Designing Mobile Applications for the Enterprise

Eric Giguère, iAnywhere Solutions — a Sybase company

As more users go mobile, organizations are discovering that it's a challenge to build and deploy applications for the wide variety of devices in use. Occasionally connected devices (like Palm handhelds) need to store and process data offline while the device is away from the network; always-connected devices (like cellphones) have access to data but can suffer resource constraints. Connection in the second instance takes place via either a dialup (a circuit-switched connection in standard 2nd generation digital and 1st generation analogue networks) at low speed, or a packet-switched connection (enhanced 2nd generation/3rd generation) that provides an 'always connected' model without the need for dialup.

This session paper explores these different connectivity models, looking at issues like synchronization, conflict detection and resolution, and security.

> *Portions of the material in this paper cover similar ground to Chapter 7 of my book,* Palm Database Programming: The Complete Developer's Guide*, published in 1999 by John Wiley & Sons.*

What is a Mobile Application?

There are many definitions of what constitutes a mobile application. For the purposes of our discussion, we define it as follows: an application is *mobile* if it runs on a portable computing device and is either always or occasionally connected to a network. This definition includes applications that run on notebook computers, personal digital assistants (PDAs) and cellphones, among others. It also implies some form of client-server architecture, where the application running on the device is a client of some service made available through the network.

What particularly interest us in this paper are *enterprise* mobile applications, which are basically mobile applications that are developed for and deployed by a corporation, potentially to thousands of users. Deployment issues aside (and not to trivialize them — the appeal of thin-client, browser-based applications is to reduce deployment costs), these types of applications need to access and share the same back office data that other, non-mobile applications use. This means access to the database and application servers holding the corporation's data and business logic.

Connectivity Models

Access to the back office is performed through some kind of network connection. However, the connection does not necessarily have to exist permanently — some applications can be written to work with occasional or sporadic network connectivity. It is useful then to distinguish three different connectivity models when describing how an application connects to its server component:

- In the **always connected** model, the application can't function at all without the network connection. In other words, the server is critical to the proper execution of the application. A typical example of this is an application running in a cellphone-based HDML/WML microbrowser, or any kind of thin-client application. While it's true that these devices all have caches and can function to some degree without a connection, cache size is limited, the user doesn't have much control over it, and applications must be carefully architected to work without a server component, which also means the device has to support some kind of scripting language (like WMLScript).

- In the **always available** model, the application *expects* a network connection to be available, but can still function when that connection is unavailable. This model can also be referred to as the **occasionally disconnected** model. The server component is important, but not critical. If your service personnel are out in the field, for example, they could easily go outside the wireless coverage area when visiting rural customers (or even while inside certain buildings), but they still need to be able to access and enter data.
- In the **occasionally connected** model, the application works primarily in an offline mode, only occasionally connecting to the network, usually on demand by the user or at regularly scheduled times. The server piece is somewhat optional. The classic example of this is Palm synchronization, of course, but even a notebook computer follows this model when it's disconnected from the corporate network.

Another way to compare these models is to ask yourself where the data resides. If the data is completely stored on a remote server, then you're using the always connected model, because you *need* a network connection to get to the data. If the data is remote but you have a local cache for failover purposes, you're using the always available model. Finally, if the primary data source for your application is local to the device, you're using the occasionally connected model.

It's important to be aware of which model your application requires. There are no real hard rules: the developer has to decide which model is appropriate based on the expected use of the application and the capabilities of the device. It may even make sense to have two different versions of the same application, each using a different connectivity model. However, it is usually harder to change an always connected application into an occasionally connected application than the other way around. (If you presume that a connection is always available, you won't necessarily worry about data caching and related issues.). Choosing the right model at the beginning of development is therefore critical.

Data Synchronization

No matter which model you're using, at some point your application is going to have to perform some kind of **data synchronization**. Data synchronization simply refers to the exchange and transformation of data between two applications maintaining separate data stores. Note that I'm using "application" in a very general sense here — a database server can be considered an application for these purposes.

Data synchronization is most important for the always available and occasionally connected models, since without a network connection the client and server pieces can't notify each other of changes as they occur. But even the always connected model has to deal with synchronization issues, because there's always some kind of time delay involved that can cause conflicts when data is updated by different users. You might think that locking the data a client is going to modify is a simple solution, but it rarely works in practice — what if the user locks some important data and then goes on vacation for a week?

And then there are transaction issues. Many changes to data are atomic in nature, such that if one of the changes fails, *none* of the changes must succeed. This is referred to as **rolling back** the changes. If all the changes succeed, on the other hand, they are **committed** and made permanent. Transactions ensure the integrity of the data.

