WHITE PAPER

Designing Mobile Applications: Why Sync Is Central

A whitepaper from Sybase iAnywhere

SYBASE iAnywhere

INTRODUCTION

Many mobile business applications adopt an "always-available" or "occasionally-connected" architecture: they store data on the device so that the application can be used whether or not a network is available. Despite this local data store, the ultimate home of the data for business applications is not the device, but is a central data store on a server. Any piece of data may be shared among many devices and each device is periodically synchronized with the central data store over wired or wireless networks.

Data synchronization is a key technology for always-available mobile applications: it's the software that moves data from the device to the server, and from the server to the device. That doesn't sound too challenging does it? Upload a few items; download a few items. What could be difficult about that?

At Sybase iAnywhere we have talked to many development teams at customer sites who assumed that data synchronization could be treated as an afterthought, rather than as a central piece of their mobile application. They have come to us after getting their fingers burned, realizing that data synchronization was taking them a lot longer to implement than they anticipated.

We'd like to help you keep your own fingers burn-free.

The first purpose of this paper is a little depressing: to show you that data synchronization is more complicated than you might think. There are hidden difficulties that can trip you up when building a secure, scalable always-available application so that what seems to be a simple task turns out to be a tricky and involved problem. Unfortunately, online applications cannot be automatically and simply ported to work in an "occasionally connected" manner.

But don't let this get you down, because the second purpose of this paper is more cheerful: to show how MobiLink data synchronization technology from Sybase iAnywhere can deal with those problems, so you don't have to.

A SIMPLE EXAMPLE

To get a feel for data synchronization, let's look at a simple example that is one small piece of many applications: a list of contacts.

At this stage we'll say nothing about how the data is stored: in principle, you could store a set of these records as a plain text file, an XML file, a database, or in an object store. But, being a list, it makes sense to store each contact in a separate record. Most of the fields in the record will be descriptive information, such as name, address, time of most recent contact, and so on. In addition, the record itself needs to be distinguished so that it can be uniquely accessed (there may be two Jane Smiths, or two contacts at the same address, and so on), so we give it a unique ID value.

A typical record looks like this:

Contact ID	Name	Address	City	Last Contact
102	Jane Smith	123 Evergreen Terrace	Springfield	2007-05-31 10:00

The "home" of the contact list is on the servers of your organization, and the first challenge for data synchronization is to deliver to each mobile user a copy of the contacts that he or she needs, *and only those*.

We'll divide up the contacts by city, so that one group of users gets the contacts in Springfield, another group gets the contacts in Shelbyville, a group of managers gets both sets of contacts, and so

on. This kind of division, in varying forms, is typical for all sorts of data in business applications: think of delivery locations, customer addresses, and so on.

The second challenge for data synchronization is to keep this list of contacts correct as changes are made. Some changes are made at the server (imagine a contact calling the company to tell them of a change of address, for example) and some are made on a mobile device (the Last Contact time will be updated when a mobile worker talks to a customer). To see what these two synchronization challenges really involve, lets walk through a few simple scenarios and see what jobs the data synchronization system has to do.

DOWNLOADING THE CONTACT LISTS

The first time a user starts her application, it downloads the list of contacts from the server. The server could store the data in any format but for now let's assume it's a central database: we call this the *consolidated database* as it consolidates all the information scattered around the mobile devices.

It would be possible to just download the complete contact list to each and every user, but that would be wasteful in a number of ways:

- Network traffic – downloading unnecessary data over a wireless connection is a waste of money, and over any connection it is a waste of time.
- Data storage – storing unnecessary data on devices uses up memory and may require the purchase of more expensive hardware. For some applications, unfiltered data will not fit on a single device.
- Application performance – filtering unnecessary items out of searches or lists will slow down all aspects of application behavior.
- User experience – if the application exposes irrelevant data to the user, it also has to help them navigate around this data.

To solve the problem, we need to *partition* the data. When a mobile user connects, they need to be identified to the system. Information associated with that user can be exploited to download the correct information. For example, the user could be mapped to a separate city (or list of cities) in a table in the consolidated database.

Row ID	User ID	City
1	0001	Springfield
2	0002	Shelbyville
3	0003	Springfield
4	0003	Shelbyville
5	...	...

The rows for user 0002 can then be downloaded by matching all rows with a City field of Shelbyville, while user 0003 gets both the Shelbyville and the Springfield contacts. Without going into too many details, MobiLink handles this problem by executing a SQL query against the database to select the rows to be downloaded:

```
SELECT "Contact ID", Name, Address, City, "Last Contact"
FROM Contacts
WHERE City IN
    ( SELECT City FROM Users WHERE User ID = {ml s.username}
```

Where `{ml s.username}` indicates the user ID, and is supplied by the MobiLink client when it contacts the server.

ADDING A CONTACT

Now that each user has their list of contacts, what happens when one of them adds a new contact. Imagine user 0002 adding this contact:

Contact ID	Name	Address	City	Last Contact
903	Eric New	123 Deciduous Drive	Shelbyville	2007-05-31 2:15PM

This new record needs to be uploaded to the server, and it also needs to go to user 0003.

