time. *time* is a number followed by D, H, M, or S (for days, hours, minutes and seconds). For example, `-t 10m` specifies that the server should be shut down in 10 minutes or when current synchronizations complete, whichever is sooner. D, H, M, and S are not case sensitive.
-w	Waits for the MobiLink synchronization server to shut down before returning from the command.

Parameters

@data Use this option to read in options from the specified environment variable or configuration file. If both exist with the same name, the environment variable is used.

☞ For more information about configuration files, see "Using configuration files" [*ASA Database Administration Guide,* page 495].

If you want to protect passwords or other information in the configuration file, you can use the File Hiding utility to obfuscate the contents of the configuration file.

☞ See "Hiding the contents of files using the dbfhide command-line utility" [*ASA Database Administration Guide,* page 524].

Server-name If the MobiLink synchronization server is started using the -zs option, it must be shut down by specifying the same server name.

☞ For more information, see “-zs option” on page 221.

Description

By default (if none of -f, -h or -t are specified), dbmlstop does a soft shutdown.

♦ **Soft shutdown** means that the MobiLink synchronization server stops accepting new connections and exits when the current synchronizations are complete.

♦ **Hard shutdown** means that the MobiLink synchronization server stops all synchronizations and exits. Some remotes may report an error.

MobiLink user authentication utility

Registers MobiLink users at the consolidated database. For Adaptive Server Anywhere remotes, the users must have previously been created at the remote databases with the CREATE SYNCHRONIZATION USER statement.

Syntax

dbmluser [*options*] **-c** "*connection-string*"
{ **-f** *file* | **-u** *user* [**-p** *password*] }

Option	Description
@*data*	Reads options from the specified environment variable or configuration file.
-c "*keyword=value;...*"	Supply database connection parameters. The connection string must give the utility permission to connect to the consolidated database using an ODBC data source. This parameter is required.
-d	Deletes the user name(s) specified by -f or -u.
-dl	Display messages in the window or on the command line and also in the log file, if specified.
-f *filename*	Read the user names and passwords from the specified file. The file should be a text file containing one user name and password pair on each line, separated by white space. You must specify either -f or -u.
-o *filename*	Log output messages to the specified file.
-os *size*	Limit the size of the output file. The *size* is the maximum file size for logging output messages, specified in bytes. Use the suffix k or m to specify units of kilobytes or megabytes, respectively. By default, there is no size limit. The minimum size limit is 10 kb.
-ot *filename*	Truncate the log file and then append output messages to it. The default is to send output to the screen.

Option	Description
-p *password*	Password to associate with the user. This option can only be used with -u.
-pc *collation-id*	Supply a database collation ID for character set translation of the user name and password. This should be one of the Adaptive Server Anywhere collation labels such as those listed in "Initialization utility options" [*ASA Database Administration Guide,* page 532]. This option is required when user names and passwords are read from a file that is encoded in a different character set than the default character set determined by locale.
-u *ml_username*	Specify the user name to add (or delete, if used with -d). Only one user can be specified on a single command line. This option is used with -p if passwords are being used. You must specify either -f or -u.

Options

@data Use this option to read in options from the specified environment variable or configuration file. If both exist with the specified name, the environment variable is used.

☞ For more information about configuration files, see "Using configuration files" [*ASA Database Administration Guide,* page 495].

If you want to protect passwords or other information in the configuration file, you can use the File Hiding utility to obfuscate the contents of the configuration file.

☞ See "Hiding the contents of files using the dbfhide command-line utility" [*ASA Database Administration Guide,* page 524].

Description

Given a user/password pair, the dbmluser utility first attempts to add the user. If the user has already been added to the consolidated database, it attempts to update the password for that user.

There are alternative ways to register user names in the consolidated database:

♦ Use Sybase Central.

- Specify the -zu+ command line option with dbmlsrv9. In this case, any existing MobiLink users that have not been added to the consolidated database are added when they first synchronize.

The MobiLink user must already exist in a remote database. To add users at the remote, you have the following options:

- For Adaptive Server Anywhere remotes, set the name with CREATE SYNCHRONIZATION USER and synchronize with that user name.
- For UltraLite remotes, you can either use the user_name field of the ul_synch_info structure; or in Java, use the SetUserName() method of the ULSynchInfo class before synchronizing.

See also

- “Authenticating MobiLink Users” [*MobiLink Clients,* page 9]

Certificate reader utility

Use the *readcert* utility to display values within a certificate and validate the chain of certificates.

Syntax

readcert *certificate-name*

Description

The certificate you specify can be elliptic-curve or RSA.

When synchronization occurs through an ECC_TLS or RSA_TLS synchronization stream, the MobiLink synchronization server sends its certificate to the client, as well as the certificate of the entity that signed it, and so on up to a self-signed root. The client checks that the chain is valid and that it trusts the root certificate in the chain.

This utility scans an X509 authentication certificate and displays the field values. It then checks that the chain of certificates is valid. A validation error is reported if any of the certificates in the chain have expired, are in the wrong order, or are missing.

See also

- "Certificate generation utility" on page 496

Certificate generation utility

Use the gencert utility to create new elliptic-curve or RSA certificates, or to sign pre-generated certificate requests.

Syntax

gencert [**-c** | **-s**] [**-r** | **-q**]

Option	Description
-c	Specifies a certificate you can use to sign other certificates. If used with **-r**, generates an enterprise root certificate.
-s	Specifies a server identity certificate. The server identity is a combination of a server's private key and public certificate. You reference the server identity certificate when you start the MobiLink synchronization server (for MobiLink transport-layer security) or database server (for Adaptive Server Anywhere client-server transport-layer security). If used with **-r**, generates a self-signed server certificate.
-r	Specifies a self-signed root certificate. If used with **-s**, gencert creates a self-signed server certificate. If used with **-c**, gencert creates an enterprise root certificate you can use to sign other certificates. If you specify **gencert -r** with no additional options, gencert creates a certificate you can use as a server certificate or an enterprise root. This option is not compatible with **-q**.
-q *request-file*	Sign a pre-generated certificate request. If used with **-s**, gencert creates a server certificate. If used with **-c**, gencert creates an enterprise root certificate you can use to sign other certificates. If you specify **gencert -q** with no additional options, gencert creates a certificate you can use as a server certificate or an enterprise root. The **-q** option is not compatible with **-r**.

If you do not specify -s or -c , the certificate contains the functionality provided by both options, so it can be used to sign other certificates or you can use it directly as a server certificate.

Description

You can use the gencert utility to generate trusted public certificates, private keys, and server certificates used to secure MobiLink synchronizations or Adaptive Server Anywhere client-server communication. This utility creates X509 certificates (a standard certificate format) for various security configurations.

Gencert prompts you for the following information:

- **Cipher** Gencert prompts you to choose an elliptic-curve or RSA cipher. If you are generating an elliptic-curve certificate, gencert generates an elliptic-curve key pair. If you are generating an RSA certificate, it prompts for a key size between 512 and 2048, and then creates a certificate using RSA. (In general, longer keys provide stronger encryption but take longer to process.)

  Whichever cipher you choose, you must specify that cipher when you start the server and client. For ECC certificates, specify ecc_tls, and for RSA certificates, you can specify rsa_tls or rsa_tls_fips.

- **Country, State/Province, and Locality** These values provide general certificate identification. The locality fields are also required by third-party Certificate Authorities if you plan to use globally-signed certificates.

  ☞ For more information about using Certificate Authorities, see "Globally-signed certificates" on page 173.

- **Organization, Organizational Unit, and Common Name** These fields provide additional security that the client is authenticating the correct certificate. On the client side, they correspond to the certificate_company, certificate_unit, and certificate_name protocol options, respectively.

