Migrating Your SQL Remote for ASE Environment to MobiLink

A whitepaper from iAnywhere Solutions, Inc., a subsidiary of Sybase, Inc.

Copyright ©2004 iAnywhere Solutions, Inc.

Contents

1.0 Assumptions and Pre-Requisites 2

1.1 Code and Documentation References 2

1.2 MobiLink Overview 2

1.3 Why Migrate? 3

1.4 Migration Goals 4

2.0 Changes at the Remote Database 4

2.1 New Software to Deploy to the Remote Site 5

2.2 Comparing SQL Remote and MobiLink 5

2.3 Migration Steps at the Remote Site 7

2.4 Possible Problems 8
- 2.4.1 Last SQL Remote Messages Sent are Lost 8
- 2.4.2 More Messages Generated by Consolidated Database 9
- 2.4.3 Solution 9

2.5 Changes to the Remote Application 9

3.0 Changes at the Consolidated Database 10

3.1 Setting Up the Consolidated Database 10

3.2 Generating Compatible Upload Scripts 11
- 3.2.1 Example Upload Scripts 11
- 3.2.2 Defining Your Synchronization Scripts 12

3.3 Generating Compatible Download Scripts 14
- 3.3.1 Filtering Out Previously Downloaded Rows 14
- 3.3.2 Filtering Out Previously Downloaded Rows using a Shadow Table . 15
- 3.3.3 Downloading Deletes 16
- 3.3.4 Partitioning Rows by Remote User 17
- 3.3.5 Downloading Deletes and Partitioning Data by Remote User 19
- 3.3.6 Partitioning Tables That Do Not Contain The Subscription Column . 20
- 3.3.7 Sharing Rows Among Several Subscriptions 26
- 3.3.8 Conflict Resolution Scripts 34

3.4 First Synchronization From Migrated Remote 37

4.0 Conclusion 38

Legal Notice 40
- Contact Us 40

iAnywhere A SYBASE COMPANY

1.0 Assumptions and Pre-Requisites

This document assumes that you have a working knowledge of SQL Remote for ASE, but not necessarily MobiLink. Much of the verbosity of this document, particularly in the code samples, is to ensure that those unfamiliar with MobiLink can still understand the technical details. SQL Remote concepts that are discussed but not explained include differences between remote and consolidated databases, publications, remote users, consolidated users and subscriptions, subscribe by columns, and default conflict resolution. Although this paper deals primarily with SQL Remote for ASE, many of the concepts can also be applied if you are migrating a SQL Remote for ASA environment to MobiLink.

1.1 Code and Documentation References

All the code in this document was tested using Adaptive Server Anywhere version 9.0.2.2451 running on Windows 2000 (Service Pack 3) and Adaptive Server Enterprise version 12.5.1 ESD #2 (EBF 11665), running on Windows 2000 (Service Pack 4). All the references to the Adaptive Server Anywhere documentation are also based on the documentation that shipped with Adaptive Server Anywhere version 9.0.2.2451. Although it's unlikely that anything in this document is specific to a particular version or build of Adaptive Server Anywhere or Adaptive Server Enterprise, it's always important to test any code in your environment before putting it into production.

1.2 MobiLink Overview

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 client opens a connection to a MobiLink synchronization server. During synchronization, the MobiLink client 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.

For more information on each of the components in a MobiLink synchronization system, you should consult the product documentation. The chapter entitled "Synchronization Basics" in the *MobiLink Administration Guide* gives an excellent overview for those looking for more information on how MobiLink works. It is recommended that you read this chapter before proceeding with this whitepaper, or at a minimum, have a quick glance over the section to ensure that you are

comfortable with the terminology and concepts that are discussed in this paper. Product documentation for all iAnywhere Solutions products is available online at *http://www.ianywhere.com/developer/product_manuals*.

1.3 Why Migrate?

If SQL Remote for ASE is meeting all of your current needs, there may be no immediate need for you to migrate. Just because something new (and possibly better) exists does not necessarily mean that a migration is required. If something isn't broken, there is no need to fix it! However, changing external factors may trigger a move to newer technology.

There is a lot of functionality in MobiLink that does not exist in SQL Remote for ASE. Following is a list of some of the major feature differences:

- In theory, MobiLink can use any ODBC-Compliant consolidated database, but SQL Remote is limited to Adaptive Server Enterprise and Adaptive Server Anywhere as consolidated databases. At the time this document was written, the supported consolidated databases for MobiLink are Adaptive Server Anywhere, Adaptive Server Enterprise, Microsoft SQL Server, Oracle, and IBM DB/2.

- MobiLink can synchronize to an Adaptive Server Anywhere remote database or an UltraLite database. SQL Remote for ASE can only replicate with an Adaptive Server Anywhere remote database.

- MobiLink gives you more flexibility when uploading and downloading data since you write the scripts that decide how data is applied to the consolidated and what data is sent to the remotes.

- There is less data latency in a MobiLink environment since upload and download can occur in a single synchronization. In a SQL Remote environment, messages are written to a shared message system, with no guarantee on when these messages will be picked up by the other side of the replicating system.

- MobiLink allows you to encrypt the synchronization stream easily using 128-bit encryption. SQL Remote messages are obfuscated as opposed to encrypted, and adding encryption to the messages involves writing a custom encoding and decoding DLL for SQL Remote.

- MobiLink has the ability to request the remote database to initiate a synchronization based on events that occur on the consolidated database. This is often referred to as **push synchronization**.

Sybase and iAnywhere Solutions provide three different technologies that allow you to move data from an Adaptive Server Enterprise database to an Adaptive Server Anywhere database.

1. Replication Server (connection based)

2. MobiLink (session based)

3. SQL Remote for ASE (message based)

For high-speed, two-way replication between a small number of always connected databases with low latency demands, Replication Server is the obvious choice. However, in a disconnected environment, MobiLink and SQL Remote for ASE occupy a very similar solution space.

In the next major release of SQL Anywhere Studio, iAnywhere Solutions has decided to announce the end of life for SQL Remote for ASE. The migration path for current SQL Remote for ASE users is to use MobiLink instead of SQL Remote for ASE. You should note that the end of life notification for SQL Remote for ASE in no way affects the status of SQL Remote for ASA. SQL Remote replication between Adaptive Server Anywhere databases continues to be a strong and viable product in SQL Anywhere Studio.

1.4 Migration Goals

When considering a migration from one product to another, and in keeping with one of the founding principles of SQL Anywhere Studio (no DBA required at deployed sites), a migration path from one iAnywhere Solutions product to another should also be easy to execute for your remote users. Remote users should not need to know that anything has changed, and there should be no training costs associated with the migration for your end users. The migration also needs to be gradual. It is unrealistic to assume that every deployed user in your system will perform an upgrade on a given day, and it's important to avoid a situation where current users cannot access new data just because they have not yet upgraded. A migration path that limits functionality is also unacceptable. The new system you implement should be able to do everything that the old system could do, and hopefully more. The remainder of this document adheres to the migration goals mentioned in this section.

The remainder of this document is divided into two sections. The first section deals with changes that need to be made at the remote site to migrate users who are currently using an Adaptive Server Anywhere remote database and running SQL Remote for ASA against their remote database to using the MobiLink client and synchronizing with a MobiLink consolidated database. The second section deals with changes that need to be made against the consolidated database.

2.0 Changes at the Remote Database

Because we're trying to make the migration for the remote user as easy as possible, the steps outlined below will *not* require a new database to be deployed to the remote user. The remote user can continue to use the existing remote database, which should make the migration much easier for the remote user.

2.1 New Software to Deploy to the Remote Site

It is possible that new Adaptive Server Anywhere software may need to be deployed if the *dbmlsync.exe* executable does not already exist on the remote database. If deploying new software to the remote users is necessary, you may also want to consider upgrading to a more recent version of Adaptive Server Anywhere. The only new Adaptive Server Anywhere files that need to be deployed to run the MobiLink client are dbmlsync.exe and the associated stream DLL (*dbmlsock9.dll* or *dbmlhttp9.dll*). For more information on deployment, see the chapter entitled “Deploying Databases and Applications” in the *ASA Programming Guide*.

2.2 Comparing SQL Remote and MobiLink

Before going into details about what changes need to be made to the remote database, let’s first discuss replication and synchronization concepts and how they are different or similar at the remote site.

SQL Remote	MobiLink
SQL Remote uses a publication on the remote database to determine which data needs to be sent to the consolidated database. The publications on the remote do not typically have subscribe by columns or subqueries, as all changes made on the remote are normally sent to the consolidated. A WHERE clause may be present to limit the rows sent from the remote to the consolidated.	A MobiLink client also uses a publication on the remote database to determine which data needs to be synchronized. The publication may also have a WHERE clause to limit rows sent to the consolidated database. In fact, the same publication used by a SQL Remote client can also be used by the MobiLink client.
SQL Remote clients have a consolidated user defined in the remote database to identify the database to which changes will be sent. This consolidated user subscribes to the publication that is defined in the remote database.	A MobiLink client has a synchronization user defined in the remote database. A synchronization subscription links a synchronization user to a publication.
Message system parameters are defined on the remote database so that SQL Remote knows where and how messages are written to a shared message system.	An Adaptive Server Anywhere MobiLink client connects directly to the MobiLink synchronization server, so instead of specifying the location of the message system, you specify the communication stream type you are using and the location of the MobiLink synchronization server.