Thinking about this problem makes it clear that what is being sent back and forth during efficient data synchronization is not "the data" but *changes* to the data: we want to send just the new row *and no others* to the server. If you implement data synchronization on your own, you need to find a way to pick out the new contacts from all the others – perhaps by adding a timestamp to the table and tracking the last time you synchronized. MobiLink clients track the changes themselves, so that you do not need to track the changes in your application.

Here is another challenge: looking at the new contact record, you will see that it is assigned a Contact ID value of 903. This value must be unique across the entire system, and yet the mobile device does not have access to records being added by other users on other devices. How can we guarantee the uniqueness of the key value?

There are several solutions to the problem, and the right one for you depends on your consolidated database. One mechanism is to use Universal Unique Identifiers (UUIDs) – long strings of alphanumeric values constructed from device-specific and time-specific data in such a way as to be guaranteed unique – but your consolidated database must support UUIDs. Another way is to partition the set of possible keys across the mobile devices. To avoid you having to implement your own mechanism for tracking key values, both of these solutions (and others) are supported by MobiLink.

DELETING A CONTACT

Things get more tricky when you think about deleting records. Imagine that a mobile user learns that a contact has moved to a distant city, so that his contact information needs to be deleted from the whole system. The mobile user deletes the record from the data store on her mobile computer. That delete needs to be sent up to the server as part of the synchronization, but (of course) the record is no longer present to be sent. How can we send up a record that no longer exists?

If you were implementing your own synchronization system, you would need to hold a tracking table that keeps deleted rows around until the delete operation has been sent to the server and the server has confirmed receipt, and then clears out the tracking table so that it doesn't get sent again. MobiLink clients have this tracking mechanism built in to them, so that application developer does not need to implement the additional code to track these deletes.

If a contact is deleted at the *consolidated* database, a separate solution is needed to ensure that the corresponding row is deleted at the mobile device. MobiLink provides design-time options to be able to track and download deleted rows from a consolidated database. For example, MobiLink can create shadow tables or a status column for all tables in your database.

UPDATING A CONTACT

Something similar happens when you update a contact's information. Imagine that two mobile workers update information about the same contact, but the first one to update their local data is unable to synchronize their device until the next day, while the second mobile worker synchronizes immediately.

When the second mobile worker synchronizes, he will send up an updated "Last Contact" time, which replaces the value at the server. When the first mobile worker synchronizes the next day, her

(earlier) “Last Contact” time is sent up. In this case, the correct behaviour is that the older “Last Contact” time should *not* replace the newer “Last Contact” time at the server.

A synchronization system needs to do two things here. The first is to identify the fact that a conflict has occurred: a record has been changed at two separate mobile computers, and just applying the changes in the order they are synchronized does not always do the right thing. The second is to take the right action to resolve the conflict: in this case, keep the later of the two times.

Depending on the nature of the data that is in conflict, and the business rules governing that data, different rules need to be implemented to resolve the change. For example, in an inventory table you might want uploads to be additive while in another situation you may want one user’s input to overwrite another’s.

To identify a conflict, you need to send up the “old” version of the values as well as the “new” version, so that you can check if the data on the server has been changed since the last time it was downloaded to the device. This is similar to the case of deleted records: information that is no longer in the database must nevertheless be sent up to the server. MobiLink clients handle the problem automatically, while the MobiLink server provides a built-in conflict resolution mechanism and also allows you to build custom conflict rules for special situations.

NON-SYNCHRONIZED DELETES

We have walked through the synchronization of deletes, but there are often cases where, for reasons of performance, space or security, you want to delete rows on the device but *not* at the server. A typical example is if you are keeping a diary of events, and only want to keep the most recent month’s events on the device. Obviously you don’t want to wipe out the old events from the entire system, just from your device.

MobiLink clients provide a way to turn off change-tracking, so that you can clean out old records and then resume change-tracking. It’s just one more non-obvious feature that you need to implement in a serious synchronization system.

Synchronization is starting to look complicated now: tracking changes on the device and at the server, sending the right changes, making sure that unique key values are preserved, identifying and resolving conflicts when they occur – this is a substantial list of tasks for a synchronization system to implement. But we are not finished yet.

WHAT NEXT?

The simple changes we have looked at so far are all for a single list, with a well-defined set of rules for who gets which contacts. But in the real world things change. Some users will be added, others will change duties; new versions of the application will be rolled out, from minor tweaks to major revisions. A data synchronization system has to provide the facilities you need to ensure that your mobile system can continue to evolve.

REASSIGNING DATA

Here is one common scenario: a promotion results in user 0002 working with contacts from Springfield rather than from Shelbyville. At the next synchronization, his Shelbyville contacts need to be deleted from the device and the Springfield contacts need to be downloaded.

This kind of reassignment of data happens in many circumstances. For example, schedule changes for field service workers can lead to a rearrangement of routes. If a repair worker is held up at an appointment, other customer visits have to be reassigned to other workers’ schedules.

The synchronization system has to make the following changes:

1. Complete a final upload from user 0002's device of the "old" set of data (any changes to Shelbyville contacts).
2. Download a set of operations that delete the Shelbyville records.
3. Download the Springfield contacts to user 0002.