  ☞ See "Verifying certificate fields" on page 180.

- **Serial number** You are prompted to choose a serial number for the certificate. The serial number must use alphanumeric characters.

- **Certificate valid for how many years** You are prompted for the period (in years) that the enterprise root certificate remains valid. If the enterprise root certificate expires, all signed server certificates will also be invalid. Following the specified period, you will need to regenerate the enterprise root, each server certificate, and the public certificates distributed to clients.

- **Enter password to protect private key** This is the password you will specify in the certificate_password protocol option.

- **Enter file path to save certificate** Choose a file name and location for the certificate.
- **Enter file path to save private key** Choose a file name and location for the private key.
- **Enter file path to save server identity** You will only be prompted for this information if you are creating a self-signed root certificate. Choose a file name and location for the server certificate.

See also

☞ For more information about using certificates to secure MobiLink synchronizations, see "MobiLink Transport-Layer Security" on page 165.

☞ For more information about using certificates to secure Adaptive Server Anywhere RDBMSs, see "Adaptive Server Anywhere Transport-Layer Security" [*SQL Anywhere Studio Security Guide*, page 27].

- "Certificate reader utility" on page 495
- "-ec server option" [*ASA Database Administration Guide*, page 135]
- "Encryption connection parameter [ENC]" [*ASA Database Administration Guide*, page 191]
- "FIPS 140-2 certification" on page 166

Examples

- The following example uses the gencert -r option to generate an RSA self-signed server certificate.

```
> gencert -r
Certificate Generation Tool
Choose certificate type ((R)SA or (E)CC): R
Enter key length (512-2048): 1024
Generating key pair...
Country: CA
State/Province: ON
Locality: Waterloo
Organization: Sybase, Inc.
Organizational Unit: IAS
Common Name: MobiLink
Serial Number: 123
Certificate valid for how many years: 12
Enter password to protect private key: mypwd123
Enter file path to save certificate: public_cert.crt
Enter file path to save private key: server_key.pri
Enter file path to save server identity: server_cert.crt
```

  You distribute the self-signed public certificate *public_cert.crt* to clients. The server certificate *server_cert.crt* includes the server certificate and private key and is stored with the MobiLink synchronization server (for MobiLink transport-layer security) or the database server (for Adaptive Server Anywhere client-server transport-layer security).

- The following example uses the gencert -r and -c options to generate an enterprise root certificate.

```
> gencert -r -c
Certificate Generation Tool
Choose certificate type ((R)SA or (E)CC): E
Generating key pair...
Country: CA
State/Province: ON
Locality: Waterloo
Organization: Sybase, Inc.
Organizational Unit: IAS
Common Name: MobiLink
Serial Number: 123
Certificate valid for how many years: 15
Enter password to protect private key: mypwd123
Enter file path to save certificate: public_root_cert.crt
Enter file path to save private key: root_key.pri
```

The public enterprise root certificate *public_root_cert.crt* is distributed to clients. Store the enterprise private key *root_key.pri* in a secure location. You can use the generated files to sign server certificates.

- The following example uses the gencert -s option to create a signed server certificate.

```
> gencert -s
Certificate Generation Tool
Choose certificate type ((R)SA or (E)CC): E
Generating key pair...
Country: CA
State/Province: ON
Locality: Waterloo
Organization: Sybase, Inc.
Organizational Unit: IAS
Common Name: MobiLink
Serial Number: 123
Certificate valid for how many years: 7
Enter file path of signer's certificate: public_root_
         cert.crt
Enter file path of signer's private key: root_key.pri
Enter password for signer's private key: mypwd123
Enter password to protect private key: SERV1privatekeypwd
Enter file path to save server identity: serv1_cert.crt
```

The enterprise root certificate created in the previous example is used as the signing certificate. You reference the server identity *serv1_cert.crt* and the password SERV1privatekeypwd when you start the MobiLink synchronization server or your database server.

- The following example uses the gencert option -q to sign a certificate request called *certreq.txt.*

```
>gencert -s -q certreq.txt
Certificate Generation Tool
Serial Number: 01
Certificate valid for how many years: 10
Enter file path of signer’s certificate: rsaroot.crt
Enter file path of signer’s private key: rsaroot.key
Enter password for signer’s private key: test
Enter file path to save certificate: testcert.crt
Save entire chain (y/n): y
```

The certificate request is created using reqtool. For more information, see

- “Globally-signed certificates” on page 173 for MobiLink transport-layer security.
- “Globally-signed certificates” [*SQL Anywhere Studio Security Guide*, page 35] for Adaptive Server Anywhere transport-layer security.

CHAPTER 18

MobiLink System Tables

About this chapter

This chapter describes the MobiLink administration tables.

Contents

Introduction

The MobiLink system tables store information about MobiLink users, subscriptions, tables, scripts, and script versions. They are created when you run the MobiLink setup scripts for your consolidated database, and are updated as you use MobiLink. They are not like RDBMS system tables in that you can alter them directly.

☞ For more information about how to create these tables, see "Setting up a consolidated database" on page 33.

The MobiLink system tables are stored in the consolidated database. Unlike most system tables, you can modify them. However, except in rare circumstances you should not alter them directly. You should never alter MobiLink system tables starting with ml_qa.

Adaptive Server Anywhere remote databases also have system tables. For more information, see "System Tables" [*ASA SQL Reference*, page 671]. UltraLite databases do not have system tables.

Notes

- This chapter provides data types for the MobiLink system tables in Adaptive Server Anywhere consolidated databases. In some RDBMSs, the data types are slightly different.
- IBM DB2 UDB version 5.2 only supports column names and other identifiers of 18 characters or less. In a DB2 5.2 consolidated database, MobiLink system tables are truncated where necessary.

ml_connection_script

For a given script version, associates a script with a given event.

Column	Description
version_id	INTEGER. Primary key. The version of the connection script. The version_id column references the version_id column of the ml_script_version table.
event	VARCHAR(128). Primary key. The event column stores the name of the event that triggers the connection script.
script_id	INTEGER. Foreign key. The script_id column references the script_id column of the ml_script system table. The text of the connection script is stored in the ml_script system table.

Remarks

There is a view, ml_connection_scripts, that makes it easier to view the contents of the ml_connection_script MobiLink system table.

ml_device

This table is used only for server-initiated synchronization. It stores device names that are required by device tracking.

Column	Description
device_name	VARCHAR(255). Primary key.
listener_-version	VARCHAR(128). Not null.
listener_-protocol	INTEGER. Not null.
info	VARCHAR(255). Not null.
ignore_tracking	CHAR(1). Not null.
source	VARCHAR(255). Not null.

Description

- **device_name** A name given to the device. This name is extracted from the operating system unless you specify a name using the dblsn -e option.
- **listener_version** This field contains the SQL Anywhere Studio version number for the installed software on the device. Changing this value does not affect the operation of the software, but may be useful for diagnostic purposes.
- **listener_protocol** This value is **0** for Listeners from versions of SQL Anywhere Studio prior to 9.0.1, and Listeners started with the -s option; **1** for post-9.0.0 Palm Listeners; and **2** for post-9.0.0 Windows Listeners started without -s.
- **info** Information about the device. This field is populated if you provide information using the dblsn -f option.
- **ignore_tracking** If this is **y**, tracking information is not written to the row. If it is **n**, tracking information is written to the row.
- **source** This is **tracking** if the row was created by automatic device tracking. Otherwise, it is blank unless you change it using stored procedures to add information about where the data in this row came from. Unless it is set to **tracking**, the value in this column does not affect the operation of the software.