Here is a sample that shows how to define SQL Remote information at the remote database.
`CREATE PUBLICATION p1 ( TABLE t1 );`
`GRANT PUBLISH to u1;`
`CREATEăREMOTEăMESSAGEăTYPE 'file'`
`   ADDRESS 'u1';`
`GRANT CONSOLIDATED TO cons TYPE`
`   file ADDRESS 'cons';`
`CREATE SUBSCRIPTION TO cons FOR p1;`
`SET REMOTE file OPTION`
`   "public"."root_directory" =`
`   'r:\\msgs';`
	Here is a sample that shows how to define MobiLink information at the remote database.
	`CREATE PUBLICATION p1 ( TABLE t1 );`
	`CREATE SYNCHRONIZATION USER u1;`
	`CREATE SYNCHRONIZATION SUBSCRIPTION`
	`  FOR u1`
	`   TO p1`
	`   TYPE 'TCPIP'`
	`          ADDRESS`
	`   'host=10.2.3.1;port=600'`
	`    OPTION 'sv=v1';`

When determining what changes to send to the consolidated database, SQL Remote scans the Adaptive Server Anywhere transaction log and filters operations from the transaction log that need to be replicated to the consolidated database. Each operation (including COMMIT statements) that is executed against the remote database is sent in a message to the consolidated database. In other words, if the same row is updated 1000 times, 1000 updates are sent to the consolidated database, and the order of operations is preserved between the remote and consolidated.	When determining what changes to send to the consolidated database, the MobiLink client scans the Adaptive Server Anywhere transaction log and filters operations from the transaction log that need to be synchronized to the consolidated database. Operations that occur on the same row that are scanned by the MobiLink client are coalesced into a single operation. 1000 updates on the same row would be sent as a single update. By default, all the changes from the remote are applied in a single transaction on the consolidated, and there is no way to guarantee the order of operations on the remote, unless each operation is its own transaction and transactional uploads are used.

In the sample code in the above table, it was *not* an accident that the synchronization user was defined with the same name as the publisher of the remote database (u1). The reason for this will become clear as we continue.

2.3 Migration Steps at the Remote Site

1. Run dbremote on the remote to send any outstanding changes to the consolidated database. This will also pick up any changes already sent from the consolidated and apply them to the remote database. Check the dbremote output log for errors before proceeding to the next step.

2. Drop the consolidated user. No more SQL Remote messages can now be received or applied by this remote.

3. Create a new synchronization user and subscription. All new changes made on the remote database will now be synchronized using MobiLink instead of SQL Remote. Use the *current publisher* of the remote database for the name of the synchronization user.

4. Run dbmlsync and perform your first synchronization.

The steps above can easily be automated in a Windows batch file. The Windows batch file and associated SQL file below are a simple implementation of the steps outlined above. The batch file provided below is very specific to the shell in the Windows NT family of operation systems.

Sample Batch File

```
@rem
@rem Run dbremote to send changes and pick up outstanding
         messages
@rem
start /wait dbremote -c "eng=remote;uid=DBA;pwd=SQL" -k -v -ot
         out.txt
@rem
@rem Parse the dbremote output file for errors
@rem
type out.txt | grep "E. " > check.txt
type out.txt | grep "Not Applying Messages" >> check.txt
type out.txt | grep "Not Applying Operations" >> check.txt
type out.txt | grep "Missing Message" >> check.txt
@rem
@rem See if check.txt contains any information in it, and if
@rem it does, do not execute migrate.sql
@rem Note: The FOR command should all be on a single line
@rem
FOR /F "tokens=3* delims= " %%A IN
    ('DIR check.txt /-C /N ^| FIND /I "check.txt"') DO SET
         ACTSIZE=%%A
IF 0 NEQ %ACTSIZE% GOTO error_exists
@rem
@rem Run the SQL File to migrate the remote to use MobiLink
@rem
dbisql -nogui -c " eng=remote;uid=DBA;pwd=SQL " read migrate.sql
goto end
```

```
:error_exists
@echo "An error occurred when running dbremote"
type check.txt
:end
```

Sample SQL File

```
create procedure do_migrate ()
begin
  declare @stmt long varchar;
  revoke consolidate from cons;
  set @stmt = 'create synchronization user ' || CURRENT
         PUBLISHER;
  execute immediate @stmt;
  set @stmt = 'CREATE SYNCHRONIZATION SUBSCRIPTION TO "p1" ';
  set @stmt = @stmt || 'FOR "' || CURRENT PUBLISHER || '" ';
  set @stmt = @stmt || 'TYPE ''TCPIP'' ';
  set @stmt = @stmt || 'ADDRESS ''host=10.2.3.1;port=600'' ';
  set @stmt = @stmt || 'option SV=''v1'' ';
  execute immediate @stmt;
end;
call do_migrate();
drop procedure do_migrate;
```

2.4 Possible Problems

2.4.1 Last SQL Remote Messages Sent are Lost

Given the above batch file and SQL file, there is still the possibility of data loss. In step one of the migration process, the final SQL Remote messages were sent to the consolidated database and the output file was parsed to ensure that no errors had occurred when SQL Remote ran, but nothing was done to ensure that the messages were successfully applied by the consolidated database. If those messages were lost or not applied on the consolidated database for some reason, because the consolidated user has already been dropped on the remote database, the messages will never be resent. Therefore, the operations in those messages would never be applied to the consolidated database if the messages were lost.

To get around this problem, you could choose only to drop the consolidated user if the log_sent value in the SYS.SYSREMOTEUSER table for the consolidated user was equal to the confirmed_received value. During this time, you'd need to ensure that no changes are made on the remote database since you don't want any more messages to be sent by SQL Remote that would cause the log_sent value for the consolidated user to change. Run SQL Remote in receive only mode on the remote, and after each time SQL Remote receives messages, compare the log_sent and confirmed_received values. When they are equal, you can guarantee that the message has been applied on the consolidated.

Note that this strategy is only really viable in a situation where the consolidated database's send frequency is quite frequent. If SQL Remote only runs on the consolidated database once a day, you might be waiting a long time for a

confirmation message from the consolidated database.

2.4.2 More Messages Generated by Consolidated Database

It is possible that additional messages could be generated by the consolidated database between running SQL Remote on the remote database during step one to receive your final messages and before your first dbmlsync run in step four. Since the process will likely be automated in a batch file, the window of opportunity in which this problem could occur is quite small. The chance of this happening is slightly higher if the send frequency on the consolidated database is quite frequent. Those concerned about this small window of opportunity could choose to increase the send frequency for SQL Remote on the consolidated to further reduce the chance of the problem occurring. Also note that the actual messages generated by SQL Remote on the consolidated will still exist in the message system. In the rare situation where the contents of a SQL Remote message need to be read, a call could be opened with iAnywhere Technical Support to complete this task.

2.4.3 Solution

You may have noticed that two possible problems were described, and the solution to one problem was to increase SQL Remote's send frequency, and the solution to the second problem was to decrease SQL Remote's send frequency on the consolidated. Obviously, both cannot be done.

The best way to guarantee that *no* data is lost is to extract a new remote database for the user. This paper attempts to give you an alternative to re-extracting all the remote databases. The process described in this paper has little impact on end users and entails very little work for the administrators of the replicating system, but there is a very small possibility of data loss.

2.5 Changes to the Remote Application

Depending on how you were running SQL Remote on the remote clients, you may need to change the application that ran dbremote to now run dbmlsync instead.

If dbremote were run as a service, a new service would have to be defined that ran dbmlsync instead of dbremote, and the old SQL Remote service would need to be removed or disabled. This process could be automated in a batch file using the dbsvc command line utility. If users clicked a batch file to initiate replication, the batch file would need to be changed to run dbmlsync instead of the SQL Remote.

If you were using the DBTOOLS API to call dbremote, you would need to change your code to now call DBSynchronizeLog() instead of DBRemoteSQL(), and define a new structure to pass into DBSynchronizeLog().

If your application spawned an external process (for example, dbremote) it would now spawn dbmlsync, so the code in the application would need to be changed.

If you are making changes to your application, you should note that the DBMLSync Integration Component, which was introduced in Adaptive Server Anywhere version 9.0.1, can be used to launch the synchronization process. The DBMLSync Integration Component is an ActiveX plug-in that can be imported into Visual Basic, PowerBuilder, Delphi, or any other application development tool that can utilize an ActiveX component. Note that there are third-party tools that allow you to import an ActiveX component into the .NET Compact Framework programming environment. There is both a visual and non-visual version of the DBMLSync Integration Component. The visual component brings up a dialog box very similar to the dbmlsync executable and allows you to set the inputs to the component. The non-visual component requires you to do more programming work, but gives you access to much more customization, including access to every row being uploaded and downloaded during synchronization. For more information about using the DBMLSync Integration Component, see the chapter in the *MobiLink Clients* manual entitled “DBMLSync Integration Component”.

3.0 Changes at the Consolidated Database

The following sections describe the changes that are required at the consolidated database.

3.1 Setting Up the Consolidated Database

In the *%ASANY%\MobiLink\setup* directory, a file named *syncase125.sql* exists that must be run against the consolidated database to be used with MobiLink. This script creates MobiLink system tables to store upload and download scripts, as well as tracking progress information for remote sites. Before running the script, you should modify the script and add a `use db_name` command at the beginning to ensure the right database is modified. This script is not typically run against the master database, but against the database that contains the tables to be synchronized. When you run this script, you should be connected to the Adaptive Server Enterprise server as the same user that MobiLink will connect with when performing synchronizations. This user will need select, insert, update, and delete permissions on all tables involved in synchronization, including the MobiLink system tables being created in the script. Also, if your synchronization scripts call any stored procedures, this user needs execute permissions on those stored procedures.

MobiLink connects to the consolidated database using ODBC, so you will need to create an ODBC DSN that will be used by MobiLink to connect to the consolidated database. Not all ODBC drivers are created equal, and for this reason, it’s important to read over the Recommended ODBC Drivers for MobiLink web page on the iAnywhere Solutions web site before creating your DSN. The link for this page is *http://www.ianywhere.com/developer/technotes/odbc_mobilink.html*. The SQL Anywhere Studio install process installs iAnywhere-branded DataDirect OBDC drivers for most supported consolidated databases. These drivers are tested prior

to release and we recommend using these drivers over any other ODBC driver.

3.2 Generating Compatible Upload Scripts

With SQL Remote, the messages that are generated and applied include actual SQL statements. When a MobiLink client generates an upload stream for the consolidated database, the upload stream does not include SQL statements. It includes a list of tables (and their structure) that are being uploaded, as well as a list of changes that are made to rows in those tables. If an update is uploaded, the before and after images of the row are included in the upload. If the operation being uploaded is an insert or delete, only the after or before images are included in the upload. You must define upload scripts that are used to determine what actions are taken for the data being passed up in the upload stream.

