MobiLink Performance

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

Executive summary 2

Introduction 3

What are the time-consuming steps in MobiLink synchronization? 3
  Initiation ............................................... 4
  Connections ........................................... 4
  Upload .................................................. 5
  Download ............................................. 7
  Scaling up to more clients ............................. 7
  Potential bottlenecks ................................. 8

Performance testing 9
  Database schema ......................................... 10
  Values ................................................. 10
  Timing framework ....................................... 11
  Testing environment .................................... 13
  Tests performed ....................................... 13

Recommendations 27
  Performance tips ........................................ 28
  Large deployments ...................................... 28
  Applicability .......................................... 29

Legal Notice 31
  Contact Us ............................................. 31
Executive summary

MobiLink provides scalable, high-performance data synchronization between remote client databases and a central database (referred to as the consolidated database). Key factors in MobiLink synchronization performance are: the performance of the consolidated database for your synchronization scripts, the remote client processing speed and network bandwidth, the speed of the computer running MobiLink, and the number of MobiLink worker threads (each with a connection to the consolidated database).

The performance of MobiLink synchronization can be summarized as follows:

♦ MobiLink has modest hardware requirements, and makes good use of multiple processors. For example, in our tests with an Adaptive Server Anywhere consolidated database, MobiLink requires less than half the processing required by the consolidated database.

♦ MobiLink scales very well with additional remote clients.

♦ Since downloads involve significant processing on the remote client, and little processing by MobiLink, download throughput is much more dependent on client speed than uploads.

♦ You must test to determine the optimal number of MobiLink worker threads, and use the smallest number that gives optimal throughput. For example, in our tests with a fully loaded server running the MobiLink synchronization server and consolidated database, the best throughput was obtained with nine MobiLink worker threads. When remote processing or networking is the bottleneck, optimal performance may be obtained with a larger number of worker threads.

♦ There is some per-synchronization overhead, so throughput rates are lower for smaller-sized synchronizations.

♦ MobiLink does not maintain state information between connections, so you can use multiple MobiLink synchronization servers. For example, for high availability and scalability you could use multiple MobiLink synchronization servers with a third-party load-balancing system to make them appear as a single server.

As an example of performance, here are the results for 10000 clients synchronizing 1000 92-byte rows each (for a total of 10 million rows being synchronized) with MobiLink and an Adaptive Server Anywhere consolidated database running on a Dell PowerEdge server with four Pentium4Xeon 2.2GHz processors:
Introduction

iAnywhere Solutions developed MobiLink synchronization technology to enable efficient and scalable data synchronization between many remote databases and a single central database (referred to as the consolidated database). MobiLink is included with SQL Anywhere Studio. It enables data synchronization between remote UltraLite or Adaptive Server Anywhere databases, and consolidated databases with ODBC access, including Adaptive Server Anywhere, Sybase Adaptive Server Enterprise, Microsoft SQLServer, Oracle, and IBM DB2.

The goal of this paper is to give you an understanding of MobiLink performance and to help you assess its applicability to your data synchronization needs. First, this paper describes the time-consuming steps in MobiLink synchronization, and presents results of performance testing of MobiLink synchronization technology, under different conditions.

After a brief description of the testing methodology, this paper presents performance results and analysis for varying the following conditions:

- the number of MobiLink worker threads
- the network backlog threshold
- the retry delay when synchronizing with large numbers of clients
- the number of remote databases synchronizing simultaneously
- the amount of data transferred in each synchronization

We then give recommendations on how to get the best performance from MobiLink.

What are the time-consuming steps in MobiLink synchronization?

To help you understand MobiLink performance, we first look at the steps in a single MobiLink synchronization, namely initiation, connections, upload, and download, with an eye to how they contribute to the time taken for a
synchronization. After that, we look at scaling to more clients and identifying potential bottlenecks.

Initiation

A MobiLink synchronization is typically initiated by the remote client (although server-initiated synchronization is an option, it is not considered here). Depending on whether the client is an UltraLite database or Adaptive Server Anywhere database, the order of events is slightly different. An UltraLite client establishes the connection with MobiLink first and then builds the upload stream as it sends it, whereas an Adaptive Server Anywhere client builds the upload stream before establishing a connection. (The difference in constructing the upload stream allows UltraLite clients to work with very limited memory.)

Connections

A MobiLink synchronization has two types of connection: a synchronization connection between the client and MobiLink, and a database connection between MobiLink and the consolidated database.

The client makes a synchronization connection with the MobiLink synchronization server. The client waits while MobiLink assigns a worker thread from the pool of worker threads. This involves a small amount of data transfer between the client and MobiLink, and the overhead of establishing a connection (such as the operating system cost for creating a TCP/IP socket if a TCP/IP stream is being used). This overhead should be small, but from a client’s point of view, much time may be spent waiting in a queue for a MobiLink worker thread to become available if all of the worker threads are busy with other clients.

MobiLink itself is not limited in the number of clients that can try to synchronize simultaneously, but the operating system imposes a limit. MobiLink queues clients waiting for worker threads, and it uses little memory per waiting client (less than 32 bytes), so its data structures allow for billions of queued clients. However, the operating system imposes a limit on the number of active or pending sockets and the amount of data that the operating system can buffer for the sockets. By default, MobiLink accepts all incoming network connections, causing the operating system to buffer the uploads of clients waiting for a worker thread. For a large number of queued synchronizations, this can exhaust the operating system network buffer memory, leading to network errors. The MobiLink backlog stream parameter can be used to ensure that synchronization requests are rejected before the operating system limit is reached. When the MobiLink backlog parameter is specified, MobiLink only queues that many requests, limiting the amount of upload data that the operating system needs to buffer. Additional connection requests are queued by the operating system, up to its limit of pending network connections, without transferring data. Additional connection requests are refused by the operating system. A pending connection queued by the operating system is accepted by the MobiLink synchronization server once a place in the backlog queue is available, or it may fail with a network timeout error from the client or server operating system. Clients with rejected or timed-out synchronizations must retry later. You can also use the backlog parameter if you
prefer to have synchronizations rejected instead of queued when the MobiLink synchronization server is busy, for example to avoid online charges for queued synchronization requests.

MobiLink maintains a pool of connections to the consolidated database, as well as one connection for accessing the MobiLink system tables. At the beginning of synchronization, a worker thread briefly uses the latter connection for client synchronization information and for fetching the synchronization scripts. The rest of the database connections are created as needed by the worker threads; one is dedicated to a worker thread for the duration of the synchronization, and then returned to the pool. All the synchronization data is sent over this connection. You can specify a larger number of database connections than the number of worker threads, but the most that will be used at once is one per worker thread, plus one for the MobiLink system tables.