Remarks

The MobiLink system tables ml_device, ml_device_address, and ml_listening contain information about devices for server-initiated synchronization. DeviceTracker gateways use this information to address target devices by MobiLink user name.

In most cases, you should not need to alter these tables. However, if your device does not support device tracking or if you want to override device tracking for troubleshooting purposes, you can add or delete rows to this system table using pre-defined stored procedures. For more information, see "Using device tracking with Listeners that don't support it" [*MobiLink Server-Initiated Synchronization User's Guide*, page 25].

If you want to stop automatic tracking, set override_tracking to **y**. In this case, it is also recommended that you use a source name other than **tracking**. For more information, see "ml_set_device" [*MobiLink Server-Initiated Synchronization User's Guide*, page 81].

See also

- "ml_set_device" [*MobiLink Server-Initiated Synchronization User's Guide*, page 81]
- "ml_delete_device" [*MobiLink Server-Initiated Synchronization User's Guide*, page 78]

ml_device_address

This table is used only for server-initiated synchronization. It stores addressing information that is required by device tracking.

Column	Description
device_name	VARCHAR(255). Primary key. Foreign key references dbo.ml_device. Not null.
medium	VARCHAR(255). Primary key. Not null.
address	VARCHAR(255). Not null.
active	CHAR(1). Not null.
last_modified	Timestamp. Not null. Default timestamp.
ignore_tracking	CHAR(1). Not null.
source	VARCHAR(255). Not null.

Description

- ♦ **device_name** A name given to the device. This name is extracted from the operating system unless you specify a name using the dblsn -e option.
- ♦ **medium** For UDP, this is **_UDP_**. Otherwise, it is the network provider ID.
- ♦ **address** For UDP, this is *ip*:*port-number*, where *ip* is an IP address or host name. For SMTP, this is the phone number.
- ♦ **active** This is **y** for active, and **n** otherwise. A DeviceTracker gateway may deactivate a UDP channel if it is unresponsive and there is a fallback SMTP delivery path.
- ♦ **last_modified** This tells when this row was last modified.
- ♦ **ignore_tracking** If this is **y**, tracking information is not written to the row. If it is **n**, tracking information is written to the row.
- ♦ **source** This is **tracking** if the row was created by automatic device tracking. Otherwise, it is blank unless you change it using stored procedures to add information about where the data in this row came from. Unless it is set to **tracking**, the value in this column does not affect the operation of the software.

Remarks

The MobiLink system tables ml_device, ml_device_address, and ml_listening contain information about devices for server-initiated synchronization. DeviceTracker gateways use this information to address target devices by MobiLink user name.

In most cases, you should not need to alter these tables. However, if your device does not support device tracking or if you want to override device tracking for troubleshooting purposes, you can add or delete rows to this system table using pre-defined stored procedures. For more information, see "Using device tracking with Listeners that don't support it" [*MobiLink Server-Initiated Synchronization User's Guide*, page 25].

If you want to stop automatic tracking, set override_tracking to **y**. In this case, it is also recommended that you use a source name other than **tracking**. For more information, see "ml_set_device_address" [*MobiLink Server-Initiated Synchronization User's Guide*, page 83].

See also

- ♦ "ml_set_device_address" [*MobiLink Server-Initiated Synchronization User's Guide*, page 83]
- ♦ "ml_delete_device_address" [*MobiLink Server-Initiated Synchronization User's Guide*, page 79]

ml_listening

This table is used only for server-initiated synchronization. It maps a MobiLink user name to a device name for use by device tracking.

Column	Description
ml_user	VARCHAR(128). Primary key. Not null.
device_name	VARCHAR(255). Foreign key references dbo.ml_device. Not null.
listening	CHAR(1). Not null.
ignore_tracking	CHAR(1). Not null.
source	VARCHAR(255). Not null.

Description

- **ml_user** This is the MobiLink user name, which uniquely identifies a remote database.

  If you use the dblsn -t+ option, this is the alias that you have defined for the ml_user. For more information, see "Listener options for device tracking" [*MobiLink Server-Initiated Synchronization User's Guide*, page 23].

- **device_name** A name given to the device. This name is extracted from the operating system unless you specify a name using the dblsn -e option.

- **listening** This is **y** for an active Listener; otherwise it is **n**. This field is set when you use the dblsn option -t, or you can manually set it with stored procedures.

- **ignore_tracking** If this is **y**, tracking information is not written to the row. If it is **n**, tracking information is written to the row.

- **source** This is **tracking** if the row was created by automatic device tracking. Otherwise, it is blank unless you change it using stored procedures to add information about where the data in this row came from. The value in this column does not affect the operation of the software.

Remarks

The MobiLink system tables ml_device, ml_device_address, and ml_listening contain tracked information about devices for server-initiated synchronization. DeviceTracker gateways use this information to address target devices by MobiLink user name.

In most cases, you should not need to alter these tables. However, if your device does not support device tracking or if you want to override device tracking for troubleshooting purposes, you can add or delete rows in this table using pre-defined stored procedures. For more information, see "Using device tracking with Listeners that don’t support it” [*MobiLink Server-Initiated Synchronization User’s Guide,* page 25].

If you want to stop automatic tracking, set override_tracking to Yes. In this case, it is also recommended that you use a source name other than **tracking**.

See also

- “ml_set_listening” [*MobiLink Server-Initiated Synchronization User’s Guide,* page 85]
- “ml_delete_listening” [*MobiLink Server-Initiated Synchronization User’s Guide,* page 80]

ml_property

Stores some MobiLink properties.

Column	Description
component_-name	VARCHAR(128). First part of the composite primary key.
property_set_-name	VARCHAR(128). Second part of the composite primary key.
property_name	VARCHAR(128). Third part of the composite primary key.
property_value	LONG VARCHAR.

Description

- **component_name** For user-defined properties, this can be **ScriptVersion** or **SIS**.
- **prop_set_name** If the component_name is **ScriptVersion**, then this is the name of the script version.

  If the component_name is **SIS**, then this is the name of the Notifier, gateway, or carrier that you are setting a property for.
- **prop_name** This is the name of the property.

  If the component_name is **ScriptVersion**, then this is a user-defined property.

  If the component_name is **SIS**, then this is a property of the Notifier, gateway, or carrier. For a list of properties, see "MobiLink Notification Properties" [*MobiLink Server-Initiated Synchronization User's Guide*, page 55].
- **prop_value** This is the value of the property.

Remarks

This table stores name-value pairs. Some of the properties in this table are used internally by MobiLink. In addition, you can use the stored procedure ml_add_property to add or delete rows in this table.

You can use the component_name **ScriptVersion** to store information on per script version basis that can be accessed by Java or .NET scripting logic.

This table has a composite primary key made up of component_name, property_set_name, and property_name.

See also

- "ml_add_property" on page 486

ml_qa_delivery

This table is used only for QAnywhere applications.

Column	Description
msgid	VARCHAR(255). Globally unique message identifier.
address	VARCHAR(255). Address of the target recipient.
client	VARCHAR(255).Client of the target recipient.
status	INTEGER. The status of the message. Can be **1** (pending), **10** (receiving), **30** (expired), **50** (receivable), or **60** (received).
statustime	TIMESTAMP. The time this status was achieved.
verbiage	VARCHAR (32767). Localized description of the status (if any).
syncstatus	INTEGER. Indicates the state of the synchronization between the client and server with respect to this message Can be **0** (not in sync), **1** (in sync), or **2** (message should not be synchronized).
receiverid	VARCHAR(128). An identifier set by the receiver that identifies the receiver of the message, if any.
last_modified	TIMESTAMP. Indicates the last time the status was changed.