3.2.1 Example Upload Scripts

Let's assume the following table structure exists at both the remote and consolidated database:

```
CREATE TABLE t1 (
  pkey INTEGER PRIMARY KEY,
  c1 INTEGER,
  c2 VARCHAR(100),
  c3 NUMERIC(10,2)
)
go
```

There are four columns in the table, so for each row in the upload stream, the associated upload script will be called and the data in each column will be passed into the synchronization scripts. The data is represented in the scripts you write using question marks, which is very similar to the ODBC or JDBC APIs.

Upload_Insert script:

```
insert into t1 (pkey,c1,c2,c3) values ( ?, ?, ?, ? );
```

Upload_Update script:

```
update t1 set c1 = ?, c2 = ?, c3 = ? where pkey = ?;
```

Upload_Delete script:

```
delete from t1 where pkey = ?;
```

It is important to note the order of the parameters in these scripts. The upload_insert is passed all four columns in the order the columns exist, but the upload_update script is passed the data columns first, followed by the primary columns, so that it is easier to write the update statement since the primary key columns are always referenced last in update statements. The upload_delete script is only passed the primary key columns, and none of the data columns.

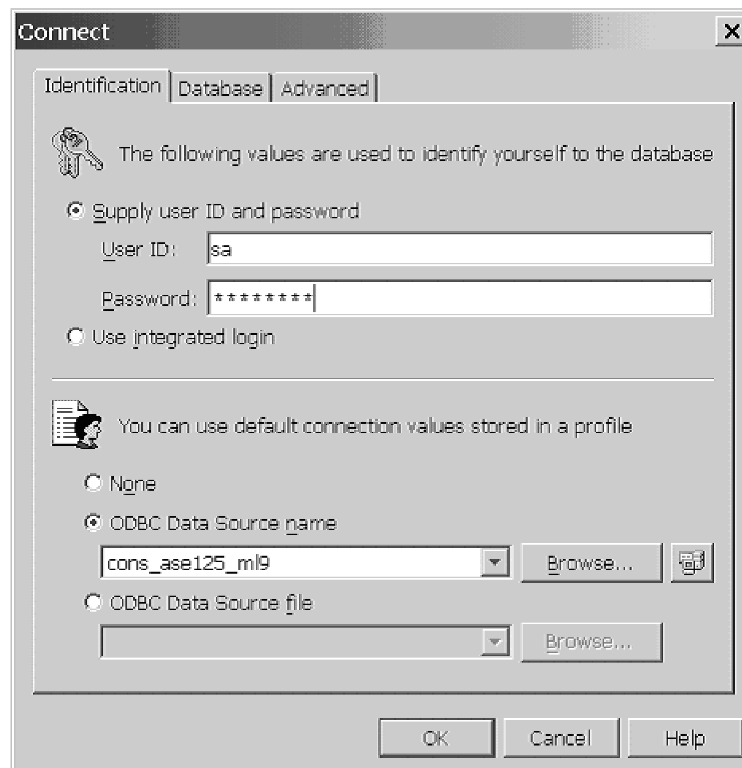
There are ways to automatically generate default scripts, which are usually acceptable for upload scripts. For more information on how to do this, consult the

MobiLink Administration Guide and search for a section entitled "Generating Scripts Automatically".

3.2.2 Defining Your Synchronization Scripts

Synchronization scripts can either be defined using Sybase Central, or can be added by executing stored procedures that were created when you ran the *syncase125.sql* scripts against your consolidated database.

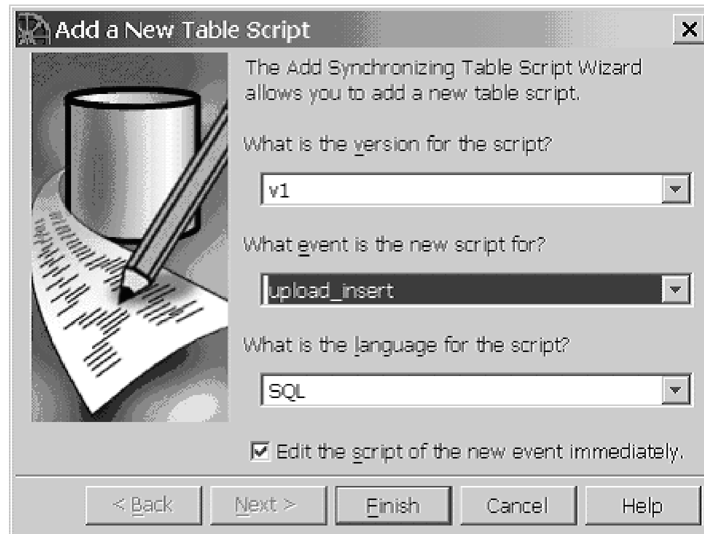
If using Sybase Central, you must first connect to the consolidated database. From the Tools menu, choose Connect and then choose to connect using the MobiLink Synchronization 9 plug-in. Choose the ODBC DSN you've created to connect to the consolidated database, optionally specify a user ID and password if they are not already specified in the DSN, and then click OK.



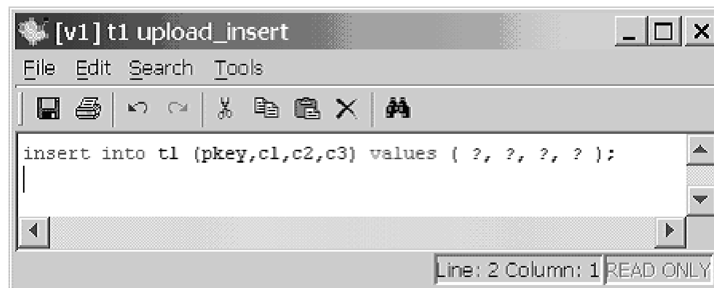
Scripts are organized into groups called **script versions**. By specifying a particular version, MobiLink clients can select which set of synchronization scripts will be used to process the upload stream and prepare the download stream. To add a script version to the consolidated database, open the Versions folder and double-click the Add Version wizard and follow the instructions to define a new script version.

The next step involves listing all the tables in the consolidated database that will be involved in synchronization. Open the Synchronized Tables folder then double-click the Add Synchronized Table wizard in the right frame and follow the instructions in the wizard for each table that will be synchronized. Once a table is

listed, click the table name in the Synchronized Tables folder in the left pane, and then click the Add Table Script wizard in the right pane. Choose the script version, event, and language for the new script you want to add, and also choose to edit the script immediately before clicking the Finish button.



Type the contents of your script in the editor window that is opened, and then choose File ➤ Close, choosing to save the script when prompted.



If you want to use the MobiLink stored procedures that were defined instead of using Sybase Central, the following SQL will define the three basic upload events for the t1 table being discussed. You could also choose to save the SQL below into a text file which could be checked in and maintained by your source control system.

```
exec ml_add_table_script 'v1', 't1', 'upload_insert',
  'insert into t1 (pkey,c1,c2,c3) values ( ?, ?, ?, ? )'
go
exec ml_add_table_script 'v1', 't1', 'upload_update',
  'update t1 set c1 = ?, c2 = ?, c3 = ? where pkey = ?'
go
exec ml_add_table_script 'v1', 't1', 'upload_delete',
  'delete from t1 where pkey = ?'
go
commit work
go
```

The remainder of this document will define synchronization scripts using stored procedures.

3.3 Generating Compatible Download Scripts

Your download scripts determine what data to send down to the remote clients. The simplest download script for a given table is simply a SELECT statement that sends all rows down to every remote client.

download_cursor script:

```
select pkey,c1,c2,c3 from t1
```

To define the script, you would execute:

```
exec ml_add_table_script 'v1', 't1', 'download_cursor',
  'select pkey,c1,c2,c3 from t1'
go
commit work
go
```

The problem with the above download_cursor is that every row is sent to every remote user for every synchronization, not just rows that were modified since the last download. One of the main goals of the migration was to ensure that you did not lose any functionality. With the very simple download cursor defined above, you have lost the ability to only send changes to the remotes.

3.3.1 Filtering Out Previously Downloaded Rows

There are two other pieces of information included in the upload stream during a synchronization request from the remote database to help us filter out previously downloaded rows. The name of the synchronizing user is passed up in the upload stream, as well as the last time that this user successfully began the download. This is referred to as the LastDownload timestamp. Even though this time is passed up from the remote, the LastDownload timestamp is generated by the consolidated database. This technique allows synchronization to function even if remotes are in different time zones from the consolidated database or the clocks are out of sync with the consolidated. Just before a download stream is generated by MobiLink, MobiLink asks the consolidated database for the time, and adds this time to the download stream being passed down to the remote. If the download stream is successfully applied on the remote, the system tables on the remote are updated and the new LastDownload timestamp is passed back up to MobiLink the next time the remote synchronizes.