There are two cases where MobiLink will close a database connection and open a new one. The first case is if an error occurred (however, you can disable this default behavior). The second case is if the client requests a synchronization version, and none of the available connections have already used that synchronization version. 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.

To help minimize the time taken to create new database connections, you can do the following:

♦ Use a small number of worker threads (-w option). You should not use more worker threads than an optimal number of simultaneous connections to your consolidated database. (Choosing an optimal number of worker threads for throughput is discussed in more detail later.)

♦ Set the maximum number of pooled MobiLink database connections to be your typical number of synchronization versions times the number of MobiLink worker threads (-cn option).

Upload

Once a client is connected to a worker thread, the client transfers its upload stream to the worker thread. The bandwidth of the client-to-MobiLink connection and the size of the upload stream are the main influences on the time taken to transfer the upload stream to the worker thread.

For an UltraLite client, the upload phase also includes some client processing time to determine what data it needs to upload (data that is new or has changed since the last synchronization). This is normally a small amount of processing; it is much less than the client processing required in the download phase.

For an Adaptive Server Anywhere client, the upload stream is created before the client connects to MobiLink, so the time taken to create the upload stream occurs before the connection to MobiLink.
The transfer of the upload stream should occur entirely in memory for MobiLink. The MobiLink worker threads share a common upload cache in memory; if the cumulative size of the upload streams for the worker threads is larger than the upload cache, then the overflow is written to disk. An additional cache is shared for BLOB columns, including large character columns, for both upload and download. This cache also overflows to disk. To ensure that MobiLink runs entirely in memory, and avoids any bottlenecks associated with disk access, you need to do the following:

- Use an upload cache that is larger than the size of your largest upload stream times the number worker threads (-u option).
- Use a BLOB cache that is larger than twice the largest BLOB data in a row times the number of worker threads if you are using conflict detection (-bc option).
- Ensure that the computer running MobiLink has enough physical memory to accommodate the upload and BLOB caches in addition to its other memory requirements.
- Do not use verbose logging. Without verbose logging, MobiLink only logs startup information, warnings, and errors, so there is no ongoing disk access when you have MobiLink write its log information to a file.

Once the MobiLink worker thread has the entire upload stream, it processes the stream on a row by row basis. For a row, it first performs character set conversion to Unicode of any text strings. Exceptions are WindowsCE clients, where strings are already in Unicode, and for non-Windows versions of MobiLink. A reverse conversion occurs for WindowsCE clients connected to MobiLink running on a non-Windows platform. This takes little time, but the time is proportional to the amount of character data.

The MobiLink worker thread then uses its database connection to apply the uploaded row to the consolidated database, using the upload scripts that you have defined. For this, MobiLink worker threads have the same performance issues as any other clients transacting with the consolidated database, including simultaneous connections, size of transactions, concurrency issues, and, if the consolidated database is on a different computer than MobiLink, the network bandwidth. In addition, if your upload scripts incorporate conflict detection, MobiLink fetches each row to be updated from the consolidated database before applying the update, to check for a conflict. Additional processing is also required for any detected conflicts. Thus, the synchronization of an update with conflict detection takes significantly longer than an update without conflict detection, typically more than double the time without conflict detection.

Once the upload transaction is complete in the consolidated database, it is committed and the MobiLink worker thread sends an implicit acknowledgment to the client. At this point, the client starts resetting its record of which data has been modified, so that the previously uploaded changes will not be uploaded again in the next synchronization. At the same time, the MobiLink worker thread starts preparing for the download phase.
Download

To prepare for the download, the MobiLink worker thread determines the list of rows to be inserted, updated, or deleted on the MobiLink client by executing your download scripts in the consolidated database. The rows are then fetched from the database and, as with uploads, MobiLink worker threads have the same performance issues as any other clients transacting with the consolidated database. Fetched BLOB values are held in the same BLOB cache that is used for uploads. For character data, the character set conversion is performed opposite to the upload. It sends these rows back to the MobiLink client in the form of a download stream. The process of fetch, process, and transfer occurs on a row by row basis, so the transfer to the client starts after the first row is fetched from the consolidated database. The steps are interleaved until all the rows have been downloaded. Typically, the download phase involves significantly less processing, for both the MobiLink worker thread and the consolidated database, than for the upload phase.

In contrast, the client does more processing in the download phase than in the upload phase. As each part of the download stream is received, old rows are deleted, new rows are inserted, updated rows are changed, indexes are updated, and referential integrity is checked. This processing may take a significant amount of time with remote clients that have slow processors. For UltraLite clients, a larger database will, in general, require more client processing time to insert new rows. The download bandwidth to the client also affects the time required for this step.

Once the client has received, processed, and committed the download stream, it sends a final acknowledgment to the MobiLink worker thread. Upon receipt of the acknowledgment, the MobiLink worker thread commits the download transaction in the consolidated database. The worker thread is then available for another synchronization.

Since downloads involve significant processing on the remote client, and little processing by MobiLink, download time is more dependent on the client's processing speed than upload time is. The MobiLink worker thread must wait until the client acknowledges that it has finished processing the download stream before it can commit the download transaction.

The bandwidth of the client-to-MobiLink connection, the size of the download stream, and the client processing speed are the main influences on the time taken to transfer the download stream to the client.

Scaling up to more clients

Now that we have examined the time-consuming steps in a single synchronization, let us consider the case of multiple clients synchronizing simultaneously.

As stated previously, if the number of clients trying to synchronize is greater than the number of MobiLink worker threads, then the excess clients are queued up waiting for an available worker thread or are refused via the backlog parameter.
Thus, more worker threads allows more simultaneously-active synchronizations. If more clients attempt to synchronize than can be handled by the operating system, then the backlog parameter ensures that the excess clients are rejected, and they should attempt to synchronize again.

If there are at least as many clients as worker threads, but not enough to overwhelm the operating system, then we might expect the total synchronization time to be approximately the sum of the times for individual synchronizations (excluding time waiting for a worker thread) divided by the number of worker threads:

\[
\text{total time} = \frac{\sum_{i=1}^{\text{# clients}} \text{time for sync } i}{\text{# workers}}
\]

You can also express this value as follows:

\[
\text{total time} = \frac{\text{# clients} \times \text{average sync time}}{\text{# workers}}
\]

This is an idealized formula; it neglects contention between the worker threads (both within MobiLink and in their transactions with the consolidated database), multitasking overhead from the operating system, and assumes all worker threads always have available CPU time. It should only apply while the MobiLink synchronization server and consolidated database have unused processing and disk access capacity. If there is no additional capacity, then increasing the number of worker threads (and thus database connections) decreases performance.