Synchronization isn't a new problem. Database vendors have been doing it for quite some time now with **data replication**, where data is 'mirrored' from one database to another for faster access and failure recovery. Replication is really a special case

of synchronization, although it tends to be one-way, whereas data synchronization in general can be two-way (bi-directional).

Note that in order to be synchronized, data *doesn't* have to be exchanged verbatim. In fact, there's usually some kind of data transformation that occurs, whether because the same data is stored in different ways, or only subsets of the data are used, or there's a need to encrypt or encode data (perhaps for localization purposes).

Anyone who's tried to write their own synchronization code, however, will admit that it's hard to do, because you need to:

- Understand and interface to different data stores
- Develop a synchronization protocol appropriate for the network connection
- Map the data from one data store to the other
- Detect and report conflicts and errors
- Interface with a transaction management service

In particular, your synchronization code has to ensure the integrity of the data. Corrupting data can cripple or even disable other applications or users, let alone the problems it can cause with your own application. If you really want to write your own code, remember the cardinal rule: if in doubt, abort the synchronization and undo any changes you made. Better to have the application report that the synchronization failed, than to have bad data make its way into the back office.

You'll no doubt find that in the long run, it's cheaper and simpler to buy someone else's synchronization solution than to code it yourself. Doing so lets you concentrate on the application, after all, and not on the synchronization protocol you're otherwise inventing. All the major database vendors (like Sybase and Oracle) have products for database synchronization, and there are companies like Pumatech that deal with application-level synchronization.

Synchronization Strategies

Even if you're using somebody else's synchronization solution, though, there's still work to do on your part. You have to decide on the synchronization strategies that are appropriate to your application.

Data Tracking

The first thing you need to concern yourself about is how any changes to data are tracked. There are two basic approaches:

- **Snapshot** synchronization takes a complete 'snapshot' of the data. This is often used for the initial synchronization, but for a large amount of data that doesn't change very frequently, this generates too much network traffic.
- **Incremental** synchronization sends only the changes since the last synchronization. This minimizes network traffic, but requires extra bookkeeping at both ends.

The goal, of course, is to minimize the time required to perform synchronization.

Data Partitioning

Often, the client only needs to work with a subset of the data maintained by the server. This subsetting is referred to as **partitioning**. Tabular data, where data is

organized into rows and columns, lends itself well to two kinds of partitioning that can be used independently of each other, or simultaneously:

- **Horizontal** partitioning defines a subset of the rows. The idea is to include only the data required by the *user* of the application. For example, salespeople are likely to care only about their own customers contained within the corporate customer database.
- **Vertical** partitioning defines a subset of the columns. Only the data required by the *application* is included — it is independent of the user. For example, the application may only require a few of the important columns in the customer database, not some of the more esoteric ones like tax codes and credit ratings.

Horizontal partitioning is particularly important in the case of wireless applications, not only to minimize synchronization time, but also to avoid overwhelming a device's storage capacity.

Conflict Detection and Resolution

With multiple clients accessing and changing the same data, you'll soon have to deal with conflicting changes. The synchronization solution you're using will *detect* the conflicts, but it's up to you to *resolve* them. Conflict resolution rules are critical to preserving data integrity. Again, there are different approaches to conflict resolution:

- **Conflict avoidance** is the simplest resolution policy: structuring the data such that no conflicts are possible. Clients can use data in a read-only fashion, or else you can use partitioning to group data into mutually exclusive sets.
- **One side always wins** is another simple policy to implement: when conflicts occur, either the client or the server always wins. Palm devices, for example, let you choose to have the handheld overwrite the desktop, or vice versa.
- **Both sides win** seems like a good strategy, but it's usually better to choose another one. This strategy says that if a conflict occurs, make two copies of the data: one with the client changes and one with the server changes. The problem is that it rarely makes sense to have two different copies of the same data.
- **Custom conflict resolution** strategies go beyond these three simpler ones. For example, it may make sense to define a resolution policy based on whom the user is — managers' changes would always override those of their subordinates. Or you can try to merge changes intelligently, perhaps even passing the conflict over to a human for arbitration. This last method is likely to be needed quite a lot to begin with, but that requirement will probably decrease over time with system improvements and better training and update co-ordination.

Whenever a conflict occurs, though, you'll want to be able to notify users about it, or at least log it somewhere. If the client is the 'loser', it may need to rerun some calculations based on the updated data.

Primary Key Assignment

In database terminology, a **primary key** uniquely identifies a piece of data. For example, a list of employees would include an employee ID value as the primary key — using the name of the employee is a bad idea, since there could be more than one employee with the same name.