You can use the last successful download time to ensure that only modified rows are sent down to the remote for each synchronization. To do this, you'll need to add a last_modified column of type datetime to each synchronizing table so that rows aren't downloaded each time. Ensure that the column has a DEFAULT value, which is defined as the current time.

```
ALTER TABLE t1 ADD last_modified datetime default getdate() null
go
```

You should set the initial value for this column as '1900-01-01 00:01:00' for all rows. The reason for this will become clear as we continue. You'll also need to add an update trigger to the table to ensure that the last_modified column is updated each time the row is modified.

```
UPDATE t1 SET last_modified = '1900-01-01 00:01:00'
go
commit
go
CREATE TRIGGER au_t1 ON t1 FOR UPDATE
AS
BEGIN
UPDATE t1
SET last_modified = getdate()
WHERE pkey IN ( SELECT pkey FROM inserted )
END
go
```

You can now re-write your download_cursor to only download rows since the last successful synchronization.

```
exec ml_add_table_script 'v1', 't1', 'download_cursor',
  'select pkey,c1,c2,c3 from t1 where last_modified >= ?'
go
commit work
go
```

It's important to note that the last_modified column *only* exists at the consolidated, and is *not* synchronized down to the remote. The schema may have changed on the consolidated, but not on the remote. You may now need to modify the publication definition on the consolidated database for current SQL Remote users. Since all columns are no longer being replicated, it is now necessary to include the list of columns that are being replicated to SQL Remote users if that was not being done before.

```
exec sp_add_article_col 'p1', 't1', 'pkey'
exec sp_add_article_col 'p1', 't1', 'c1'
exec sp_add_article_col 'p1', 't1', 'c2'
exec sp_add_article_col 'p1', 't1', 'c3'
go
```

3.3.2 Filtering Out Previously Downloaded Rows using a Shadow Table

In some situations, you cannot modify the base table because of the way existing applications access the database. In this scenario, instead of modifying the base table, you can add a shadow table that stores the last modified time for each row in the table.

```
CREATE TABLE t1_st (
  pkey integer primary key,
  last_modified datetime default getdate()
)
go
```

This table should be populated with one row for each row in the base table, and have last_modified values set to '1900-01-01 00:01:00'. Three triggers need to be defined to track changes on the base table if you are using shadow tables.

```
insert into t1_st select pkey, '1900-01-01 00:01:00' from t1
go
commit work
go

create trigger ai_t1 on t1 for insert as
begin
  insert into t1_st select pkey, getdate() from inserted
end
go

create trigger au_t1 on t1 for update as
begin
  update t1_st set last_modified = getdate()
  where pkey in ( select pkey from inserted )
end
go

create trigger ad_t1 on t1 for delete as
begin
  delete from t1_st where pkey in ( select pkey from deleted )
end
go
```

When using shadow tables, the download_cursor would now have to join two tables to filter out rows that have been previously downloaded:

```
exec ml_add_table_script 'v1', 't1', 'download_cursor',
  'select t1.pkey,t1.c1,t1.c2,t1.c3 from t1,t1_st
    where t1.pkey = t1_st.pkey
      and t1_st.last_modified >= ?'
go
commit work
go
```

For the remainder of this document, we will assume that the last_modified column was added to the base table, and shadow tables for the last_modified column will no longer be discussed.

3.3.3 Downloading Deletes

Since MobiLink does not scan the transaction log of the consolidated database, and since downloaded data is based on a SQL statement, another mechanism is needed if you want to be able to send deletes to the remote databases in the download stream. Another download cursor called the download_delete_cursor is used to track which rows should be deleted from the remote database. A table we refer to as a **shadow table** can be used to track which rows should be deleted on the remote database. Using the same table t1 from the previous sections, we'd

create a table called t1_del.

```
CREATE TABLE t1_del (
   pkey integer primary key,
   del_time datetime default getdate()
)
go
```

Now define a delete trigger on t1 that inserts a row into t1_del each time a row is deleted. This tracks the time the row was deleted, and this will allow us to write our download_delete_cursor for table t1.

```
create trigger ad_t1 on t1 for delete as
begin
   insert into t1_del select pkey, getdate() from deleted
end
go

exec ml_add_table_script 'v1', 't1', 'download_delete_cursor',
   'select pkey from t1_del where del_time >= ?'
go
```

Note that allowing deletes on shared rows at the remote databases introduces the same possible referential integrity (RI) problems that exist in SQL Remote.

For example, if you allow users to update and delete data at the remotes, then one remote user could update the row, while another deletes it. If the delete is applied first, then when the update arrives, it updates zero rows. Although this doesn't actually result in any data inconsistencies, since the delete will soon be sent down to the remote that executed the update, if that update contained important business information, it was just lost.

Another problem resulting from allowing deletes to occur at the remotes can be more serious if you allow remote users to delete parent records in a foreign key relationship. Assume one remote (RemoteA) deletes all the child records, followed by the parent record and synchronizes those changes to the consolidated. Another remote database (RemoteB) now inserts a child record that references the same parent record RemoteA deleted. RemoteA synchronizes, and the child and parent records are deleted from the consolidated. RemoteB synchronizes, and the insert of the child record causes a foreign key violation.

You need to be extremely careful if you allow deletes to occur at the remote site in a distributed environment.

3.3.4 Partitioning Rows by Remote User

With SQL Remote you partition rows by remote user by specifying a subscribe by column in the publication definition on the consolidated. With MobiLink, you use the same column in your download_cursor to partition the rows. We'll define a new table called t2 at this point, which includes a last_modified column to filter out previously downloaded rows, and also includes a rem_user column that we will use to partition rows based on the remote database that is synchronizing.

```
CREATE TABLE t2 (
  pkey INTEGER PRIMARY KEY,
  c1 INTEGER,
  c2 VARCHAR(100),
  c3 NUMERIC(10,2),
  last_modified datetime default getdate(),
  rem_user VARCHAR(128)
)
go
CREATE TRIGGER au_t2 ON t2 FOR UPDATE
AS
BEGIN
  UPDATE t2
  SET last_modified = getdate()
  WHERE pkey IN ( SELECT pkey FROM inserted )
END
go
sp_create_publication p1
sp_add_remote_table t2
sp_add_article p1, t2, NULL, rem_user
sp_add_article_col p1, t2, pkey
sp_add_article_col p1, t2, c1
sp_add_article_col p1, t2, c2
sp_add_article_col p1, t2, c3
sp_add_article_col p1, t2, rem_user
go
```

During synchronization, the name of the synchronizing user is passed up in the upload stream, this value is passed into most synchronization scripts, including the download_cursor. To create a download cursor on table t2 to match the functionality of the subscribe buy clause on the publication, you just need to specify the rem_user column in the WHERE clause of the download cursor.

```
exec ml_add_table_script 'v1', 't2', 'download_cursor',
   'select pkey,c1,c2,c3,rem_user from t2
   where last_modified >= ?
   and rem_user = ?'
go
```

With the above download_cursor, only rows meant for a given remote user that have been modified since the download will be synchronized to the remote database, just like SQL Remote. The question marks are positional. The first question mark in a download_cursor is the LastDownload timestamp. The second question mark in download_cursor is the name of the synchronizing user. If you only want to use the MobiLink User parameter you must still include two question marks in your download_cursor. The LastDownload timestamp can never be null, by adding the `? is not null` allows us to still reference two question marks, but does not change the results of the select statement.

```
exec ml_add_table_script 'v1', 't2', 'download_cursor',
   'select pkey,c1,c2,c3 from t2 where ? is not null and rem_
         user = ?'
go
```

Copyright ©2004 iAnywhere Solutions, Inc.

3.3.5 Downloading Deletes and Partitioning Data by Remote User

If distinct data is being kept at each remote site for a given table, and if you are downloading deletes to your remote users, you must specify which remote user is getting the deleted row in your download delete shadow table as well. We will continue to use the t2 table from previous sections, but will need to define a slightly different shadow table and delete trigger.

```
CREATE TABLE t2_del (
   pkey integer primary key,
   rem_user varchar(128) NOT NULL,
   del_time datetime default getdate()
)
go
```

The delete trigger on the t2 table needs to insert the name of the remote database that the delete will be sent to.

```
create trigger ad_t2 on t2 for delete as
begin
   insert into t2_del select pkey, rem_user, getdate() from
         deleted
end
```

We now need to consider the situation where the rem_user column on table t2 is changed. When this occurs, we need to send a insert of this row to the remote database associated with the new rem_user value, and send a delete of this row to the remote database associated with the old rem_user value. This modification will cause the update trigger to fire and set the last_modified column in table t2, so the insert of the row will be downloaded to the new remote database using the existing download_cursor. However, we still want a delete for that row to be sent down to the previous remote user that used to own the row. We can do that by modifying the update trigger on table t2 to insert a row into table t2_del for the remote user that used to own the row.

```
CREATE TRIGGER au_t2 ON t2 FOR UPDATE
AS
  IF UPDATE ( rem_user )
    BEGIN
      INSERT INTO t2_del
        SELECT deleted.pkey, deleted.rem_user, getdate()
          FROM deleted, inserted
         WHERE deleted.pkey = inserted.pkey
           AND deleted.rem_user <> inserted.rem_user
    END
  UPDATE t2
  SET last_modified = getdate()
  WHERE pkey IN ( SELECT pkey FROM inserted )
go
```

Finally, we need to make sure that the download_delete_cursor also only sends deletes that are meant for the correct user.

```
exec ml_add_table_script 'v1', 't2', 'download_delete_cursor',
   'select pkey from t2_del where del_time >= ? and rem_user =
         ?'
go
```

3.3.6 Partitioning Tables That Do Not Contain The Subscription Column

In many cases, the rows of a table need to be partitioned even when the subscription column does not exist in the table. This solution also needs to consider the problem we refer to as the territory re-alignment problem, where rows that do not contain the subscription column need to migrate from one remote to another as a result of a change on another table. This situation, and its solution when using SQL Remote for ASE, is well described in the Adaptive Server Anywhere documentation. If you are not familiar with this concept, you should read over the following section of the documentation before proceeding: "Partitioning tables that do not contain the subscription column" in the *SQL Remote User's Guide.*

3.3.6.1 The Schema for the Contact Example

In this section, we will start with the sample described in the documentation (the Contact example), and modify it so that MobiLink can perform the exact same functionality that SQL Remote for ASE performs. The schema used by the SQL Remote for Adaptive Server Enterprise sample is provided below and is followed by a discussion of how the schema needs to changed to be used by MobiLink as well.

```
--
-- Create the Tables
--
CREATE TABLE SalesRep (
   rep_key CHAR(12) NOT NULL,
   name CHAR(40) NOT NULL,
   PRIMARY KEY (rep_key)
)
go
CREATE TABLE Customer (
   cust_key CHAR(12) NOT NULL,
   name CHAR(40) NOT NULL,
   rep_key CHAR(12) NOT NULL,
   FOREIGN KEY ( rep_key )
   REFERENCES SalesRep,
   PRIMARY KEY (cust_key)
)
go
```