Potential bottlenecks

The overall performance of any system, including throughput for MobiLink synchronization, is usually limited by a bottleneck at one or more points 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 for the chosen number of simultaneously-active connections (the number of MobiLink worker threads plus any other active connections). For best throughput, you need to be careful to avoid database contention in your synchronization scripts.

♦ The bandwidth for communication between MobiLink and the consolidated database. 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 processor speed and memory of the computer running MobiLink.

♦ The number of MobiLink worker threads (and hence the number of simultaneously-active connections to the consolidated database). While a
smaller number of threads will incur fewer database connections, less chance of contention in the consolidated database, and less operating system overhead, too small a number may leave clients waiting for a free worker thread, or underutilize the consolidated database.

♦ The bandwidth for client-to-MobiLink communications. For slow connections, such as those over wide-area wireless networks, the network may cause clients and MobiLink worker threads to wait for data to be transferred.

♦ The client processing speed. This is more likely to be a bottleneck in downloads than uploads, since downloads involve more client processing.

The potential bottlenecks that are not specific to MobiLink are beyond the scope of this whitepaper. For example, we will not consider the effect of network bandwidth or general client-server performance of the consolidated database.

Performance testing

The goal of the tests for this whitepaper was to measure the performance characteristics of MobiLink, so that you can gain insight into how MobiLink will perform in your situation. In particular, we sought to determine the following:

♦ The performance of MobiLink with a large number clients synchronizing simultaneously.

♦ The optimal number of MobiLink worker threads to maximize throughput.

♦ Whether or not the backlog parameter is required for up to 10000 simultaneous synchronization requests.

♦ The optimal retry delay when synchronization requests in excess of the backlog are being rejected.

♦ The effect of differing the number of clients synchronizing simultaneously.

♦ The effect of differing the amount of data transferred in each synchronization.

♦ The hardware requirements for a MobiLink synchronization server, relative to those for the consolidated database.

To make the results easier to understand, our testing methodology employed the following principles:

♦ Vary only one parameter at a time. Since there are many variables that affect MobiLink synchronization throughput, we took care to vary only one variable at a time as much as possible. Often we would vary one parameter until optimal throughput was achieved, fix that parameter, and then vary another one.

♦ Stress MobiLink and/or the consolidated database. Since we are concentrating on MobiLink performance, we wanted to make sure it was being stressed. In general, we ran MobiLink and the consolidated database on the same server computer, with clients on separate computers, with enough simultaneous synchronizations to keep the server CPU utilization at 100%. MobiLink and the
consolidated database were run on the same computer so that the relative CPU utilization of each could be used to infer their relative hardware requirements.

Keep it simple. In order to concentrate on the inherent performance of MobiLink, the synchronization scripts, the client application, and the database schema were all kept very simple. More complex schema and synchronization scripts would increase the load on the consolidated database, so by keeping them simple we maximized the relative workload of the MobiLink synchronization server.

The following sections describe specific aspects of the performance tests.

Database schema

All of the tests are based on synchronizing the data in a single table. The column types were chosen so that several popular data types would be used. The following SQL statement shows the table definition:

```sql
CREATE TABLE Purchase (
    emp_id INT NOT NULL,
    purch_id INT NOT NULL,
    cust_id INT NOT NULL,
    cost NUMERIC(30,6) NOT NULL,
    order_date TIMESTAMP NOT NULL,
    notes VARCHAR(64),
    PRIMARY KEY (emp_id, purch_id)
)
```

A two-column primary key was used to simplify partitioning the data among the remote clients; the first column indicates which remote client the row is associated with. To avoid the complication of contention in the consolidated database, there is no sharing of data between clients. In a real-world scenario, it is likely that users would share data, which could result in conflicts if multiple users changed the same row before synchronizing. These conflicts would distort timing information, so data sharing was prevented in these tests.

Values

The data values that are used to fill the Purchase table are generated in either the client or consolidated database. The algorithms to generate the values ensure that large values are used for integer data, so that the packing scheme did not shrink the data when it was transferred to or from the client. The initial value for cost was 123456789.12 for all rows, and each update increments the cost value by one. All values used for the string column are exactly sixty-four characters long. This was done to ensure that a constant amount of bytes would be transferred for each row; in this case, each row transferred happened to be 92 bytes. In a real-world scenario, each row would likely not be exactly the same size, but this simplification was necessary for accuracy in our tests.
Timing framework

To run the performance tests, we developed a framework for timing MobiLink synchronizations. The components of the framework are the following:

♦ **Extra tables in the consolidated database** There is one table to hold information on the tests to be performed, such as the number of runs, number of clients per run, and what every client should do. There is another table to hold the timing information, which holds four times for each synchronization: the start and end time as recorded on both the client and the consolidated database. Another table is used to keep track of the next data to download to each client. There are also some temporary tables and views related to the run information and timings.

♦ **The MobiLink synchronization scripts** Several synchronization versions, or sets of scripts, are employed. Two versions control the setup: two are for timed synchronizations (with or without conflict detection when updates are uploaded), and one is for the clients to send their timings to the consolidated database at the end of the run. For the timed synchronizations, the following scripts are defined:
  - `begin_synchronization` records the server start time
  - `end_synchronization` records the server end time
  - `download_cursor` controls the number of rows to be downloaded
  - `upload_cursor` defines where the upload will be applied
  - `old_row_cursor` for conflict detection when synchronizing updates
  - `new_row_cursor` for conflict detection when synchronizing updates
  - `resolve_conflict` called if a conflict is detected (never used in our tests)

♦ **The client application** Since we want to stress MobiLink with a lot of simultaneously-synchronizing clients, we need a small, efficient client program that can have multiple instances of itself run on one or more computers. So that each instance of the client would have a small memory footprint, we chose to use UltraLite clients. For efficiency, we chose to implement the client in C, using the embedded SQL interface to UltraLite. To avoid slowing the clients with disk access, we used a special version of the UltraLite runtime library that did not use file-based persistent storage for the UltraLite databases. For ease of multiprocessing, we used a Windows console program that can spawn multiple instances of itself as a client. The first instance is used as a master process, and it spawns child processes that act as clients. The client application also has an option to retry, after a specified delay, any synchronizations that fail from communication errors.

♦ **A supervisor application** This coordinates clients running on different computers.

All clients are kept in step through the use of gates. At a gate, each client waits for all the others to reach the same point before proceeding, like the starting gate at a race. For efficiency, the gate implementation uses operating system...
primitives (Win32 event objects are used to wait for clients on the same computer, and waiting across computers uses Win32 Named Pipes to the supervisor). The gate implementation is very efficient; for a thousand clients on ten or fifteen computers, all clients start up after a gate within a second or two of each other.