Remarks

The owner of this table is ml_qa_user_group.

ml_qa_delivery_client

Column	Description
msgid	VARCHAR(255). Globally unique message identifier.
address	VARCHAR(255). Address of the target recipient.
client	VARCHAR(255).Client of the target recipient.
status	INTEGER. The status of the message. Can be **1** (pending), **10** (receiving), **30** (expired), **50** (receivable), or **60** (received).
statustime	TIMESTAMP. The time this status was achieved.
verbiage	VARCHAR (32767). Localized description of the status (if any).
syncstatus	INTEGER. Indicates the state of the synchronization between the client and server with respect to this message Can be **0** (not in sync), **1** (in sync), or **2** (message should not be synchronized).
receiverid	VARCHAR(128). An identifier set by the receiver that identifies the receiver of the message, if any.

Remarks

The owner of this table is ml_qa_user_group.

ml_qa_global_props

This table is used only for QAnywhere applications. It contains global *name-value* pairs that are used in transmission rules.

Column	Description
client	VARCHAR(255). Primary key. The client associated with the property. A client value of ' ' indicates a property that is global to all clients.
name	VARCHAR(255). Primary key. The name of the property.
modifiers	INTEGER. Bitfields used to further describe the property. Currently, only the first bit is used to indicate a property that should not be synchronized. All other bit fields are reserved for future use.
value	VARCHAR(32767). The value of the property.
last_modified	TIMESTAMP. Indicates the last time the value was changed. This is necessary to indicate when a property needs to be synchronized with the client.

Remarks

This table has a composite primary key made up of client and name.

The owner of this table is ml_qa_user_group.

Do not modify the contents of this table.

ml_qa_global_props_client

This table is used only for QAnywhere applications. It is identical to ml_qa_global_props. This table is created on the remote database by the QAnywhere Agent as needed.

Do not modify the contents of this table.

Column	Description
name	VARCHAR(255). Primary key. The name of the property.
modifiers	INTEGER. Bitfields used to further describe the property. Currently, only the first bit is used to indicate a property that should not be synchronized. All other bit fields are reserved for future use.
value	VARCHAR(32767). The value of the property.

Remarks

The owner of this table is ml_qa_user_group.

Do not modify the contents of this table.

See also

- ♦ "ml_qa_global_props" on page 514

ml_qa_notifications

This table is used only for QAnywhere applications. It is used by the Notifier to determine which QAnywhere users to notify to initiate synchronization.

Column	Description
user_id	INTEGER. Primary key.
name	VARCHAR(128). The MobiLink user name that uniquely identifies a remote database.

Remarks

The owner of this table is ml_qa_user_group.

Do not modify the contents of this table.

ml_qa_repository

This table is used only for QAnywhere applications. It stores messages and their properties.

Column	Description
seqno	BIGINT. Used to give a total ordering to the messages, this is necessary for true queuing.
msgid	VARCHAR(255). Primary key. Globally unique message identifier.
originator	VARCHAR(255). The name of the originating MobiLink user.
priority	INTEGER. A number from **0** to **9**. Messages with a higher priority number are delivered before messages with a lower priority. The default is **4**.
expires	TIMESTAMP . Expiry time after which the message might not be delivered.
kind	INTEGER. Indicates whether the message is binary (**1**) or text (**2**).
contentsize	BIGINT. The size of the message. For binary messages, this is the number of bytes. For text messages, this is the number of characters.
props	LONG BINARY. An encoding of the message properties. For information about the properties, see "ml_qa_repository_props" on page 520.
content	LONG BINARY. The content of the message. Text messages are encoded as UTF-8.

Remarks

The owner of this table is ml_qa_user_group.

Do not modify the contents of this table.

ml_qa_repository_client

This table is used only for QAnywhere applications. It is identical to ml_qa_repository. This table is created on the remote database by the QAnywhere Agent as needed.

Column	Description
seqno	BIGINT. Used to give a total ordering to the messages, this is necessary for true queuing.
msgid	VARCHAR(255). Primary key. Globally unique message identifier.
syncstatus	INTEGER. Indicates the state of the synchronization between the client and server with respect to this message. **0** means that they are not in sync; **1** means that they are in sync; **2** means that a message should not be synchronized.
originator	VARCHAR(255). The name of the originating MobiLink user.
priority	INTEGER. A number from **0** to **9**. Messages with a higher priority number are delivered before messages with a lower priority. The default is **4**.
expires	TIMESTAMP . Expiry time after which the message might not be delivered.
kind	INTEGER. Indicates whether the message is binary (**1**) or text (**2**).
contentsize	BIGINT. The size of the message. For binary messages, this is the number of bytes. For text messages, this is the number of characters.
props	LONG BINARY. An encoding of the message properties. For information about the properties, see “ml_qa_repository_props” on page 520.
content	LONG BINARY. The content of the message. Text messages are encoded as UTF-8.

Remarks

The owner of this table is ml_qa_user_group.

Do not modify the contents of this table.

See also

- “ml_qa_repository” on page 517

ml_qa_repository_content_client

This table is used only for QAnywhere applications. It is identical to ml_qa_repository_content. This table is created on the remote database by the QAnywhere Agent as needed.

The owner of this table is ml_qa_user_group.

Do not modify the contents of this table.

ml_qa_repository_props

This table is used only for QAnywhere applications. This is an expansion of the props column in the ml_qa_repository table. Properties are only expanded as needed by the transmission rules engine. If there are no associated rules, a property is not expanded.

Column	Description
msgid	VARCHAR(255). Primary key. Globally unique message identifier.
name	VARCHAR(255). Primary key. The name of the property. If the property name was provided in Unicode, it is translated to the native character set of the database.
value	VARCHAR(32767). The value of the property.

Remarks

The owner of this table is ml_qa_user_group.

Do not modify the contents of this table.

This table has a composite primary key made up of msgid and name.

ml_qa_repository_props_client

This table is used only for QAnywhere applications. It is identical to ml_qa_repository_props. This table is created on the remote database by the QAnywhere Agent as needed.

Do not modify the contents of this table.

Column	Description
msgid	VARCHAR(255). Primary key. Globally unique message identifier.
name	VARCHAR(255). Primary key. The name of the property. If the property name was provided in Unicode, it is translated to the native character set of the database.
value	VARCHAR(32767). The value of the property.

Remarks

The owner of this table is ml_qa_user_group.

See also

♦ "ml_qa_repository_props" on page 520

ml_qa_repository_staging

This table is used only for QAnywhere applications. It contains messages that are to be sent to a QAnywhere client.

Column	Description
seqno	BIGINT. Used to give a total ordering to the messages, this is necessary for true queuing.
msgid	VARCHAR(255). Primary key. Globally unique message identifier.
destination	VARCHAR(255). The address of the message.
originator	VARCHAR(255). The name of the originating MobiLink user.
status	VARCHAR(255). The status of the message. Can be **pending**, **receiving**, **received**, **unreceivable**, **expired**, or **cancelled**. The default is **pending**.
statustime	TIMESTAMP. The last time the status was changed.
uploaded	BIT. Indicates whether the sender and recipient are in sync on the status of the message. **0** means that they are not in sync and **1** indicates that they are in sync. The default is **0**.
expires	TIMESTAMP. Expiry time after which the message will not be delivered.
priority	INTEGER. A number from **0** to **9**. Messages with a higher priority number will always be delivered before messages with a lower priority. The default is **4**.
props	LONG BINARY. An encoding of the message properties.
kind	INTEGER. Indicates whether the message is binary (**1**) or text (**2**).
content	LONG BINARY. The content of the message. Text messages are encoded as UTF-8.
contentsize	BIGINT. The size of the message. For binary messages, this is the number of bytes. For text messages, this is the number of characters.
mluser	VARCHAR(128). The MobiLink user name that uniquely identifies a remote database.