Copyright ©2004 iAnywhere Solutions, Inc.

```
CREATE TABLE Contact (
   contact_key CHAR( 12 ) NOT NULL,
   name CHAR( 40 ) NOT NULL,
   cust_key CHAR( 12 ) NOT NULL,
   subscription_list CHAR( 12 ) NULL,
   FOREIGN KEY ( cust_key )
   REFERENCES Customer ( cust_key ),
   PRIMARY KEY ( contact_key )
)
go

--
-- Define the SQL Remote Definitions
--
exec sp_add_remote_table 'SalesRep'
exec sp_add_remote_table 'Customer'
exec sp_add_remote_table 'Contact'
go
exec sp_create_publication 'SalesRepData'
go
exec sp_add_article 'SalesRepData', 'SalesRep'
exec sp_add_article 'SalesRepData', 'Customer', NULL, 'rep_key'
exec sp_add_article 'SalesRepData',
                    'Contact', NULL, 'subscription_list'
go

--
-- Set the subscription_list column when a new row is added
-- to the Contact table
--
CREATE TRIGGER set_contact_sub_list
ON Contact
FOR INSERT
AS
BEGIN
   UPDATE Contact
   SET Contact.subscription_list = (
         SELECT rep_key
         FROM Customer
         WHERE Contact.cust_key = Customer.cust_key )
   WHERE Contact.contact_key IN ( SELECT contact_key FROM
         inserted )
END
go

--
-- Maintain the subscription_list column on the Contact table
-- when the cust_key column is modified
--
CREATE TRIGGER update_contact_sub_list
ON Contact
FOR UPDATE
AS
IF UPDATE ( cust_key )
   BEGIN
      UPDATE Contact
      SET subscription_list = Customer.rep_key
      FROM Contact, Customer
      WHERE Contact.cust_key = Customer.cust_key
   END
go
```

```
--
-- Maintain the subscription_list column on the Contact table
-- when the rep_key column on the Customer table is changed
--
CREATE TRIGGER transfer_contact_with_customer
ON Customer
FOR UPDATE
AS
IF UPDATE ( rep_key )
BEGIN
   UPDATE Contact
   SET Contact.subscription_list = (
      SELECT rep_key
      FROM Customer
      WHERE Contact.cust_key = Customer.cust_key )
   WHERE Contact.cust_key IN ( SELECT cust_key FROM inserted )
END
go
```

3.3.6.2 Modify the Schema To Use MobiLink

In this section, we'll modify the schema from the previous section to include the last_modified column in each table, add a shadow table to handle downloading deletes, and write the MobiLink scripts to handle uploads and downloads.

```
CREATE TABLE SalesRep_del (
  rep_key CHAR( 12 ) PRIMARY KEY,
  del_time DATETIME DEFAULT getdate()
)
go
ALTER TABLE SalesRep ADD last_modified datetime default
         getdate() null
go
UPDATE SalesRep SET last_modified = '1900-01-01 00:01:00'
go
commit
go
CREATE TRIGGER ad_SalesRep on SalesRep for delete as
BEGIN
  INSERT INTO SalesRep_del SELECT rep_key, getdate() FROM
         deleted
END
go
CREATE TRIGGER au_SalesRep ON SalesRep FOR UPDATE
AS
BEGIN
  UPDATE SalesRep
  SET last_modified = getdate()
  WHERE rep_key IN ( SELECT rep_key FROM inserted )
END
go
```

```
exec sp_add_article_col 'SalesRepData', 'SalesRep', 'rep_key'
exec sp_add_article_col 'SalesRepData', 'SalesRep', 'name'
go
exec ml_add_table_script 'v1', 'SalesRep', 'upload_insert',
  'insert into SalesRep values ( ?, ? ) '
go
exec ml_add_table_script 'v1', 'SalesRep', 'upload_update',
  'update SalesRep set name = ? where rep_key = ? '
go
exec ml_add_table_script 'v1', 'SalesRep', 'upload_delete',
  'delete from SalesRep where rep_key = ? '
go
exec ml_add_table_script 'v1', 'SalesRep', 'download_cursor',
  'select rep_key, name from SalesRep where last_modified >= ? '
go
exec ml_add_table_script 'v1', 'SalesRep', 'download_delete_
         cursor',
  'select rep_key from SalesRep_del where del_time >= ? '
go
commit work
go
CREATE TABLE Customer_del (
  cust_key CHAR( 12 ) PRIMARY KEY,
  rep_key CHAR(12) NOT NULL,
  del_time DATETIME DEFAULT getdate()
)
go
ALTER TABLE Customer ADD last_modified datetime default
         getdate() null
go
UPDATE Customer SET last_modified = '1900-01-01 00:01:00'
go
commit
go
CREATE TRIGGER ad_Customer ON Customer FOR DELETE AS
BEGIN
  INSERT INTO Customer_del
    SELECT cust_key, rep_key, getdate() FROM deleted
END
go
DROP TRIGGER transfer_contact_with_customer
go
CREATE TRIGGER transfer_contact_with_customer
ON Customer
FOR UPDATE
AS
  IF UPDATE ( rep_key )
  BEGIN
     UPDATE Contact
     SET Contact.subscription_list = (
        SELECT rep_key
        FROM Customer
        WHERE Contact.cust_key = Customer.cust_key )
     WHERE Contact.cust_key IN ( SELECT cust_key FROM inserted )
     INSERT INTO Customer_del
        SELECT deleted.cust_key, deleted.rep_key, getdate()
          FROM deleted, inserted
         WHERE deleted.cust_key = inserted.cust_key
           AND deleted.rep_key <> inserted.rep_key
```

```
   END
   UPDATE Customer
   SET last_modified = getdate()
   WHERE cust_key IN ( SELECT cust_key FROM inserted )
go
exec sp_add_article_col 'SalesRepData', 'Customer', 'cust_key'
exec sp_add_article_col 'SalesRepData', 'Customer', 'name'
exec sp_add_article_col 'SalesRepData', 'Customer', 'rep_key'
go
exec ml_add_table_script 'v1', 'Customer', 'upload_insert',
  'insert into Customer (cust_key,name,rep_key) values ( ?, ?, ?
         ) '
go
exec ml_add_table_script 'v1', 'Customer', 'upload_update',
  'update Customer set name = ?, rep_key = ? where cust_key = ?
         '
go
exec ml_add_table_script 'v1', 'Customer', 'upload_delete',
  'delete from Customer where cust_key = ? '
go
exec ml_add_table_script 'v1', 'Customer', 'download_cursor',
  'select cust_key, name, rep_key from Customer
   where last_modified >= ? and rep_key = ?'
go
exec ml_add_table_script 'v1', 'Customer', 'download_delete_
         cursor',
  'select cust_key from Customer_del
   where del_time >= ? and rep_key = ?'
go
commit work
go

CREATE TABLE Contact_del (
  contact_key CHAR( 12 ) PRIMARY KEY,
  subscription_list CHAR(12) NOT NULL,
  del_time DATETIME DEFAULT getdate()
)
go
ALTER TABLE Contact ADD last_modified datetime default getdate()
         null
go
UPDATE Contact SET last_modified = '1900-01-01 00:01:00'
go
commit
go
CREATE TRIGGER ad_Contact ON Contact FOR DELETE AS
BEGIN
  INSERT INTO Contact_del
    SELECT contact_key, subscription_list, getdate() FROM
         deleted
END
go
DROP TRIGGER update_contact_sub_list
go
```

```
CREATE TRIGGER update_contact_sub_list
ON Contact
FOR UPDATE
AS
  IF UPDATE ( cust_key )
     BEGIN
        UPDATE Contact
        SET subscription_list = Customer.rep_key
        FROM Contact, Customer
        WHERE Contact.cust_key = Customer.cust_key
        INSERT INTO Contact_del
          SELECT deleted.contact_key,
                 deleted.subscription_list,
                 getdate()
            FROM deleted, Customer c_del, inserted, Customer c_
         ins
           WHERE deleted.cust_key = c_del.cust_key
             AND inserted.cust_key = c_ins.cust_key
             AND deleted.contact_key = inserted.contact_key
             AND deleted.cust_key <> inserted.cust_key
             AND c_del.rep_key <> c_ins.rep_key
     END
  UPDATE Contact
  SET last_modified = getdate()
  WHERE contact_key IN ( SELECT contact_key FROM inserted )
go
exec sp_add_article_col 'SalesRepData', 'Contact', 'contact_key'
exec sp_add_article_col 'SalesRepData', 'Contact', 'name'
exec sp_add_article_col 'SalesRepData', 'Contact', 'cust_key'
exec sp_add_article_col 'SalesRepData', 'Contact',
         'subscription_list'
go
exec ml_add_table_script 'v1', 'Contact', 'upload_insert',
  'insert into Contact (contact_key, name, cust_key,
         subscription_list)
   values ( ?, ?, ?, ? ) '
go
exec ml_add_table_script 'v1', 'Contact', 'upload_update',
  'update Contact set name = ?, cust_key = ?, subscription_list
         = ?
   where contact_key = ? '
go
exec ml_add_table_script 'v1', 'Contact', 'upload_delete',
  'delete from Contact where contact_key = ? '
go
exec ml_add_table_script 'v1', 'Contact', 'download_cursor',
  'select contact_key, name, cust_key, subscription_list from
         Contact
   where last_modified >= ? and subscription_list = ?'
go
exec ml_add_table_script 'v1', 'Contact', 'download_delete_
         cursor',
  'select contact_key from Contact_del
   where del_time >= ? and subscription_list = ?'
go
commit work
go
```

Everything in this section has been discussed before, with the exception of the

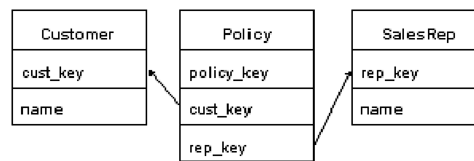
update trigger on the Contact table, and the fact that no entries are added to Contact_del table when an update on rep_key is performed on the Customer table.

In the update trigger on the Contact table, we need to send a delete down to the remote who no longer owns the row as a result of the change to the cust_key table. The important thing to note is that we can't automatically assume that the value in the subscription_list column of the deleted table should receive a delete, even if the cust_key changed. If two customers are represented by a single sales representative, and a contact is modified so that they move between these two customers, then a delete should *not* be sent. We need to verify that the change will actually move the contact from one rep_key to another. For that reason, we need to join the Customer table to the deleted table and another instance of the Customer table to the inserted table, and compare the rep_key values in the two instances of the Customer table to see if a delete really needs to be sent.