The clients use gates before and after each timed synchronization. The gate before the synchronization ensures that all clients try to start synchronizing at the same time. The gate after synchronization ensures that no other client processing will occur until all client synchronizations are completed.

For synchronizations that are timed, the following steps occur:

<table>
<thead>
<tr>
<th>Step</th>
<th>Client</th>
<th>MobiLink</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Prepare for synchronization.</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Wait for all other clients (via gate)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Record client start time</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Start synchronization, via ULSynchronize()</td>
<td>Record server start time (in begin_synchronization script)</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>Perform synchronization</td>
</tr>
<tr>
<td>6</td>
<td>Perform synchronization</td>
<td>Record server end time (in end_synchronization script)</td>
</tr>
<tr>
<td>7</td>
<td>Record client end time</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Record server start time (in begin_synchronization script)</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Wait for all other clients (via gate)</td>
<td></td>
</tr>
</tbody>
</table>

In the above table, there are extra steps associated with the timing. Steps 3, 5, 7, and 8 involve recording a timestamp, while steps 2 and 9 involve waiting for all other clients at a gate.

The client times are recorded in the memory of the clients, and then the times are sent to the consolidated database after all the timed synchronizations. The server times are recorded directly into the consolidated database.

In discussing the timing results, we refer to the following quantities derived from the recorded timestamp values:
If each client does more than one synchronization, these are grouped into sets of simultaneous synchronizations. All synchronizations in a set are initiated at the same time, and the next set will not start until after they have all completed. In this case we use the following formula to calculate the total server time:

\[
\text{client sync time} = \text{client end time} - \text{client start time}
\]

\[
\text{server sync time} = \text{server end time} - \text{server start time}
\]

\[
\text{total server time} = \text{last server end time} - \text{first server start time}
\]

\[
\text{throughput} = \frac{\text{total number of rows synchronized}}{\text{total server time}}
\]

In other words, even if there are multiple sets of simultaneous synchronizations, we only count the times spent between the gates.

**Testing environment**

The timing tests were performed with the following software and hardware:

The tested software was SQL Anywhere Studio 9.0.2. The consolidated database used the Adaptive Server Anywhere network database server, the synchronization server was MobiLink, and the clients (described above) used the UltraLite embedded SQL interface. Unless otherwise noted, default options were used for MobiLink and Adaptive Server Anywhere. The tests used the TCP/IP communication protocol, without encryption.

MobiLink and Adaptive Server Anywhere ran on a Dell PowerEdge 6650 server computer. This computer had four 2.2 GHz Pentium 4 Xeon processors with hyperthreading and 4 GB of RAM. It also had the software on one physical drive, the database log file on another physical drive, and the database file on a separate drive array. The operating system was WindowsServer 2003.

The clients were run on a rack of up to 12 Dell PowerEdge 750 computers, running WindowsServer 2003. Each computer had a 3.2 GHz Pentium 4 processor with hyperthreading and 1GB of RAM. These were networked to the server using a 1 Gbps option. The network was isolated for timing runs. For almost all tests, 10 PowerEdge 750 computers were used to run the clients.

**Tests performed**

In order to assess the performance and scalability characteristics of MobiLink, we used the timing framework to perform the following tests:
Vary the number of worker threads to determine the optimal number to maximize throughput. This was done using 1000 remote clients synchronizing simultaneously.

Determine the network backlog threshold for 10000 remote clients synchronizing simultaneously.

Vary the delay between client retry attempts when the number of clients exceeds the backlog threshold.

Vary the number of clients synchronizing simultaneously, to see if throughput drops off with more clients.

Vary the size of synchronizations, to see the effect on throughput.

In each case, four types of synchronization were measured:

- New data (insertions) downloaded from the central database.
- Updates uploaded from the remote databases, with conflict detection enabled.
- Deletions uploaded from the remote database.
- Insertions uploaded from the remote database.

All the timing runs follow the same steps. First, a new consolidated database is created and filled with enough data for the downloads. Then each client does one empty synchronization, to set up the client names in the MobiLink user table and to make sure that MobiLink has established all database connections with the right synchronization version. Then the timed synchronizations are performed, in the following order: downloads, updates, deletes, and then inserts. This exercises four types of synchronization.

Downloads of updates and downloads of deletes were not tested. Downloads of updates were not thought to be significantly different than the downloads presented here. Downloads of deletes would likely be faster, since only the primary key is sent to the remote client.

The order used (downloads, updates, deletes, inserts) ensures that the consolidated database will end up with the same number of rows as it started with, and the UltraLite databases will not have more rows than those downloaded to it.

The details of these tests, including results and analysis, are described in the following sections.

**Test 1: Optimal number of MobiLink worker threads**

In this test, we chose to use one thousand simultaneously synchronizing clients while varying the number of MobiLink worker threads (and hence the number of simultaneously active connections to the consolidated database). We chose this large number of clients to ensure that MobiLink was stressed and so that we might hit any point of diminishing throughput before running out of clients. The following were kept constant:

- 1000 clients on 10 P4-3.2 GHz computers
1000 rows per client synchronization (with 92 bytes per row)

10 sets of synchronizations for each client

Total of 10,000 synchronizations and 10 million rows

The results are shown in the following chart and table. The chart shows time versus the number of MobiLink worker threads for 1000 clients each synchronizing 1000 rows of 92 bytes 10 times, for a total of 10 million rows.

It is important to note that each point represents the total time for a 10 million row sync. With each number of worker threads, 10 million rows were downloaded, followed by 10 million rows that were updated, and so on. The time shown for each sync is the total time for all clients to finish their synchronization of that type. In other words, the time shown represents the time seen by the longest running client.

<table>
<thead>
<tr>
<th>Number of workers</th>
<th>Download</th>
<th>Update</th>
<th>Delete</th>
<th>Insert</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>332.488</td>
<td>2567.58</td>
<td>1645.99</td>
<td>1097.66</td>
</tr>
<tr>
<td>8</td>
<td>281.858</td>
<td>2037.27</td>
<td>1165.21</td>
<td>813.211</td>
</tr>
<tr>
<td>9</td>
<td>250.693</td>
<td>1953.27</td>
<td>1087.67</td>
<td>738.3</td>
</tr>
<tr>
<td>10</td>
<td>254.921</td>
<td>1951.61</td>
<td>1091.7</td>
<td>752.922</td>
</tr>
<tr>
<td>20</td>
<td>247.492</td>
<td>1971.04</td>
<td>1097.19</td>
<td>910.085</td>
</tr>
<tr>
<td>50</td>
<td>255.916</td>
<td>2095.13</td>
<td>1187.56</td>
<td>1067.46</td>
</tr>
</tbody>
</table>

The best times in this test were obtained for 9 worker threads. However, using between 9 and 20 worker threads did not significantly slow down the overall time for four synchronizations. When more than 20 worker threads were used, a
definite increase in synchronization time was noticed.