Remarks	The owner of this table is ml_qa_user_group. Do not modify the contents of this table.

ml_qa_status_staging

This table is used only for QAnywhere applications. It is a staging table that is used when synchronizing status changes with the originating client.

Column	Description
msgid	VARCHAR(255). Primary key. Globally unique message identifier.
status	VARCHAR(255). The status of the message. Can be **pending**, **receiving**, **received**, **unreceivable**, **expired**, or **cancelled**. The default is **pending**.
statustime	TIMESTAMP. The last time the status was changed.
mluser	VARCHAR(128). The MobiLink user name that uniquely identifies a remote database.

Remarks

The owner of this table is ml_qa_user_group.

Do not modify the contents of this table.

ml_script

Stores the text of all scripts.

Column	Description
script_id	INTEGER. Primary key. The script_id column stores a unique integer that identifies the script.
script	TEXT. The script column stores the text of the script.
script_-language	VARCHAR(128). The script_language column stores the scripting language used for the script. The scripting language can be **sql**, **java**, or **dnet**.

ml_script_version

Stores the name, ID and comment of script associated with each script version.

Column	Description
version_id	INTEGER. Primary key. The version_id column stores a unique integer that identifies the version.
name	VARCHAR(128). The name column stores the arbitrary name given to the version.
description	TEXT. The description column stores the arbitrary description given to the version. The description is not used by MobiLink, but is useful for application-specific comments. For example, you could describe the purpose of a given script version.

ml_scripts_modified

Stores the last time script tables were changed. MobiLink server checks this table to determine if it must load new scripts.

Column	Description
last_modified	DATETIME. Primary key. The last_modified column stores the last time when the ml_script_version, ml_script or ml_-connection_script system table was altered.

ml_subscription

Keeps track of the log offsets (or progress) per subscription for Adaptive Server Anywhere remote databases.

Column	Description
user_id	INTEGER. Primary key. The user_id column references the user_id column of the ml_user table. This is the ID of a user who has subscribed to the publication.
subscription_id	VARCHAR(128). Primary key. The subscription_id is a number that is generated by Adaptive Server Anywhere on the remote.
publication_name	VARCHAR(128). Primary key. The publication_name stores the user-defined name for the publication.
progress	NUMERIC(20,0). The progress, also called the offset or state, refers to the point in the remote transaction log up to which all operations for the subscription have been uploaded. This column is used for version 8.0 and up databases. For version 7 databases, the progress is stored in the ml_user table.
last_download_time	TIMESTAMP. Indicates the last time a download was applied to the consolidated for a given user or subscription. The default is January 1, 1900, 00:00:00.
last_upload_time	TIMESTAMP. Indicates the last time an upload was applied to the consolidated for a given user or subscription. The default is January 1, 1900, 00:00:00.

Remarks

The **progress**, also called the **offset** or **state**, refers to a position in the transaction log of the remote database. It indicates the point to which all operations for the subscription have been uploaded and acknowledged. Dbmlsync uses the offset to decide what data to upload.

On the remote database, the offset is stored in the progress column of the SYS.SYSSYNC system table.

On the consolidated database, the offset is stored in the progress column of the ml_user table for version 7.x databases, and in the progress column of the ml_subscription table for version 8.0 and up databases.

ml_table

Stores names of remote tables. This list includes any table marked as a synchronized table in Sybase Central.

Column	Description
table_id	INTEGER. Primary key. The table_id column stores a unique integer identifying the table.
name	VARCHAR(128). The name column stores the arbitrary name given to the table.

ml_table_script

For a given script version, associates a table script with a given table and event.

Column	Description
version_id	INTEGER. Primary key. The version_id column references the version_id column of the ml_script_version table.
table_id	INTEGER. Primary key. The table_id column references the table_id column of the ml_table system table.
event	VARCHAR(128). Primary key. The event column stores the name of the event.
script_id	INTEGER. The script_id column references the script_id column of the ml_script table. The script is stored in the ml_script table.

Remarks

There is a view, ml_table_scripts, that makes it easier to view the contents of the ml_table_script MobiLink system table.

ml_user

Stores all registered MobiLink users, including their password and their synchronization state. The progress in this table is used only for UltraLite remotes or version 7 Adaptive Server Anywhere remotes.

Column	Description
user_id	INTEGER. Primary key. The user_id column stores a unique integer identifying the user. This value is only used internally by MobiLink.
name	VARCHAR(128). The name column stores the arbitrary name given to the user.
commit_state	INTEGER. The commit_state column stores the state.
progress	NUMERIC(20,0). The progress, also called the log offset or state, refers to the point in the transaction log up to which all operations for subscriptions have been uploaded and acknowledged. This column is used for version 7 databases. For version 8.0 and up databases, the progress is stored in the ml_subscription table.
hashed_password	BINARY(20). The hashed_password column stores the MobiLink user's password in obfuscated form. If there is no password, this value is NULL. (This is not recommended.)
last_download_time	TIMESTAMP. Indicates the last time a download was applied to the consolidated for a given user or subscription. The default is January 1, 1900, 00:00:00.
last_upload_time	TIMESTAMP. Indicates the last time an upload was applied to the consolidated for a given user or subscription. The default is January 1, 1900, 00:00:00.

Remarks

The **progess**, also called the **log offset** or **state**, refers to a position in the transaction log of the remote database. It indicates the point to which all operations for the subscription have been uploaded and acknowledged. dbmlsync uses the offset to decide what data to upload.

On the remote database, the offset is stored in the progress column of the SYS.SYSSYNC system table.

On the consolidated database, the offset is stored in the progress column of the ml_user table for version 7.x databases, and in the progress column of the ml_subscription table for version 8.0 and up databases.

CHAPTER 19

DataType Conversions

About this chapter

This chapter provides information about the conversion of data types that must take place when a MobiLink synchronization server communicates with a consolidated database that is not Adaptive Server Anywhere. The following tables identify these conversions.

If you are writing synchronization scripts in .NET languages or in Java, you may need to know how to map SQL data types to Java and .NET data types. For more information, see "SQL-.NET data types" on page 287 and "SQL-Java data types" on page 260.

Note
Only supported data types are presented in this chapter.

Contents

Adaptive Server Enterprise data mapping

The following table identifies how Adaptive Server Anywhere or UltraLite data types are mapped to Adaptive Server Enterprise data types.

Adaptive Server Anywhere or UltraLite data type	**Adaptive Server Enterprise data type**
bit	bit
tinyint	tinyint
smallint	smallint
int	int
integer	integer
decimal [defaults p=30, s=6]	numeric(30,6)
numeric [defaults p=30 s=6]	numeric(30,6)
float	real
real	real
double	float
smallmoney	numeric(10,4)
money	numeric(19,4)
date	datetime
time	datetime
timestamp	datetime
smalldatetime	datetime
datetime	datetime
char(n)	varchar(n)
character(n)	varchar(n)
varchar(n)	varchar(n)
character varying(n)	varchar(n)
long varchar	text
text	text
binary(n)	binary(n)

Adaptive Server Anywhere or UltraLite data type	Adaptive Server Enterprise data type
long binary	image
image	image
bigint	numeric(20,0)
uniqueidentifier	varchar(n) [1]

[1] *n* must be greater than 36.

IBM DB2 data mapping

The following table identifies how Adaptive Server Anywhere or UltraLite data types are mapped to IBM DB2 data types.