The other outstanding issue is how deletes are sent down to the remotes on the Contact table where a change occurs in the rep_key column on the Customer table. The update trigger on the Customer table ensures that a row is added to the Customer_del table for the old rep_key value, but no values are added to the Contact_del table. This is because dbmlsync (and UltraLite) will silently handle RI violations on the remote. When the delete on the Customer table occurs on the remote and an RI violation is about to occur when the commit occurs at the end of applying the download stream, the rows in the Contact table are deleted by dbmlsync.

3.3.7 Sharing Rows Among Several Subscriptions

This section discusses the situation where rows at the remote may exist in several different remote databases. As in the previous section, the remote user name will not necessarily exist in a table, but we still need a way to maintain which rows belong to which remotes when a many-to-many relationship exists between two tables. The documentation describes a sample using Policies, and determining which rows in the customer table should be sent to each remote database.



The Policy table has a row for each of a set of policies. Each policy is drawn up for a customer by a particular sales representative. There is a many-to-many relationship between customers and sales representatives, and there may be several policies drawn up between a particular rep/customer pair. Any row in the Customer table may need to be shared with none, one, or several sales representatives. The schema to solve this problem with SQL Remote for ASE is provided in the next section, followed by an explanation of how to modify the schema to work with MobiLink.

3.3.7.1 The Schema for the Policy Example

One thing to note about the sample provided in the documentation is that it does not take into account the fact that an update can occur on the rep_key column in the Policy table.

```
--
-- Create Tables
--
CREATE TABLE SalesRep (
   rep_key CHAR( 12 ) NOT NULL,
   name CHAR( 40 ) NOT NULL,
   PRIMARY KEY ( rep_key )
)
go
CREATE TABLE Customer (
   cust_key CHAR( 12 ) NOT NULL,
   name CHAR( 40 ) NOT NULL,
   subscription_list VARCHAR( 255 ) NULL,
   PRIMARY KEY ( cust_key )
)
go
CREATE TABLE Policy (
   policy_key   INTEGER NOT NULL,
   cust_key CHAR( 12 ) NOT NULL,
   rep_key CHAR( 12 ) NOT NULL,
   FOREIGN KEY ( cust_key ) REFERENCES Customer ( cust_key ),
   FOREIGN KEY ( rep_key ) REFERENCES SalesRep ( rep_key ),
   PRIMARY KEY ( policy_key )
)
go
--
-- Define the SQL Remote Definitions
--
exec sp_add_remote_table 'SalesRep'
exec sp_add_remote_table 'Policy'
exec sp_add_remote_table 'Customer'
go
exec sp_create_publication 'SalesRepData'
exec sp_add_article 'SalesRepData', 'SalesRep'
exec sp_add_article 'SalesRepData', 'Policy', NULL, 'rep_key'
exec sp_add_article 'SalesRepData', 'Customer', NULL,
         'subscription_list'
exec sp_add_article_col 'SalesRepData', 'Customer', 'cust_key'
exec sp_add_article_col 'SalesRepData', 'Customer', 'name'
go
--
-- This stored procedure is used to maintain the subscription_
         list
-- column in the Customer Table
--
CREATE PROCEDURE SubscribeCustomer
  @cust_key CHAR(12)
AS
BEGIN
   --  Rep returns the rep list for customer @cust_key
   DECLARE Rep CURSOR FOR
      SELECT DISTINCT RTRIM( rep_key )
      FROM Policy
      WHERE cust_key = @cust_key
```

iAnywhere
A SYBASE COMPANY

```
 DECLARE @rep_key CHAR(12)
 DECLARE @subscription_list VARCHAR(255)
   -- build comma-separated list of rep_key
 -- values for this Customer
 OPEN Rep
 FETCH Rep INTO @rep_key
 IF @@sqlstatus = 0 BEGIN
   SELECT @subscription_list = @rep_key
   WHILE 1=1 BEGIN
      FETCH Rep INTO @rep_key
      IF @@sqlstatus != 0 BREAK
      SELECT @subscription_list =
         @subscription_list + ',' + @rep_key
   END
 END
 ELSE BEGIN
   SELECT @subscription_list = ''
 END
 -- update the subscription_list in the
 -- Customer table
 UPDATE Customer
 SET subscription_list = @subscription_list
 WHERE cust_key = @cust_key
END
go

--
-- Maintain the subscription_list column on the Customer
-- Customer table when a new Policy is created
--
CREATE TRIGGER InsPolicy
ON Policy
FOR INSERT
AS
BEGIN
   -- Cust returns those customers inserted
   DECLARE Cust CURSOR FOR
      SELECT DISTINCT cust_key
      FROM inserted
   DECLARE @cust_key CHAR(12)
   OPEN Cust
   -- Update the rep list for each Customer
   -- with a new rep
   WHILE 1=1 BEGIN
      FETCH Cust INTO @cust_key
      IF @@sqlstatus != 0 BREAK
      EXEC SubscribeCustomer @cust_key
   END
END
go
--
-- Maintain the subscription_list column on the Customer
-- Customer table when a Policy is deleted
--
CREATE TRIGGER DelPolicy
ON Policy
FOR DELETE
AS
BEGIN
```

Copyright ©2004 iAnywhere Solutions, Inc.

```
   -- Cust returns those customers deleted
   DECLARE Cust CURSOR FOR
      SELECT DISTINCT cust_key
      FROM deleted
   DECLARE @cust_key CHAR(12)
   OPEN Cust
   -- Update the rep list for each Customer
   -- losing a rep
   WHILE 1=1 BEGIN
      FETCH Cust INTO @cust_key
      IF @@sqlstatus != 0 BREAK
      EXEC SubscribeCustomer @cust_key
   END
END
go
```

3.3.7.2 Modify Schema To Use MobiLink

In keeping with the simplification in the documentation for this example, we'll also assume that no updates can take place on the cust_key or rep_key column in the Policy table. In theory, that meant that the update trigger on the Policy table wasn't needed, but it was included here for completeness.

```
CREATE TABLE SalesRep_del (
   rep_key CHAR( 12 ) PRIMARY KEY,
   del_time DATETIME DEFAULT getdate()
)
go
ALTER TABLE SalesRep ADD last_modified datetime default
         getdate() null
go
UPDATE SalesRep SET last_modified = '1900-01-01 00:01:00'
go
commit
go
CREATE TRIGGER ad_SalesRep on SalesRep for delete as
 BEGIN
   INSERT INTO SalesRep_del SELECT rep_key, getdate() FROM
         deleted
 END
go
CREATE TRIGGER au_SalesRep ON SalesRep FOR UPDATE
AS
BEGIN
  UPDATE SalesRep
  SET last_modified = getdate()
  WHERE rep_key IN ( SELECT rep_key FROM inserted )
END
go
exec sp_add_article_col 'SalesRepData', 'SalesRep', 'rep_key'
exec sp_add_article_col 'SalesRepData', 'SalesRep', 'name'
go
exec ml_add_table_script 'v1', 'SalesRep', 'upload_insert',
  'insert into SalesRep values ( ?, ? ) '
go
exec ml_add_table_script 'v1', 'SalesRep', 'upload_update',
  'update SalesRep set name = ? where rep_key = ? '
go
```

```
exec ml_add_table_script 'v1', 'SalesRep', 'upload_delete',
   'delete from SalesRep where rep_key = ? '
go
exec ml_add_table_script 'v1', 'SalesRep', 'download_cursor',
   'select rep_key, name from SalesRep where last_modified >= ?
         '
go
exec ml_add_table_script 'v1', 'SalesRep', 'download_delete_
         cursor',
   'select rep_key from SalesRep_del where del_time >= ? '
go
commit work
go
CREATE TABLE Customer_del (
  cust_key CHAR( 12 ) NOT NULL,
  rep_key CHAR( 12 ) NOT NULL,
  del_time DATETIME DEFAULT getdate(),
  PRIMARY KEY ( cust_key, rep_key, del_time )
)
go
ALTER TABLE Customer ADD last_modified datetime default
         getdate() null
go
UPDATE Customer SET last_modified = '1900-01-01 00:01:00'
go
commit
go
CREATE TRIGGER au_Customer ON Customer FOR UPDATE
AS
  UPDATE Customer
  SET last_modified = getdate()
  WHERE cust_key IN ( SELECT cust_key FROM inserted )
go
exec ml_add_table_script 'v1', 'Customer', 'upload_insert',
  'insert into Customer (cust_key,name) values ( ?, ? ) '
go
exec ml_add_table_script 'v1', 'Customer', 'upload_update',
  'update Customer set name = ? where cust_key = ? '
go
exec ml_add_table_script 'v1', 'Customer', 'upload_delete',
  'delete from Customer where cust_key = ? '
go
exec ml_add_table_script 'v1', 'Customer', 'download_cursor',
   'select cust_key, name from Customer
    where last_modified >= ? and charindex( ?, subscription_list
         ) > 0'
go
exec ml_add_table_script 'v1', 'Customer', 'download_delete_
         cursor',
   'select cust_key from Customer_del
    where del_time >= ? and rep_key = ?'
go
commit work
go
CREATE TABLE Policy_del (
  policy_key INTEGER PRIMARY KEY,
  rep_key CHAR( 12 ) NOT NULL,
  del_time DATETIME DEFAULT getdate()
)
go
```

```
ALTER TABLE Policy ADD last_modified datetime default getdate()
         null
go
UPDATE Policy SET last_modified = '1900-01-01 00:01:00'
go
exec sp_add_article_col 'SalesRepData', 'Policy', 'policy_key'
exec sp_add_article_col 'SalesRepData', 'Policy', 'cust_key'
exec sp_add_article_col 'SalesRepData', 'Policy', 'rep_key'
go
COMMIT
go

CREATE TRIGGER au_Policy ON Policy FOR UPDATE
AS
  UPDATE Policy
  SET last_modified = getdate()
  WHERE policy_key IN ( SELECT policy_key FROM inserted )
go
DROP TRIGGER DelPolicy
go

CREATE TRIGGER DelPolicy
ON Policy
FOR DELETE
AS
BEGIN
   -- Cust returns those customers deleted
   DECLARE Cust CURSOR FOR
      SELECT DISTINCT cust_key
      FROM deleted
   DECLARE CustRep CURSOR FOR
      SELECT cust_key, rep_key
      FROM deleted
   DECLARE @cust_key CHAR(12)
   DECLARE @rep_key CHAR(12)

   -- Add a row to Policy_del table
   INSERT INTO Policy_del
      SELECT policy_key, rep_key, getdate() FROM deleted

   -- See if we need to add any rows to the Customer_del table
   OPEN CustRep
   WHILE 1=1
   BEGIN
      FETCH CustRep INTO @cust_key, @rep_key
      IF @@sqlstatus != 0 BREAK
      -- See if we just deleted the last policy that link this
      -- customer to this sales rep
      IF NOT EXISTS ( SELECT 1 FROM Policy
                       WHERE cust_key = @cust_key
                         AND rep_key = @rep_key )

      BEGIN
         INSERT INTO Customer_del
         VALUES ( @cust_key, @rep_key, getdate() )
      END
   END
```

```
   OPEN Cust
   -- Update the rep list for each Customer
   -- losing a rep
   WHILE 1=1 BEGIN
      FETCH Cust INTO @cust_key
      IF @@sqlstatus != 0 BREAK
      EXEC SubscribeCustomer @cust_key
   END
END
go
exec ml_add_table_script 'v1', 'Policy', 'upload_insert',
   'insert into Policy (policy_key,cust_key,rep_key) values ( ?,
         ?, ? )'
go
exec ml_add_table_script 'v1', 'Policy', 'upload_update',
   'update Policy set cust_key = ?, rep_key = ?
     where policy_key = ? and 1=0'
go
exec ml_add_table_script 'v1', 'Policy', 'upload_delete',
   'delete from Policy where policy_key = ? '
go
exec ml_add_table_script 'v1', 'Policy', 'download_cursor',
   'select policy_key, cust_key, rep_key from Policy
    where last_modified >= ? and rep_key = ?'
go
exec ml_add_table_script 'v1', 'Policy', 'download_delete_
         cursor',
   'select policy_key from Policy_del
    where del_time >= ? and rep_key = ?'
go
commit work
go
DROP PROCEDURE SubscribeCustomer
go
CREATE PROCEDURE SubscribeCustomer
  @cust_key CHAR(12)
AS
BEGIN
   --  Rep returns the rep list for customer @cust_key
   DECLARE Rep CURSOR FOR
      SELECT DISTINCT RTRIM( rep_key )
      FROM Policy
      WHERE cust_key = @cust_key
   DECLARE @rep_key    CHAR(12)
   DECLARE @subscription_list   VARCHAR(255)
   -- build comma-separated list of rep_key
   -- values for this Customer
   OPEN Rep
   FETCH Rep INTO @rep_key
   IF @@sqlstatus = 0 BEGIN
      SELECT @subscription_list = @rep_key
      WHILE 1=1 BEGIN
         FETCH Rep INTO @rep_key
         IF @@sqlstatus != 0 BREAK
         SELECT @subscription_list =
            @subscription_list + ',' + @rep_key
      END
   END
```

```
   ELSE BEGIN
      SELECT @subscription_list = ''
   END

   -- update the subscription_list and last_modified
   -- columns in the Customer table
   UPDATE Customer
   SET subscription_list = @subscription_list,
       last_modified = getdate()
   WHERE cust_key = @cust_key
END
go
```