Looking at the download case, the minimum time for this test was achieved with 20 worker threads. This can be explained by experimental variability. The time taken for download synchronizations appears roughly constant in the range of 9–20 worker threads. In some cases involving slow clients, it may be advantageous to use more worker threads if synchronizations only consist of downloads.

This parameter is most affected by the capabilities of the consolidated database and the load placed on it by synchronization scripts.

There is no advantage to using 20 or more worker threads for 1000 clients that try to synchronize simultaneously. More worker threads results in decreased performance. The two most likely causes of decreased performance with increased worker threads are:

♦ Contention in the consolidated database. The chance of a connection being blocked by another connection increases with the number of worker threads, and blocked connections reduce performance. In this test, we chose to have each client access separate data so that database contention would not limit performance. In most applications this will not be possible, and contention in the consolidated database may be the performance bottleneck.

♦ Saturation of server processor or disk resources. If there is no further processing power available, adding worker threads will increase the operating system overhead for multitasking and increase hardware contention. In this test, the CPU utilization for all four processors was essentially 100% for 9 or more worker threads, so more worker threads would not have additional resources to use.

With 9 worker threads, only 0.9% of the 1000 simultaneous synchronizations will be active at once. The rest of the clients will either be queued waiting for a worker thread, or will have already finished synchronizing. From a client's perspective, the time to synchronize will be longer the closer the client is to the back of the queue, because of the increased time waiting. However the average client time will be minimized if the total time is minimized. This maximizes the throughput, and keeps the queue moving most quickly.

You may also note the relative time taken by the different types of synchronizations. The downloads are significantly faster than the uploads, and for uploads, inserts are fastest and updates slowest. In other tests with update conflict detection disabled, we found that updates synchronized at about the same rate as inserts. This is because conflict detection requires a fetch of each row to be updated from the consolidated database to see if its value has changed since the last synchronization.

We believe that the clients running on the 10 P4-3.2 GHz computers were sufficient to saturate MobiLink and Adaptive Server Anywhere. To check this, we repeated some tests using 12 and 8 computers for the clients, and the times were not significantly different.

Thus, it seems that when the clients are fast enough to keep the MobiLink worker threads busy, you should only use a few threads. In this case, 9 worker threads is
a good choice.

However, if your clients are not fast enough to keep each MobiLink worker thread busy, you may find better throughput with a larger number of worker threads.

When choosing a number of worker threads to use, remember the following points:

♦ The best throughput is achieved with a relatively small number of worker threads. If server CPU or disk is already saturated, or contention in the consolidated database is a bottleneck, then adding more worker threads will be counterproductive.

♦ Perform tests that reflect the type of synchronizations that will occur under your expected real-world conditions.

♦ Performance of the consolidated database, including the other loads placed on the database, will affect synchronization performance.

♦ For clients fast enough to saturate the server, using 9 worker threads gave the best throughput in our tests. This happens to be one more than the number of logical processors (since the four Xeon processors featured hyperthreading).

♦ For slower clients, you may want to use more worker threads (-w option), but limit the number that can simultaneously apply uploads to the consolidated database (-wu option) to reduce database contention.

Test 2: Determining the backlog parameter

As described in the Connections section, you may need to use the MobiLink backlog stream parameter to avoid operating system limitations when thousands of remote clients try to synchronize simultaneously.

The purpose of the backlog parameter is to limit the number of simultaneous connections queued by MobiLink. Since MobiLink only accepts a limited number of connections specified by the backlog parameter, the operating system limitations can be avoided.

When synchronization connection requests are refused because of the backlog stream parameter, the remote client application or user must retry the synchronization periodically until it succeeds. A MobiLink client does not automatically retry synchronizations.

For this test, 10000 clients attempted to synchronize 1000 rows each, initially without specifying a backlog or enabling client retries, and later with retries enabled and a backlog parameter. The backlog parameter prevented all 10000 clients from being accepted into MobiLink’s queue on their first synchronization attempt. The following was kept constant:

♦ 10000 clients on 10 P4-3.2 GHz computers

♦ 1000 rows per client synchronization (with 92 bytes per row)

♦ Total of 10000 synchronizations
9 MobiLink worker threads

In this test we found that the operating system could support a larger number of pending download synchronizations than upload synchronizations. For downloads, 5000 synchronizations succeeded before connections failed with communication errors. For uploads, the threshold was around 2000 synchronizations. In other tests with other operating systems, such as UNIX, Linux and Mac OS/X, we have found it necessary to specify smaller backlog values that correspond to the maximum number of file handles (which for some operating systems can be no larger than 1024). Note also that non-server versions of Windows impose a limit of 10 connections.

The difference between the number of clients that could synchronize successfully with downloads is greater than the number of clients that could synchronize uploads due to the fact that the operating system buffers the uploaded client data on uploads, but not on downloads. The uploaded data eats away at the available memory, allowing fewer client connections.

If you expect to handle thousands of remote clients synchronizing simultaneously, you will need to specify the backlog parameter for the MobiLink synchronization server and have your client applications retry synchronization after a delay when a synchronization request fails with a communication error.

In the rest of these tests, the backlog stream parameter was set to 1500 and retrying (on communication errors) was enabled for the remote client application. Using this backlog value with Windows 2003 Server, we saw no problems other than the expected refused connections in our tests.

Test 3: Optimal client retry delay

The next test varied the amount of time between retries for 10000 remote clients synchronizing 1000 rows each, with the MobiLink backlog set to 1500. The backlog parameter prevented all 10000 clients from being accepted into MobiLink’s queue on their first synchronization attempt. The following were kept constant:

- 10000 clients on 10 P4-3.2 GHz computers
- 1000 rows per client synchronization (with 92 bytes per row)
- Total of 10000 synchronizations
- 9 MobiLink worker threads

The results are specific to our tests, and should only be used to help understand the testing methodology used. The results are shown in the chart and table for synchronization time versus retry delay for 10000 clients, each synchronizing 1000 rows of 92 bytes:
The retry delay has an effect on the throughput. The minimum synchronization time depends on the type of synchronization. For example, the minimum synchronization time for uploading updates and deletes is reached when the retry delay is 10 seconds. For inserts, the minimum time is reached when the retry delay is 5 seconds. For downloads, the minimum time is reached when the delay is 1 second. This corresponds to the synchronization times for the different types; the faster synchronizations see the best throughput with shorter delays between retries. With too short of a delay, too much time will be spent by the server rejecting connection requests. With too long a delay, the queue of pending synchronizations may empty, leaving one or more idle MobiLink worker threads.