Adaptive Server Anywhere or UltraLite data type	IBM DB2 data type
bit	smallint
tinyint	smallint
smallint	smallint
int	int
integer	int
bigint	decimal(20,0)
char(1–4000)	varchar(n)
char(4001–32767)	long varchar
character(1–4000)	varchar(n)
character(4001–32767)	long varchar
varchar(1–4000)	varchar(n)
varchar(4001–32767)	long varchar
character varying(1–4000)	varchar(n)
character varying(4001–32767)	long varchar or CLOB(n)
long varchar	long varchar or CLOB(n)
text	long varchar
binary(1–4000)	varchar for bit data or BLOB(n)
binary(4001–32767)	long varchar for bit data or BLOB(n)
long binary	long varchar for bit data or BLOB(n)
image	long varchar for bit data or BLOB(n)
decimal [defaults p=30, s=6]	decimal(30,6)
numeric [defaults p=30 s=6]	decimal(30,6)
real	real
float	float

Adaptive Server Anywhere or UltraLite data type	IBM DB2 data type
double	float
smallmoney	decimal(10,4)
money	decimal(19,4)
date	date
time	time
smalldatetime	timestamp
datetime	timestamp
timestamp	timestamp
uniqueidentifier	varchar(n) [1]

[1] *n* must be greater than 36.

Oracle data mapping

The following table identifies how Adaptive Server Anywhere or UltraLite data types are mapped to Oracle data types.

Adaptive Server Anywhere or UltraLite data type	Oracle data type
bit	number(1,0)
tinyint	number(3,0)
smallint	number(5,0)
int	number(11,0)
integer	number(11,0)
bigint	number(20,0)
decimal(prec, scale)	number(prec, scale)
numeric(prec, scale)	number(prec, scale)
float	float
real	real
smallmoney	numeric(10,4)
money	number(19,4)
date	date
time	date
timestamp	date
smalldatetime	date
datetime	date
char(n)	varchar(n) or CLOB(n)
varchar(n)	varchar(n) or CLOB(n)
long varchar	CLOB
binary(n)	raw(n) or BLOB(n)
varbinary(n)	raw(n) or BLOB(n)
long binary	BLOB

Adaptive Server Anywhere or UltraLite data type	Oracle data type
uniqueidentifier	varchar2(n) [1]

[1] *n* must be greater than 36.

The LONG data types are deprecated in Oracle 8, 8i and 9i.

For Oracle LONG data types to synchronize properly, you must check the Oracle **Force Retrieval of Long Columns** ODBC option in the ODBC data source configuration dialog.

Microsoft SQL Server data mapping

The following table identifies how Adaptive Server Anywhere or UltraLite data types are mapped to Microsoft SQL Server data types.

Adaptive Server Anywhere or UltraLite data type	**Microsoft SQL Server data type**
bit	bit
tinyint	tinyint
smallint	smallint
int	int
integer	int
bigint	numeric(20,0) or bigint (SQL Server 2000 only)
decimal [defaults p=30, s=6]	decimal(30, 6)
numeric [defaults p=30 s=6]	numeric(30, 6)
float	float
real	real
smallmoney	smallmoney
money	money
date	datetime
time	datetime
timestamp	datetime
smalldatetime	datetime
datetime	datetime
char(n)	varchar(n) or text
character(n)	varchar(n)
varchar(n)	varchar(n) or text
long varchar	text
binary(n)	binary(n) or image
long binary	image

Adaptive Server Anywhere or UltraLite data type	Microsoft SQL Server data type
double	float
uniqueidentifier	uniqueidentifier

CHAPTER 20

Character Set Considerations

About this chapter

This chapter describes how to handle international language issues in MobiLink applications.

Contents

Character set considerations

Each character of text is represented in one or more bytes. The mapping from characters to binary codes is called the **character set encoding**. Some character sets used for languages with small alphabets, such as European languages, use a single-byte representation. Others, such as Unicode, use a double-byte representation. Because they use twice the storage space for each character, double-byte character sets can represent a much larger number of characters.

Conversion errors can occur or data can be lost when text using one character set must be translated to another character set. Not all characters can be represented in all character sets. In particular, single-byte character sets can represent a much smaller number of characters than multi-byte systems because of the limited number of codes available.

When the character set of your MobiLink remote database is the same as your consolidated database, character translation issues are avoided.

Text often needs to be sorted to build indexes and to prepare ordered result sets, such as directory listings. The **sort order** identifies the order of the characters. For example, a sort order typically states that the letter “a” comes before the letter “b” , which comes before the letter “c” .

Each database has a **collation sequence**. You set the collation sequence when you create the database, although how you do so can differ between database systems. The collation sequence defines both the character set and the sort order for that database.

Tip
Whenever possible, define the collation sequence of your remote database to be the same as that of your consolidated database. This arrangement reduces the chance of erroneous translations.

☞ For more information, see “Character sets in UltraLite” [*UltraLite Database User’s Guide,* page 43] and “International Languages and Character Sets” [*ASA Database Administration Guide,* page 319].

Character set translation during synchronization: Windows

During synchronization, characters may need to be translated from one character set to another. The following translations occur as characters are passed between the remote application and the consolidated database.

Character set translation during upload

The MobiLink client sends data to the MobiLink synchronization server using the character set of the remote database.

1. The MobiLink synchronization server communicates with the consolidated database using the Unicode ODBC API. To do so, the MobiLink synchronization server translates all characters received from the remote database into Unicode.

2. If necessary, the ODBC driver for the consolidated database server translates the characters from Unicode into the character set of your consolidated database. This translation is controlled solely by the ODBC driver for your consolidated database system. Hence, behavior can differ between two different database systems, particularly systems made by different manufacturers. MobiLink synchronization works with a number of database systems. Check the documentation of your particular consolidated server and ODBC driver for details.

Character set translation during download

1. The ODBC driver for the consolidated database system receives characters in the coding of the consolidated database. It translates these characters into Unicode to pass them through the Unicode API to the MobiLink synchronization server. This translation is controlled solely by the ODBC driver for your consolidated database system. Check the documentation of your particular consolidated server and ODBC driver for details.

2. The MobiLink synchronization server receives characters through the Unicode ODBC API. If the remote database uses a different character set, the MobiLink synchronization server translates the characters before downloading them.

Examples

- ♦ UltraLite applications on Windows CE devices use the Unicode character set.

  When you synchronize a Windows CE application, no character translation occurs within the MobiLink synchronization server. The server finds that data arriving from the application is already in Unicode and passes it directly to the ODBC driver. Similarly, no character set translation is necessary when downloading data.

- ♦ All Adaptive Server Anywhere databases and all UltraLite applications on platforms other than Windows CE use the character set determined by the collating sequence of the remote database.

  When you synchronize a remote database, the MobiLink synchronization server performs character set translations between the character set of the remote database and Unicode.

Controlling ODBC driver character set translation

Because most consolidated databases are unlikely to use Unicode, it is important to understand how the ODBC driver for your consolidated database system converts data to and from Unicode. Some ODBC drivers use the language settings of the machine running MobiLink to determine what character set to use. In these cases, it is best if the language and code-page settings of the machine running the MobiLink synchronization server match those of the consolidated database.

Other ODBC drivers, such as the driver for Sybase Adaptive Server Enterprise, allow each connection to use a specific character set. To avoid translation errors, the character set used by MobiLink should be set to match that of the consolidated database.

☞ For a detailed description of how character set translations take place in your consolidated database server's ODBC driver, consult that product's ODBC driver documentation.

Character set translation during synchronization: non-Windows

The ODBC drivers that iAnywhere Solutions provides on non-Windows platforms do not have a Unicode ODBC API. The MobiLink synchronization server exchanges data with the ODBC driver using the character set determined by the collating sequence of the remote database.