Again, most of the code in this section has been discussed previously, but there are a few things that warrant further discussion.

The customer_del table has a composite primary key that includes the cust_key, rep_key, and del_time columns. Deletes on the same row in the Customer table could be sent to two different remotes, so both the cust_key and rep_key columns are needed. A sales rep could also lose their last customer, gain the customer back as a result of a new policy, and then lose the customer again. To solve the above, the del_time column is also needed in the primary key.

There is no delete trigger on the Customer table to add rows into the Customer_del table. A delete on the Customer table will fail with an RI violation as long as rows exist in the Policy table which reference the customer you are trying to delete. A row in the Customer table can only be deleted if there are no rows in the Policy table that reference the row in the Customer table. When the last row in the Policy is deleted that removes the relationship between a sales rep and a customer, the delete trigger on the Policy table will add a row to the Customer_del table. The ability to delete the row in the customer table at the consolidated database means that the row as already been deleted at all the remote sites.

No changes were made to the insert trigger on the Policy table, but a minor change was made to the SubscribeCustomer stored procedure called in the insert and delete triggers on the Policy table. The last_modified column in the customer table is changed at the same time as the subscription_list, so that the addition of a new rep_key in the subscription_list would automatically be picked up by the download_cursor.

The delete trigger on the Policy table was changed and another cursor was added that looped through the rows in the deleted table. For each row deleted in the Policy table, check if a delete on the Customer table is needed for the sales rep that just lost a Policy for a given Customer. It's possible that this sales rep has many policies for this customer. Deleting one of these policies may not result in a delete for the customer. A delete is only sent down if the last policy that associates a sales rep with a customer is deleted.

The upload_update cursor on the Policy table has a `WHERE 1=0` clause to prevent it from updating any rows. We cannot simply delete the event since MobiLink will complain about a potential data loss since no script exists. In our simple example, updates are not allowed on this table, so any update that may come from the remote (which should also have not have been allowed) will be ignored. If your application has additional columns in the relationship table that are updateable,

but changes to the rep_key and cust_key column are still not allowed, it would be advisable to ensure your upload scripts follow these rules as well. You could write a stored procedure that takes the rep_key and cust_key values as parameters, but the stored procedure itself would not update those values on the consolidated. In theory, if your application at the remote does not allow these updates to occur, the values would never change, but it never hurts to be extra cautious and prevent something bad from happening that should be impossible anyway.

3.3.8 Conflict Resolution Scripts

To define conflict resolution scripts for SQL Remote for ASE, when adding the table into the list of replicating tables, you would also specify a resolve_procedure, an old_row table, and a remote_row table. The old_row table will store the values in the consolidated database before the update takes place, the remote_row table will store the previous values from the remote database, and the table itself will have already been updated with the new values from the remote. Here's a simple example that uses the t1 table from previous examples. This simple example uses the c3 column to determine which row wins the conflict, with the highest value in the column determining the winner of the conflict.

```
CREATE TABLE t1 (
  pkey INTEGER PRIMARY KEY,
  c1 INTEGER,
  c2 VARCHAR(100),
  c3 NUMERIC(10,2),
  last_modified DATETIME DEFAULT getdate()
)
go
CREATE TABLE t1_old (
  pkey INTEGER PRIMARY KEY,
  c1 INTEGER,
  c2 VARCHAR(100),
  c3 NUMERIC(10,2),
  last_modified DATETIME
)
go
CREATE TABLE t1_remote (
  pkey INTEGER PRIMARY KEY,
  c1 INTEGER,
  c2 VARCHAR(100),
  c3 NUMERIC(10,2),
  last_modified DATETIME
)
go
```

```
CREATE PROCEDURE sp_resolve_t1
AS
BEGIN
  DECLARE @lost_c3 NUMERIC(10,2)
  DECLARE @won_c3 NUMERIC(10,2)
  DECLARE @rem_c3 NUMERIC(10,2)
  DECLARE @lost_c2 VARCHAR(100)
  DECLARE @won_c2 VARCHAR(100)
  DECLARE @rem_c2 VARCHAR(100)
  DECLARE @lost_c1 INTEGER
  DECLARE @won_c1 INTEGER
  DECLARE @rem_c1 INTEGER
  DECLARE @pkey INTEGER

  SELECT @lost_c1=c1, @lost_c2=c2, @lost_c3=c3, @pkey=pkey FROM
         t1_old
  SELECT @won_c1=c1, @won_c2=c2, @won_c3=c3 FROM t1 WHERE
         pkey=@pkey
  SELECT @rem_c1=c1, @rem_c2=c2, @rem_c3=c3 FROM t1_remote

  IF @lost_c3 > @rem_c3
    BEGIN
      IF @lost_c3 > @won_c3
        BEGIN
          UPDATE t1 SET c1=@lost_c1, c2=@lost_c2, c3=@lost_c3
           WHERE pkey=@pkey
        END
    END
  ELSE
    BEGIN
      IF @rem_c3 > @won_c3
        BEGIN
          UPDATE t1 SET c1=@rem_c1, c2=@rem_c2, c3=@rem_c3
           WHERE pkey=@pkey
        END
    END
END
go

exec sp_create_publication p1
exec sp_add_remote_table 't1', 'sp_resolve_t1', 't1_old', 't1_
         remote'
exec sp_add_article 'p1', 't1'
exec sp_add_article_col 'p1', 't1', 'pkey'
exec sp_add_article_col 'p1', 't1', 'c1'
exec sp_add_article_col 'p1', 't1', 'c2'
exec sp_add_article_col 'p1', 't1', 'c3'
go
commit work
go
```

The conflict resolution process performed by MobiLink is very similar to the conflict resolution process used by SQL Remote for ASE. With MobiLink, you define an upload_fetch cursor that compares the current state of the row about to be updated with the pre-image of the row stored in the upload stream. If these two values don't match, then a conflict is about to occur and three additional events fire (if they are defined).

- **upload_new_row_insert** This event is passed the new row values from the remote database.
- **upload_old_row_insert** This event is passed the old row values from the

remote database.

- **resolve_conflict** This event is called for each row that causes a conflict.