To further explain this behavior, it is useful to look at the number of times clients needed to retry their synchronizations for 10000 clients each synchronizing 1000 rows of 92 bytes:
From this data, you can see that with shorter retry delays, clients retry more often before successfully completing their synchronizations. Keep in mind that each entry in the table is the sum total of the number of retries for 10000 clients. With the backlog parameter set to 1500, at least the first 1500 clients are able to synchronize without any retries.

For all different values of the retry delay, downloads have the lowest number of retry attempts, followed by inserts. Since downloads and inserts are the fastest types of synchronization, the server is able to process more clients between retry attempts, thus freeing up more spaces in the queue. The additional free spaces allow more clients to successfully synchronize on their next retry attempts, thus resulting in a lower number of overall retry attempts.

To minimize the total amount of time taken by the synchronizations, it is necessary to balance the number of client retry attempts with the size of the queue. Having clients retry more frequently refills the queue more frequently, but failed connection attempts also cause additional load on the client computers, the network and the server. However, a retry delay that is too long will cause the MobiLink queue to run out of clients to process. This would result in lower
throughput. When the times for the four types of synchronization are summed, then the best total time occurred with a delay of 10 seconds. This value was used in subsequent tests, unless otherwise mentioned.

It is important to note that the time the clients take to synchronize determines the ideal retry delay. Since all our tests transferred the same amount of data, downloads were faster than uploads. For faster synchronizations (downloads in our tests), a shorter retry delay is advantageous. In a real-world scenario, if clients are making large downloads but only small updates, a different retry delay would be ideal. By running tests that mirror the types of synchronizations that you will have, you can determine an optimal retry delay for your specific scenario.

**Test 4: Scalability with respect to number of clients**

The next test varied the number of clients while keeping the number of worker threads at the best value from Test 1, to see how MobiLink scales with the number of clients. The number of synchronizations per client was adjusted in order to ensure that the same amount of data would be transferred overall, and that the size of each synchronization was constant. For example, with 10000 clients we used 1 synchronization (of each type) per client, with 500 clients we used 20 synchronizations per client, and with 200 clients we used 50 synchronizations each. The following were kept constant:

♦ 9 MobiLink worker threads
♦ Clients on 10 P4-3.2 GHz computers
♦ 1000 rows per client synchronization (with 92 bytes per row)
♦ Total of 10000 synchronizations

The tests were performed twice—once with a retry parameter of 10 seconds, and once with a retry parameter of 1 second.

The results are shown in the following graph and table. The graph shows the synchronization time, (in seconds) versus number of remote UltraLite clients each synchronizing 1000 rows of 92 bytes at a time. Each point represents a total of 10000 synchronizations; with fewer clients more sets of synchronizations were performed. The retry delay was set to 10 seconds for the solid lines and 1 second for the dashed lines.
The following table shows the results for synchronization time, in seconds, versus the number of remote UltraLite clients each synchronizing 1000 rows of 92 bytes at a time. Each value represents a total of 10000 synchronizations; with fewer clients, more sets of synchronizations were performed. The retry delay was set to 10 seconds when the number of clients exceeded the backlog threshold of 1500.

<table>
<thead>
<tr>
<th>Number of clients</th>
<th>Download</th>
<th>Update</th>
<th>Delete</th>
<th>Insert</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>467.572</td>
<td>2203.655</td>
<td>1320.728</td>
<td>990.826</td>
</tr>
<tr>
<td>500</td>
<td>319.45</td>
<td>2045.412</td>
<td>1172.983</td>
<td>797.225</td>
</tr>
<tr>
<td>1000</td>
<td>242.37</td>
<td>1994.253</td>
<td>1116.224</td>
<td>771.721</td>
</tr>
<tr>
<td>2000</td>
<td>422.749</td>
<td>1978.61</td>
<td>1111.551</td>
<td>740.751</td>
</tr>
<tr>
<td>5000</td>
<td>638.079</td>
<td>2007.595</td>
<td>1116.415</td>
<td>763.686</td>
</tr>
<tr>
<td>10000</td>
<td>644.159</td>
<td>2071.628</td>
<td>1117.33</td>
<td>772.814</td>
</tr>
</tbody>
</table>

This table shows the results for synchronization time, in seconds, versus the number of remote UltraLite clients each synchronizing 1000 rows of 92 bytes at a time. Each value represents a total of 10000 synchronizations; with fewer clients, more sets of synchronizations were performed. The retry delay was set to 1 second when the number of clients exceeded the backlog threshold of 1500.
The chart shows the results for retry parameters of ten seconds (the solid lines) and 1 second (the dashed lines). These retry times were chosen by looking at the results from the previous test. Setting the retry parameter to 10 seconds optimized the synchronization for updates and deletes. A retry parameter of 1 second improved the throughput for downloads, but caused reduced throughput with the other types of synchronizations.

As you can see from the graph, the faster retry delay decreases the time for download synchronizations. However, it significantly increases the synchronization time for the update and delete synchronizations. The reason for the increase in time is because the client computers are retrying more frequently, and failing the requests is using more of the server resources. This diverts server resources that could otherwise be used to process existing clients in the queue.

Unless synchronizations are download only, a fast retry delay would result in increased average synchronization times. This shows that it is important to run tests that accurately reflect how the system will ultimately be used.

In general, this test showed that MobiLink scales very well with additional remote clients. Although the best download throughput is obtained with 1000 clients, with more clients the throughput decreases, and then stabilizes, with little change between 5000 and 10000 remotes. The decrease corresponds with the backlog parameter of 1500; it seems that the need to retry shifts throughput to a different, but scalable, range.

The increase in download time between 1000 and 5000 clients is nearly insignificant when compared to the time it takes to synchronize updates or deletes. For this reason, we recommend that you optimize the retry parameter based on the time taken by your specific synchronizations.

For delete, update, and insert synchronizations, a minimum synchronization time is reached at 2000 clients. As more clients are added, the synchronization time stays roughly constant for inserts and deletes, while rising very gradually for updates. The likely cause of this is the large number of retries made by the clients, since the update synchronizations take such a long time compared to the other synchronizations.

The observed scalability suggests that tests with a smaller number of clients could be used to estimate results with a larger number of clients.
When you consider using MobiLink for a large number of clients, remember the following points:

♦ It is important to run tests that accurately simulate real-world use to minimize synchronization times.

♦ In our tests, MobiLink reached a minimum average synchronization time when 1000 clients synchronized simultaneously, and scaled fairly linearly as more clients were added. With greater than 1000 clients, the need to use a backlog and have clients retry lowered the throughput.