When the remote database is an UltraLite application running under Windows CE, the MobiLink synchronization server performs character set translation between Unicode and the character set being used with ODBC.

CHAPTER 21

iAnywhere Solutions ODBC Drivers

About this appendix

This appendix describes the ODBC drivers available for use with MobiLink.

Contents

ODBC drivers supported by MobiLink

The MobiLink synchronization server can work with a variety of consolidated databases and ODBC drivers, as shown in the table below. Some drivers, though compatible for use with MobiLink, may have functional restrictions associated with their use.

☞ For updated information and complete functional specifications, see *http://www.ianywhere.com/developer/technotes/odbc_mobilink.html*.

☞ For information about configuring ODBC drivers, see "Introduction to iAnywhere Solutions ODBC Drivers" [*ODBC Drivers for MobiLink and Remote Data Access*, page 1]. On UNIX, a standalone version of this book is installed in the *drivers* subdirectory of your SQL Anywhere Studio install directory.

Database	**ODBC Drivers**
Oracle 8i	iAnywhere Solutions 8 - Oracle 8, 8i & 9i ODBC Driver Merant DataDirect Connect ODBC Driver for Oracle
Oracle 9i	iAnywhere Solutions 8 - Oracle 8, 8i & 9i ODBC Driver Oracle 9i ODBC Driver
Microsoft SQL Server 7, Microsoft SQL Server 2000	Microsoft SQL Server ODBC Driver Merant DataDirect SQL Server Wire Protocol ODBC Driver
Sybase Adaptive Server Enterprise 11.5 or later	iAnywhere Solutions 8 - Sybase ASE ODBC Driver Sybase ASE ODBC Driver Merant DataDirect Sybase Wire Protocol ODBC Driver Merant DataDirect Connect ODBC Driver for Sybase ASE Merant Connect ODBC Driver for Sybase ASE
IBM DB2 UDB 7.1, 7.2	IBM DB2 UDB 7.1 ODBC driver IBM DB2 UDB 7.2 ODBC driver Merant DataDirect DB2 Wire Protocol ODBC Driver
Sybase Adaptive Server Anywhere 9	Adaptive Server Anywhere 9.0 ODBC Driver

CHAPTER 22

Deploying MobiLink Applications

About this appendix

This appendix describes how to deploy the MobiLink server and MobiLink clients in a production environment. It identifies the files required for deployment.

Check your license agreement
Redistribution of files is subject to your license agreement. No statements in this document override anything in your license agreement. Please check your license agreement before considering deployment.

Contents

Deployment overview

Deploying MobiLink applications involves the following activities:

- Deploy the MobiLink server into a production setting.
- Deploy any Adaptive Server Anywhere MobiLink clients.
- Deploy any UltraLite MobiLink clients.

This chapter describes the files you need to include in your application's install program for each of these items.

Deploying the MobiLink server

The simplest way to deploy a MobiLink synchronization server into a production environment is to install a licensed copy of SQL Anywhere Studio onto the production machine.

However, if you are redistributing a MobiLink synchronization server in a separate install program (subject to your license agreement), you may want to include only a subset of the files. In this case, you need to include the following files in your installation.

Notes

- Test on a clean machine before redistributing.
- Files must be installed within the SQL Anywhere Studio install directory.
- The files should be in the same directory, unless otherwise mentioned.
- When a location is given, the files must be copied into a directory of the same name.
- On UNIX, environment variables must be set for the system to be able to locate SQL Anywhere applications and libraries. It is recommended that you use the appropriate file for your shell, either *asa_config.sh* or *asa_config.csh* (located in the directory */opt/sybase/SYBSsa9/bin*) as a template for setting the required environment variables. Some of the environment variables set by the asa_config files include PATH, LD_LIBRARY_PATH, ASANY9, and ASANYSH9.
- To use Java synchronization logic, and to use the graphical administration tools (Sybase Central and the MobiLink Monitor), you must have JRE 1.4.2 installed.

Windows applications

Description	Windows files
MobiLink synchronization server	*charsets\unicode* *win32\dbmlsv9.dll* *win32\dbmlsrv9.exe* *win32\dbmsql9.dll* *win32\dbunic9.dll*
Language library	*win32\dblgen9.dll*[1]

Description	Windows files
Windows Performance Monitor support	*win32\dbmlctr9.dll*[2] *win32\dbmlctr9.h* *win32\dbmlctr9.ini*
Synchronization stream libraries (deploy the ones you use)	*win32\dbmlhttp9.dll* *win32\dbmlsock9.dll*
Java synchronization logic	*java\activation.jar*[3] *java\imap.jar*[3] *java\jodbc.jar* *java\log4j.jar*[3] *java\mailapi.jar*[3] *java\mlscript.jar* *java\mlsupport.jar* *java\pop3.jar*[3] *java\smtp.jar*[3] *win32\dbmjava9.dll* *win32\mljodbc9.dll*
.NET synchronization logic	*MobiLink\setup\dnet\mlDomConfig.xml* *win32\dbmdnet9.dll* *win32\dnetodbc9.dll* *win32\iAnywhere.MobiLink.dll* *win32\iAnywhere.MobiLink.Script.dll* *win32\iAnywhere.MobiLink.Script.xml* *win32\mlDomConfig.xsd*
Security option[4]	*win32\dbmlhttps.dll* *win32\dbmlhttpsfips9.dll* *win32\dbmlrsafips9.dll* *win32\dbmljrsa9.dll* *win32\dbmljtls9.dll* *win32\dbmlrsa9.dll* *win32\dbmltls9.dll*
Script files (deploy the ones for your consolidated database)	*MobiLink\setup* *MobiLink\upgrade*
iAnywhere Solutions ODBC drivers	*drivers*

Description	**Windows files**
dbmluser utility	*win32\dbmluser.exe*
dbmlstop utility	*win32\dbmlstop.exe*
Error names	*h\sserror.h*
MobiLink Monitor	*java\mlmon.jar* *shared\java\HelpManager11.jar* *shared\java\JComponents142.jar* *shared\java\jsyblib142.jar* *shared\win32\jsyblib142.dll* *ultralite\java\lib\ulrt.jar* *win32\dbmlmon.exe*
Online help for the MobiLink plug-in and Monitor	*java\dbmaen9.jar*[1]
MobiLink Redirector	*MobiLink\redirector*
Sybase Central	*shared\Sybase Central 4.3*
Sybase Central plug-in	*win32\dbmlput9.dll*
Notifier	*java\activation.jar*[3] *java\jodbc.jar* *java\log4j.jar* *java\mailapi.jar*[3] *java\mlnotif.jar* *java\mlscript.jar* *java\smtp.jar*[3]
QAnywhere	*java\log4j.jar* *java\qaconnector.jar*

Note: Files may be located in the *win32* or *ia64* directory, depending on your version of the software.

[1] For French, German, Japanese, and Chinese editions, substitute **en** with **fr**, **de**, **ja**, and **zh**, respectively.

[2] Your setup program must self-register this file.

[3] If you are redistributing an application, you must obtain these files directly from Sun.

[4] Transport-layer security requires that you obtain the separately-licensable SQL Anywhere Studio security option and is subject to export regulations. To order this component, see "Separately-licensable components" [*Introducing SQL Anywhere Studio*, page 5].