Doing this for the contact list is one thing. If each contact has separate information associated with them, held in other tables (items purchased, say, or appointments) then that information must be cleared up as well, while respecting foreign key constraints.

Again, MobiLink provides built-in features to help this kind of change get synchronized. You would update the user table (above) and assign a new value of the city to user 0002. MobiLink provides a separate hook for downloading deletes at the server that allows easy wiping out of records from the client to be triggered by the reassignment. Cascading deletes at the client ensure that all associated data in other tables is automatically cleaned out, without having to take up the network bandwidth of downloading the deletes for each record explicitly. Finally, the download of new data follows automatically from the reassignment.

APPLICATION CHANGES

Getting a single application into the field is one step along the road to a mobile solution. A long-term solution has minor and major system upgrades, perhaps pilot projects, and other special cases where a non-standard application must be used. Application management can be handled as part of a device management and security solution such as Afaria from Sybase iAnywhere; here we focus on the data requirements associated with upgrades.

A new application may need different synchronization logic to the old application. Additional columns, new tables, and more elaborate logic may be added on.

The upgrade problem is complicated by the occasionally-connected nature of mobile applications: an upgrade cannot happen instantaneously. For some period there will inevitably be two versions of an application running against the same server. If you carry out pilot projects with small portions of the workforce, then multiple application versions may be the norm.

The ability to implement multiple sets of synchronization logic on a single server requires a separation of the logic from the underlying schema. MobiLink provides this by allowing multiple versions of synchronization logic to be implemented: clients supply a version string when they synchronize, and the right set of logic is applied. This kind of structure avoids the problem that some "publish-and-subscribe" systems have, where the data is tied to the application.

Beyond the logic in the server, the UltraLite database together with MobiLink provides a schema upgrade capability, so that database upgrades can be rolled out to existing users.

Applications are not the only thing that get upgraded in a mobile system. Hardware and operating systems also receive periodic upgrades. A synchronization system must stay compatible with new tools and platforms. A synchronization solution written in embedded Visual Basic for Windows Mobile 2003. may have to be moved to a .NET language on Windows Mobile 5.0 and Windows Mobile 6, and the problem is even worse if you chose to move to another device type.

DATA INTEGRITY IN THE FIELD

Getting synchronization logic to work in a testing environment, with a reliable high-speed network and a handful of users, is one thing. Making it work in the field - with many users, intermittent wireless networks, no IT access to the devices and so on – is something else. Events that are rare in small-scale, controlled environments become important in the field. Here are a few.

INTERRUPTED SYNCHRONIZATION

If network coverage is lost part-way during a synchronization, there are several concerns:

- The data on each side must be left in a correct and consistent state so that applications can continue working properly. Given the frequency with which users leave network connectivity, turn off mobile devices, drop them, take out SD cards and so on, anything that leaves a possibility of an inconsistent state is going to lead to problems. Incorrect query results are one symptom of inconsistent data, of course, but application errors can also result when – for example – a row that is expected to be present does not exist. And once you start making changes to incorrect data and synchronizing those changes, the lack of correctness can propagate throughout the system.
- To enforce data correctness, each data upload or download must be atomic (take place in a single transaction). If referential integrity is broken at the consolidated database, for example, the consistency of the central repository of the data can be compromised. Conflict resolution must take place within the same transaction as the other changes that are being made. On the other hand, error reporting operations must take place outside the main transaction so that the error report is not lost if the transaction fails.
- The synchronization system must successfully track what the state of that data is so that it can send the proper set of changes in subsequent synchronizations. This requires an acknowledgement step that is guaranteed to be atomic. MobiLink clients have such an atomic operation.
- If a partial download is accomplished, the application should not have to download that data again. A resumable download feature is particularly important over wireless networks, where data transfer can be expensive.
- The dropped connection must not tie up resources, particularly at the server.

These are each challenging issues. What if changes to a two-table database are being downloaded, and the connection is dropped? If a new contact is downloaded, but the company they work for is not, can the application handle that?

PERFORMANCE

The bookkeeping required to track changes, track the state of each client at the server, and maintain the state of synchronization, can cripple some synchronization systems. The MobiLink system has been built over years to optimize these operations and to scale up to many users synchronizing simultaneously. It provides separate thread pools to manage database connections and client connections; configurable timeouts to ensure that resources are used properly; a robust protocol to make sure that connections are kept alive if expensive operations are being carried out on device or in the server; and has all the gear to ensure the integrity of the data is maintained. MobiLink works efficiently with a few remote client, or hundreds or thousands.

AND AFTER THAT…

This article just scrapes the surface of what is needed in a robust data synchronization system. As you develop mobile applications, it becomes obvious that there are other features you may want to tap into. Perhaps some changes are more urgent than others, and need to be synchronized with higher priority than the remainder; what about secure authentication at each point in the chain; does your data need to be encrypted; and what about high-availability, or synchronizations initiated by the server rather than by the client?

MobiLink already handles these issues. As mobile applications grow in scope and complexity, these and many other core capabilities are best built into a data synchronization layer: a mobile data platform that can be used to build whatever mobile applications you need.