**Test 5: Size of each synchronization**

This test varied the number of rows transferred in each synchronization, to see the effect of synchronization size on performance. We used 1000 clients so that the number of rows per synchronization could be widely varied without changing the total number of rows synchronized.

In this test, the number of rows per synchronization was changed without changing the total amount of data transferred by changing the number of synchronizations per client. For example, each client could synchronize 10000 rows 1 time, 5000 rows 2 times, 2000 rows 5 times, and so on. The following were kept constant:

♦ 9 MobiLink worker threads

♦ Clients were on 10 P4-3.2GHz computers

♦ Total of 10000 92-byte rows synchronized per client, 20 million rows overall

The results are shown in the following chart and table. This chart shows throughput versus the number of rows per synchronization for 1000 remote UltraLite clients. Each point represents a total of 10 million 92-byte rows synchronized; with fewer rows per synchronization more sets of synchronizations were performed.
The above table shows the results for throughput, in rows per second, versus the number of rows per synchronization for 1000 remote UltraLite clients. Each value represents a total of 10 million 92-byte rows synchronized. With fewer rows per synchronization, more sets of synchronizations were performed. For example, with 10000 rows/sync, one sync was performed by each of the 1000 clients. With 5000 rows/sync, each of the 1000 clients performed 2 syncs.

The size of synchronizations has an effect on the throughput. A large number of small synchronizations are the slowest, and the synchronization times get faster as we move towards a smaller number of larger synchronizations. This is consistent with a fixed overhead for each synchronization, and its significance is diminished with larger synchronization sizes.

This becomes clearer when we plot the net average time for a synchronization (the total time divided by the number of synchronizations) versus the number of rows synchronized, as shown in the following chart and table.

This chart shows the average synchronization time, in seconds, for one synchronization versus the number of rows synchronized for 1000 remote UltraLite clients.

<table>
<thead>
<tr>
<th>Rows/sync</th>
<th>Download</th>
<th>Update</th>
<th>Delete</th>
<th>Insert</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>4426.207</td>
<td>5797.104</td>
<td>5217.821</td>
<td>3832.586</td>
</tr>
<tr>
<td>100</td>
<td>2372.494</td>
<td>3948.929</td>
<td>3149.418</td>
<td>2154.701</td>
</tr>
<tr>
<td>200</td>
<td>975.461</td>
<td>2902.951</td>
<td>2247.231</td>
<td>1430.015</td>
</tr>
<tr>
<td>500</td>
<td>441.705</td>
<td>2528.332</td>
<td>1443.502</td>
<td>1037.213</td>
</tr>
<tr>
<td>1000</td>
<td>303.288</td>
<td>2134.48</td>
<td>1175.286</td>
<td>891.332</td>
</tr>
<tr>
<td>2000</td>
<td>250.163</td>
<td>1927.665</td>
<td>1105.329</td>
<td>720.615</td>
</tr>
<tr>
<td>5000</td>
<td>247.863</td>
<td>1943.733</td>
<td>941.877</td>
<td>753.447</td>
</tr>
<tr>
<td>10000</td>
<td>195.638</td>
<td>1973.423</td>
<td>934.278</td>
<td>782.693</td>
</tr>
</tbody>
</table>
This table shows the average synchronization time, in seconds, for one sync versus the number of rows synchronized for 1000 remote UltraLite clients.

<table>
<thead>
<tr>
<th>Rows/sync</th>
<th>Download</th>
<th>Update</th>
<th>Delete</th>
<th>Insert</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>22.1</td>
<td>29</td>
<td>26.1</td>
<td>19.2</td>
</tr>
<tr>
<td>100</td>
<td>23.7</td>
<td>39.5</td>
<td>31.5</td>
<td>21.5</td>
</tr>
<tr>
<td>200</td>
<td>19.5</td>
<td>58.1</td>
<td>45</td>
<td>28.6</td>
</tr>
<tr>
<td>500</td>
<td>22.1</td>
<td>126.4</td>
<td>72.2</td>
<td>51.9</td>
</tr>
<tr>
<td>1000</td>
<td>30.3</td>
<td>213.4</td>
<td>117.5</td>
<td>89.1</td>
</tr>
<tr>
<td>2000</td>
<td>50</td>
<td>385.5</td>
<td>221.1</td>
<td>144.1</td>
</tr>
<tr>
<td>5000</td>
<td>123.9</td>
<td>971.9</td>
<td>471</td>
<td>376.7</td>
</tr>
<tr>
<td>10000</td>
<td>195.6</td>
<td>1973.4</td>
<td>934.3</td>
<td>782.7</td>
</tr>
</tbody>
</table>

As you can see, there is a near linear relationship between the average time for a synchronization and the number of rows synchronized.

In an analogous test with only 50 remote clients, we confirmed that the same trend exists. The linear relationship observed thus seems independent of the number of clients attempting to synchronize at the same time.

When considering the sizes of your synchronizations, remember that there is some per-synchronization overhead, so throughput rates are slower for smaller-sized synchronizations.
Hardware requirements

In order to assess the relative hardware requirements of MobiLink, we ran it on the same computer as the consolidated database, under stress, and watched the CPU utilization for the processes. Since we were running on Windows 2003, we used Task Manager to see the overall CPU utilization, and the CPU usage attributed to MobiLink and Adaptive Server Anywhere.

The overall CPU utilization was usually 98% to 100% during timed synchronizations. The exceptions were for downloads with a small number of clients, and for the smallest number of worker threads. When the CPU utilization was around 100% for all processors, we were confident that MobiLink and Adaptive Server Anywhere were being stressed, and that the bottleneck was processing speed on the server.

During timed synchronizations when total CPU utilization was around 100%, the CPU utilization for MobiLink varied from 25% to 35%, and the utilization for Adaptive Server Anywhere varied from 65% to 75%. This ratio was observed for all types of synchronizations: downloads, updates, deletes, and inserts. This suggests that if MobiLink and Adaptive Server Anywhere were on different computers, the computer running MobiLink would need less than half the processing power of the computer running Adaptive Server Anywhere in order to saturate Adaptive Server Anywhere. Unlike the consolidated database, MobiLink uses very little disk access (if you have sufficient RAM and have set cache sizes appropriately), so it does not have the same requirement for fast disk access. During these tests, the consolidated database was not disk bound. Many consolidated databases are disk bound, which could result in different performance characteristics.

To ensure that the network interface was not a bottleneck, the server computer had two network ports. Tests were repeated where half the clients used one of the network interfaces, and the other half used the other. This did not result in a change in performance.

When you consider the hardware requirements for MobiLink, remember the following points:

♦ MobiLink should need less processing power, and much less disk capacity or performance, than the consolidated database.