Synchronization depends on the existence of primary keys (whether the data comes from a database or some other source) in order to track the data as it moves from the server to the client and back. The synchronization process will always include the primary keys in the data exchange.

Generating a unique primary key is a problem when a client needs to create new data to store on the server. The two common techniques are:

- **Pre-generate primary key pools** for each user. As the user creates new data, the primary keys are pulled from the pool. The pool gets replenished as part of the synchronization process if it's running low. Of course, if the pool is too small then the user won't be able to create new data until synchronization is performed.
- **Generate "globally unique" keys** based on client-specific information. For example, a primary key for a customer ID could be generated by combining the customer's initials with their first order number (order numbers will generally be unique, if based on the order submitted to the company).

Alternatively, in some cases the client can get away with letting the server assign primary key values. Simply incrementing the number, however, is a bad method, as this could result in the situation where a number of devices try to allocate the same number, all thinking it is unused. If keys *are* allocated in this fashion, then the server may need to change key values to keep data in order; this demonstrates the need for bi-directional communication, since those changes must be communicated back to the client.

Security

Security is *always* an issue with mobile applications, but especially in wireless communications over public data networks. Security is about more than just encrypting data; it's also about authenticating users — verifying that they are who they say they are. This "login" process is so basic that almost every application requires it, even if it all it means is prompting the user for a user ID and password.

Ideally, of course, the security systems already in place in a corporation should automatically extend to include mobile applications, but this ideal expansion is not always possible. For example, the company may depend on virtual private networking (VPN) technology to provide remote computers with access to the corporate intranet. A VPN extends the corporate intranet out onto the Internet by encrypting all communications from the client to the intranet and otherwise blocking out non-authenticated clients. How do you then deal with devices with no VPN support?

These and other problems are slowly being addressed by device manufacturers, but in the short term you may be forced to work within a less secure environment than you'd like. If you can't secure the client, place limits at the server end to limit any damage that might occur if security is breached. For example, the server should only have access to the resources (especially databases) it requires. And the server should avoid using administrator privileges when accessing those resources if at all possible.

An interesting feature of many mobile devices is the ability to identify a device uniquely. The server, for example, can determine that a *particular* device is connecting to it. You may be tempted to use this information to associate a user or role with the device automatically, but don't abandon user authentication completely. Portable devices are easily lost or stolen, or (in the worst case) may even be cloned, so even hard-coded IDs may be duplicated. This is certainly the case with mobile phones, and a possibility with other devices as well. And if the

device uses a third-party gateway to access your site, consider authenticating the gateway as an added security measure.

So, if you do use automatic role association, be sure to provide a way to suspend a device's privileges quickly and easily upon notification of loss or theft.

Emerging Standards

As more developers write mobile applications, standards are beginning to emerge. The particularly notable ones are:

- **WAP:** The Wireless Application Protocol is an obvious set of standards for wireless thin-client communication, encompassing WAE, WSL, WTLS and WTP. Be aware, though, that when many people talk about WAP they're really just referring to WML. Some would argue that WML (and WAP) itself may be irrelevant in the years to come, superseded by XHTML Basic and other technologies closely based on existing Internet standards.
- **SOAP:** The Simple Object Access Protocol is a way to invoke methods on objects (the definitions of "method" and "object" are quite arbitrary) using the HTTP protocol, with request and response information encoded using XML. It's basically a portable, network-independent way to do distributed computing, although it's not really a replacement for low-level remote procedure calls due to the communication and processing overhead involved. But by leveraging HTTP, SOAP calls can be made through firewalls and across non-IP networks, while the data is easily validated because it's XML.
- **SyncML:** A consortium of companies defining portable data synchronization protocols, most of which are XML-based. They're expected to release their first specifications later this summer.
- **XHTML Basic:** A distillation of XHTML (an XML-compliant form of HTML) for small devices. XHTML turns HTML into an XML language, while maintaining compatibility with existing HTML clients (provided that you follow some simple rules). XHTML Basic is a core set of XHTML functionality suitable for small devices.
- **Java 2 Micro Edition:** A smaller virtual machine and stripped down and rewritten libraries are allowing Java to move back down to the client, making it possible to write end-to-end Java programs.

You'll be sure to see other standards arrive as the marketplace focuses more attention on mobile application building.

Summary

Designing a good mobile application is not trivial: there's a *lot* to think about if you want to make it fit in with your existing data and applications. It involves leading edge and evolving technology and standards, and generally requires a lot of learning on the part of everyone involved. Understanding the issues presented here will make your job a bit easier, and save you from making critical design mistakes.