UNIX, Linux, and Macintosh applications

Description	**UNIX files**
MobiLink synchronization server	*bin/dbmlsrv9* *charsets/unicode* *lib/libdbodm9_r.so*[3] *lib/libdbmsql9_r.so*[3] *lib/libdbtasks9_r.so*[3] *lib/libdbunic9_r.so*[3]
Language library	*res/dblgen9.res*[1]
Synchronization stream libraries (deploy the ones you use)	*lib/libdbmlhttp9_r.so*[3] *lib/libdbmlsock9_r.so*[3]
Java synchronization logic	*java/activation.jar*[2] *java/imap.jar*[2] *java/jodbc.jar* *java/log4j.jar*[2] *java/mailapi.jar*[2] *java/mlscript.jar* *java/mlsupport.jar* *java/pop3.jar*[2] *java/smtp.jar*[2] *lib/libdbmjava9_r.so*[3] *lib/libmljodbc9.so*[3]
.NET synchronization logic	N/A

Description	UNIX files
Security option[4]	*lib/libdbmlhttps9_r.so*[3] *lib/libdbmlrsa9_r.so*[3] *lib/libdbmltls9_r.so*[3]
Script files (deploy the ones for your consolidated database)	*MobiLink/setup* *MobiLink/upgrade*
iAnywhere Solutions ODBC drivers	*drivers*
dbmluser utility	*bin/dbmluser*
dbmlstop utility	*bin/dbmlstop*
Error names	*h/sserror.h*
MobiLink Monitor	*bin/dbmlmon* *java/mlmon.jar* *shared/java/HelpManager11.jar* *shared/java/JComponents142.jar* *shared/java/jsyblib142.jar* *ultralite/java/lib/ulrt.jar*
MobiLink Redirector	*Mobilink/redirector/redirector.config* *MobiLink/redirector/java*
Online help for the MobiLink plug-in and MobiLink Monitor	*java/dbmaen9.jar*
Sybase Central	*shared/sybcentral43*
Sybase Central plug-in	*lib/libdbmlput9_r.so*[3]

Description	UNIX files
Notifier	*java/activation.jar*[2] *java/jodbc.jar* *java/log4j.jar* *java/mailapi.jar*[2] *java/mlnotif.jar* *java/mlscript.jar* *java/smtp.jar*[2]
QAnywhere	*java/log4j.jar* *java/qaconnector.jar*

[1] For French, German, Japanese, and Chinese editions, substitute **en** with **fr**, **de**, **ja**, and **zh**, respectively.

[2] If you are redistributing an application, you must obtain these files directly from Sun.

[3]For Solaris and Linux, the file extension is *.so.* For AIX, the file extension is *.a.* For HP, the file extension is *.sl.* For the Macintosh, the file extension is *.dylib.*

[4] Transport-layer security requires that you obtain the separately-licensable SQL Anywhere Studio security option and is subject to export regulations. To order this component, see "Separately-licensable components" [*Introducing SQL Anywhere Studio,* page 5].

Deploying Adaptive Server Anywhere MobiLink clients

For Adaptive Server Anywhere clients, you need to deploy an Adaptive Server Anywhere database server and the MobiLink client.

☞ For information about deploying Adaptive Server Anywhere databases, see “Deploying Databases and Applications” [*ASA Programming Guide*, page 519].

If you are redistributing MobiLink synchronization clients (subject to your license agreement) you need to include the following files in your installation in addition to those required for the Adaptive Server Anywhere database:

The files should be in the same directory, unless otherwise mentioned.

Windows applications

Description	Windows files
MobiLink synchronization client	*win32\dblgen9.dll*[1] *win32\dbmlsync.exe* *win32\dbtool9.dll* *Note*: *dbtool9.dll* is not required on Windows CE unless you are using the dbtools interface.
Dbmlsync integration component	MobiLink synchronization client files *dbmlsynccomg.dll* for the visual component or *dbmlsynccom.dll* for the non-visual component
Synchronization stream libraries (deploy the ones you use)	*win32\dbmlhttp9.dll* *win32\dbmlsock9.dll*
Security option[2]	*dbmlhttps9.dll* *dbmlhttpsfips9.dll*[3] *dbmlrsafips9.dll*[3] *dbmlrsa9.dll* *dbmltls9.dll*
ActiveSync and HotSync utilities	*win32\dbasinst.exe* *win32\dbcon9.exe*
Listener	*win32\dblgen9.dll*[1] *win32\dblsn.exe* *win32\lsn_udp.dll* *win32\lsn_swi510.dll* *win32\maac555.dll* *win32\maac750.dll* *win32\maac750r3.dll* *win32\mabridge.dll*

[1] For French, German, Japanese, and Chinese editions, substitute **en** with **fr**, **de**, **ja**, and **zh**, respectively.

[2] Transport-layer security requires that you obtain the separately-licensable SQL Anywhere Studio security option and is subject to export regulations. To order this component, see "Separately-licensable components"

[*Introducing SQL Anywhere Studio,* page 5].

[3] FIPS security does not apply to Windows CE.

UNIX, Linux, and Macintosh applications

Description	**UNIX files**
MobiLink synchronization client	*bin/dbmlsync* *res/dblgen9.res* *lib/libdbtool9.so*[1] *lib/libdbtool9_r.so*[1]
Synchronization stream libraries (deploy the ones you use)	*lib/libdbmlhttp9_r.so*[1] *lib/libdbmlsock9_r.so*[1]
Security option[2]	*lib/libdbmlhttps9_r.so*[1] *lib/libdbmlrsa9_r.so*[1] *lib/libdbmltls9_r.so*[1]

[1]For Solaris and Linux, the file extension is *.so.* For AIX, the file extension is *.a.* For HP, the file extension is *.sl.* For the Macintosh, the file extension is *.dylib.*

[2] Transport-layer security requires that you obtain the separately-licensable SQL Anywhere Studio security option and is subject to export regulations. To order this component, see "Separately-licensable components" [*Introducing SQL Anywhere Studio,* page 5].

Deploying UltraLite MobiLink clients

For UltraLite clients, the UltraLite runtime library or the UltraLite component includes the required synchronization stream functions. The UltraLite runtime library is compiled into your application. Deployment is subject to your license agreement.

☞ For more information, see the documentation for the UltraLite component or development model you are using.

Palm applications

Description	**Palm files**
Windows Listener configuration utility	*dblsncfg.exe*
Listener for Treo 180	*palm\Treo180\lsnT180.prc*
Listener for Kyocera 6035	*palm\Kyocera6035\lsnK6035.prc*

Deploying QAnywhere applications

QAnywhere applications require client and server files.

The QAnywhere client requires all files used by dbmlsync, in addition to the files listed below.

☞ For more information about dbmlsync, see "Deploying Adaptive Server Anywhere MobiLink clients" on page 557.

Windows applications

Description	Windows files
QAnywhere client	*qanywhere.db* *win32\iAnywhere.QAnywhere.Client.dll* or *ce\iAnywhere.-QAnywhere.Client.dll* (required for the C# interface only) *win32\qaagent.exe* or *ce\arm.30\qaagent.exe* or *ce\x86.30\qaagent.exe* *win32\qany9.dll* or *ce\arm.30\qany9.dll* or *ce\x86.30\qany9.dll*
Server side	*java\log4j.jar* *java\qaconnector.jar*

UNIX, Linux, and Macintosh applications

Description	UNIX files
Server side	*java/log4j.jar* *java/qaconnector.jar*

Registering the QAnywhere .NET API dll

The QAnywhere .NET API dll (*win32\iAnywhere.QAnywhere.Client.dll*) needs to be registered in the Global Assembly Cache on Windows (except WindowsăCE). The Global Assembly Cache lists all the registered programs on your machine. When you install SQL Anywhere Studio, the installation program registers it. On WindowsăCE you do not need to register the DLL.

If you are deploying QAnywhere, you must register the QAnywhere .NET API dll (*win32\iAnywhere.QAnywhere.Client.dll*) using the gacutil utility that is included with the .NET Framework.

Index