♦ To saturate an Adaptive Server Anywhere consolidated database, MobiLink seems to require less than half the processing power of the consolidated database.

Recommendations

This section summarizes the MobiLink performance tips mentioned previously, gives you advice on how you might perform small-scale tests to predict how MobiLink will perform in your large-scale deployment, and also predicts the hardware resources you should dedicate to MobiLink.
Performance tips

Following these tips will help you to get the best performance out of MobiLink:

♦ Be very careful to avoid contention in your synchronization scripts. In our experience, this is the most common performance problem in MobiLink setups. Be sure to test the concurrent performance of your scripts.

♦ Use the smallest number of MobiLink worker threads that gives you optimum throughput (-w option). For example, 9 worker threads gave optimum throughput in our tests. For slower clients, you may need more worker threads but limit the number that can upload simultaneously (-wu option). Keeping the number of worker threads as small as possible reduces the chance of contention in the consolidated database, the number of connections to the consolidated database (which are time consuming to create), and the memory you should use for caches.

♦ Set the maximum number of MobiLink database connections (-cn option) to be your typical number of synchronization versions times the number of MobiLink worker threads to reduce the need for MobiLink to close and create database connections.

♦ It is important to note that the MobiLink server and consolidated database operate as a system. Tuning one without tuning the other is less effective than tuning them together.

♦ Use an upload cache (-u option) that is larger than the size of your largest upload stream times the number worker threads to avoid having the upload cache overflow to disk.

♦ If your rows contain BLOB data, you can avoid having the BLOB cache access disk if you use a BLOB cache (-bc option) that is larger than twice the largest BLOB data in a row times the number of worker threads.

♦ Ensure that the computer running MobiLink has enough physical memory to accommodate the upload and BLOB caches in addition to its other memory requirements.

♦ Dedicate enough processing power to MobiLink so that, if needed, it can saturate the consolidated database. In our 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 fully loaded.

♦ 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.

Large deployments

A single MobiLink synchronization server can handle tens of thousands or hundreds of thousands of remote databases. In these tests, we looked at up to
ten thousand clients synchronizing simultaneously. Depending on the type of synchronization, this took from 10 to 30 minutes, which is equivalent to 100,000 to 600,000 remote databases synchronizing over an hour, or 2.4 million to 14.4 million synchronizations in a day. Scalability depends primarily on the volume of data being synchronized, so the number of clients supported varies by the size of the average synchronization. The hardware requirements for this were quite modest; MobiLink used less than a third of the CPU time on a computer with four 2.2 GHz Pentium 4 Xeon processors. Thus, a single instance of MobiLink can handle a very large number of remote databases, with significantly less hardware than that required by the consolidated database to which it is connected.

If you determine that a single instance of MobiLink running on a dedicated server computer would not meet your performance or availability requirements, you can use multiple MobiLink synchronization servers. MobiLink does not maintain state in memory between connections, so you can use multiple MobiLink synchronization servers for high availability and scalability. For example, you could use multiple MobiLink synchronization server computers, and employ a third-party load-balancing device (such as Cisco’s LocalDirector and CSS devices, or F5’s BIG-IP device) to make them appear to your clients as one server computer. In such a setup, the load balancer periodically polls each MobiLink synchronization server to gauge its load from its responsiveness, and to see if it is still running. If one MobiLink synchronization server becomes unavailable, new connections are only directed to the servers that are still running, providing failover capability. This is the simplest and most effective way to scale beyond a single instance of MobiLink. Customers have successfully employed this architecture in scalable and highly-available data synchronization solutions.

A synchronization hierarchy is another architecture that employs multiple MobiLink synchronization servers. In a synchronization hierarchy, clients synchronize with secondary consolidated databases that are periodically synchronized with a primary consolidated database. You could use MobiLink for both synchronization layers, as long as the secondary consolidated databases are Adaptive Server Anywhere databases. A synchronization hierarchy is much more complicated than using a load balancer. Also, it does not address scalability directly since you still have the problem of synchronizing the total amount of data with the primary consolidated database, perhaps through a single MobiLink synchronization server (if you use MobiLink for both layers). However, a synchronization hierarchy may fit your infrastructure or business needs. For example, if your MobiLink synchronization servers need to be geographically distributed or if you want to avoid having your primary consolidated database tied up with lengthy synchronizations from clients with slow processors or slow network connections.

Applicability

While the tests reported here can give you some idea of quantitative MobiLink performance, if you want to assess performance for your MobiLink setup, you should do tests using your schema, data, consolidated database, synchronization scripts, and clients. For example, this report does not address using non-Adaptive Server Anywhere consolidated databases.
If you want to do this type of testing, we suggest the following procedure:

1. Determine your synchronization needs. You should know or estimate how many users will be synchronizing data over the time period of interest, to estimate how many would be likely to synchronize simultaneously. You should also determine the characteristics of a typical synchronization, including the type and size of data that is uploaded and downloaded, and whether the data was inserted, updated, or deleted.

2. Set up a pilot implementation. As much as possible, use your actual synchronization scripts, consolidated database, and client and server hardware. Create clients that perform the typical synchronizations with typical data, using or simulating actual clients on your intended client hardware and network. For the number of clients, if it is impractical to use the total number you expect, then you can test with a smaller number. You should at least pick a multiple of the number of MobiLink worker threads you want to try. For example, you may choose to use 40 clients and run tests with 5, 10, and 20 worker threads. When you have many more clients than worker threads, optimizing throughput for that number of clients should also give optimal throughput for a larger number, since excess clients are either queued or refused via the backlog. One might consider running multiple client applications on the same physical machine.

3. In your pilot implementation, you may want to run MobiLink and your consolidated database on the same server computer. Then you could use the relative CPU utilization to help you decide how to allocate your hardware resources between MobiLink and the consolidated database.

4. Run MobiLink with the MobiLink Monitor and non-verbose logging to disk enabled. The Monitor will collect detailed timing information for each synchronization, and the log file will record any warnings or errors that occur. In our tests, we found that using the MobiLink Monitor (on a separate computer than the MobiLink synchronization server) does not affect throughput.

5. Perform your test synchronizations. For example, to test 20 clients you could recruit 20 volunteers to start synchronizations on 20 devices at the same time.

If you are interested in using the timing framework (called MLBench) that we developed for the tests reported in this whitepaper, contact iAnywhere Professional Services (http://www.ianywhere.com/support/services.html). iAnywhere Professional Services provides Performance and Tuning services, as well as Architectural Definition and Review, Prototyping, Solutions Development, and many other services.
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.

MobiLink technology includes components supplied by Certicom, Inc. These components are protected by patents.

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
Email  contact.us@ianywhere.com