The only real difference between the two modes of conflict resolution is that with Mobilink conflict resolution, the update has not taken place yet, but in each process, you have access to the same three rows to determine how to resolve the conflict. Most of the hard work has already been done, as the logic to determine which row wins is already written. In a perfect world, we would re-use the same tables and stored procedure, but SQL Remote for ASE has checks in it to ensure that only one thread is ever doing conflict resolution at the same time, so if MobiLink were to do conflict resolution at the same time as SQL Remote for ASE, we would run into problems. The code to implement MobiLink conflict resolution is included below. Although new tables were created and a new stored procedure was used, all the logic is the same, and the work is mostly cut-and-paste work.

```
CREATE PROCEDURE sp_resolve_ml_t1
AS
BEGIN
  DECLARE @cur_c3 NUMERIC(10,2)
  DECLARE @new_c3 NUMERIC(10,2)
  DECLARE @old_c3 NUMERIC(10,2)
  DECLARE @cur_c2 VARCHAR(100)
  DECLARE @new_c2 VARCHAR(100)
  DECLARE @old_c2 VARCHAR(100)
  DECLARE @cur_c1 INTEGER
  DECLARE @new_c1 INTEGER
  DECLARE @old_c1 INTEGER
  DECLARE @pkey INTEGER

  SELECT @new_c1=c1, @new_c2=c2, @new_c3=c3, @pkey=pkey FROM
         #t1_ml_new
  SELECT @old_c1=c1, @old_c2=c2, @old_c3=c3 FROM #t1_ml_old
    WHERE pkey=@pkey
  SELECT @cur_c1=c1, @cur_c2=c2, @cur_c3=c3 FROM t1 WHERE
         pkey=@pkey

  IF @new_c3 > @old_c3
    BEGIN
      IF @new_c3 > @cur_c3
        BEGIN
          UPDATE t1 SET c1=@new_c1, c2=@new_c2, c3=@new_c3
           WHERE pkey=@pkey
        END
    END
  ELSE
    BEGIN
      IF @old_c3 > @cur_c3
        BEGIN
          UPDATE t1 SET c1=@old_c1, c2=@old_c2, c3=@old_c3
           WHERE pkey=@pkey
        END
    END

  DELETE FROM #t1_ml_new
  DELETE FROM #t1_ml_old
END
Go
```

```
exec ml_add_connection_script 'v1', 'begin_connection',
  'CREATE TABLE #t1_ml_new (
     pkey INTEGER PRIMARY KEY,
     c1 INTEGER,
     c2 VARCHAR(100),
     c3 NUMERIC(10,2)
   )
   CREATE TABLE #t1_ml_old (
     pkey INTEGER PRIMARY KEY,
     c1 INTEGER,
     c2 VARCHAR(100),
     c3 NUMERIC(10,2)
   )'
go
exec ml_add_table_script 'v1', 't1', 'upload_fetch',
  'select pkey,c1,c2,c3 from t1 where pkey = ?'
go
exec ml_add_table_script 'v1', 't1', 'upload_new_row_insert',
  'insert into #t1_ml_new values ( ?, ?, ?, ? )'
go
exec ml_add_table_script 'v1', 't1', 'upload_old_row_insert',
  'insert into #t1_ml_old values ( ?, ?, ?, ? )'
go
exec ml_add_table_script 'v1', 't1', 'resolve_conflict',
  'exec sp_resolve_ml_t1'
go
```

3.4 First Synchronization From Migrated Remote

When a remote first synchronizes, the LastDownload timestamp is '1900-01-01 00:00:00'. If an empty remote database is deployed, the first time it synchronizes all data changed since Jan 1, 1900 is downloaded to the remote. This is an effective way to populate a remote database fully. In this case, we have an already fully populated remote database so we do not want MobiLink to download all the data the first time we run dbmlsync.

When we discussed migrating the remote databases, we glossed over some important details about what was done on the consolidated database during the first synchronization to ensure that the next download that came from MobiLink did not include all the data that had previously been downloaded through SQL Remote. When a MobiLink remote database synchronizes for the first time, the last_download value that is passed up is a value of '1900-01-01 00:00:00'. We can use this fact to recognize that this is a first synchronization and to take some additional steps during the synchronization.

All the extra steps will be taken in the modify_last_download_timestamp event. First, we'll check and see if the last_download values is '1900-01-01 00:00:00'. If it is, then check and see if the ml_username passed up is also a user name that exists in the sr_remoteuser table. If yes, retrieve the value of the time_sent column from the sr_remoteuser table for this user. This represents the time the last SQL Remote message was sent to this user. Modify the last_download value passed in ('1900-01-01 00:00:00') to the time from the sr_remoteuser table, and

then remove the SQL Remote user from the sr_remoteuser table so no more SQL Remote messages are sent to this user. Here's what the code might look like:

```
CREATE PROCEDURE sp_mldt
  @ldts      DATETIME OUTPUT,
  @ml_user   VARCHAR(128)
AS
BEGIN
  IF @ldts = '1900-01-01 00:00:00'
  BEGIN
    IF EXISTS ( SELECT 1 FROM sr_remoteuser
                WHERE user_name(user_id) = @ml_user )
    BEGIN
      SELECT @ldts=time_sent
        FROM sr_remoteuser
       WHERE user_id = user_id(@ml_user)
DELETE FROM sr_subscription WHERE user_id = user_id( @ml_user )
DELETE FROM sr_remoteoption WHERE user_id = user_id( @ml_user )
      DELETE FROM sr_remoteuser WHERE user_id = user_id( @ml_
         user )
    END  -- if exist
  END  -- if @ldts
END  -- end proc
exec ml_add_connection_script 'v1', 'modify_last_download_
         timestamp',
  '{call sp_mldt ?, ?}'
go
```

Given that new databases you deploy to remote sites will now be synchronized using MobiLink and not SQL Remote, it's also important to make sure that a new remote database for a MobiLink user will be able to synchronize successfully. The big difference between a new MobiLink remote database and a SQL Remote database is that there is no data in a new MobiLink remote database. All the data will synchronized because of the last_download timestamp of '1900-01-01 00:00:00' that is passed up. We can distinguish between a recently migrated SQL Remote database and a new MobiLink database based on whether or not a row exists in the sr_remoteuser table. When a row doesn't exist, you don't modify the last_download timestamp value in the modify_last_download_timestamp event, and the value of '1900-01-01 00:00:00' is used in the rest of the synchronization.

Because the initial values of the new last_modified columns were set to '1900-01-01 00:01:00' for all existing tables, all rows will be downloaded to the new MobiLink client when they first synchronize. The reason that '1900-01-01 00:01:00' was chosen was because it was greater than '1900-01-01 00:00:00', but guaranteed to be less than the minimum time_sent value in the sr_remoteuser table.

4.0 Conclusion

No migration from one product to another is as easy as just installing the new product. You will need to configure the new product, and the old product may need to be configured to ensure that the changes made to configure the new product do not cause problems. This is very much true when migrating from SQL

Remote for ASE to MobiLink, but the benefits far out weight the disadvantages. Hopefully, this whitepaper has shown the steps that need to be taken to migrate your SQL Remote for ASE environment successfully to one that uses MobiLink, all the while considering the migration goals we defined in Section 1.3. Although there is work to be done by the administrators at the consolidated database, there is very little impact at the remote sites during this migration.

If you require further assistance, there are many options available to you. With years of experience, iAnywhere Professional Services can deliver tailored services to ensure a successful migration. Examples of services provided by iAnywhere Professional Services include:

- Analyzing and reviewing replication rules
- Designing and architecting a MobiLink solution equivalent to or enhancing your current SQL Remote for ASE solution
- Developing and testing MobiLink scripts
- Providing customized knowledge transfer
- Delivering MobiLink Synchronization training

For more information, contact your local iAnywhere Solutions sales representative, e-mail *iAnywhere_Services@ianywhere.com*, visit *http://www.ianywhere.com/services*, or call 1-800-801-2069.

Sybase and iAnywhere Solutions also provides free forums that are monitored by our customers, members of Technical Support, and iAnywhere Solutions developers. For more information on accessing the newsgroups, please visit *http://www.ianywhere.com/support/newsgroups.html*. You can also visit the iAnywhere Solutions Developer Community at *http://www.ianywhere.com/developer*. The Developer Community contains a wealth of information including the latest iAnywhere Solution software patches, code samples, technical whitepapers such as this one, and much more.

For a specific problem you are having, you can always open a call with iAnywhere Solutions technical support. Within North America, you can reach Technical Support by calling 1-800-8SYBASE. If you are located outside of North America, you can find the contact information for your local Sybase Support Center by going to the Sybase home page at *http://www.sybase.com*, clicking the Support link at the top of the page, and then choosing Support Centers from the left frame. You can always report a bug for free by visiting CaseXpress at *http://casexpress.sybase.com/cx/cx.stm*. When submitting calls through CaseXpress, please make sure to give an accurate description of the steps you take to reproduce the bug, and include any files that may be needed (database, source code) to reproduce the problem.

For product or sales inquiries you can contact iAnywhere Solutions by calling 1-800-801-2069 within North America. Alternatively, you can contact us by email at *contact.us@ianywhere.com*.

Legal Notice

Copyright © 2005 iAnywhere Solutions, Inc. All rights reserved. Sybase, the Sybase logo, iAnywhere Solutions, the iAnywhere Solutions logo, Adaptive Server, MobiLink, and SQL Anywhere are trademarks of Sybase, Inc. or its subsidiaries. All other trademarks are property of their respective owners.

The information, advice, recommendations, software, documentation, data, services, logos, trademarks, artwork, text, pictures, and other materials (collectively, "Materials") contained in this document are owned by Sybase, Inc. and/or its suppliers and are protected by copyright and trademark laws and international treaties. Any such Materials may also be the subject of other intellectual property rights of Sybase and/or its suppliers all of which rights are reserved by Sybase and its suppliers.

Nothing in the Materials shall be construed as conferring any license in any Sybase intellectual property or modifying any existing license agreement.

The Materials are provided "AS IS", without warranties of any kind. SYBASE EXPRESSLY DISCLAIMS ALL REPRESENTATIONS AND WARRANTIES RELATING TO THE MATERIALS, INCLUDING WITHOUT LIMITATION, ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT. Sybase makes no warranty, representation, or guaranty as to the content, sequence, accuracy, timeliness, or completeness of the Materials or that the Materials may be relied upon for any reason.

Sybase makes no warranty, representation or guaranty that the Materials will be uninterrupted or error free or that any defects can be corrected. For purposes of this section, 'Sybase' shall include Sybase, Inc., and its divisions, subsidiaries, successors, parent companies, and their employees, partners, principals, agents and representatives, and any third-party providers or sources of Materials.

Contact Us

iAnywhere Solutions Worldwide Headquarters One Sybase Drive, Dublin, CA, 94568 USA

Phone 1-800-801-2069 (in US and Canada)

Fax 1-519-747-4971

World Wide Web *http://www.ianywhere.com*

E-mail contact.us@ianywhere.com