Data Synchronization Patterns in Mobile Application Design

Zach McCormick and Douglas C. Schmidt, Vanderbilt University
{zach.mccormick,d.schmidt}@vanderbilt.edu

1. Introduction

As Internet-enabled devices become more prevalent in the form of smartphones and tablets, the need for mobile application development patterns grows in importance. Different technologies, such as Nokia's Symbian, Apple's iOS, Google's Android, and Microsoft's Windows Mobile, have arisen and will continue evolving to provide platforms for developing applications for mobile devices. These technologies build on years of experience developing flexible, open-ended frameworks and platforms, and the developers of these technologies have provided many resources for application developers. While best practices have been documented for nearly every component, such as the guidelines for Android Design[i] and the iOS App Programming Guide[ii], a comprehensive pattern collection or pattern language for mobile application development has not yet been produced, which exacerbates the difficulty of solving problems or conveying solutions effectively in this domain.

Many mobile applications are data-centric, and are designed to replace pocket atlases, dictionaries, and references, as well as create new digital pocket references for data that changes dynamically by leveraging technologies that did not exist in these form factors before. This paper is intended as an initial step in a larger work on mobile application development patterns and will focus only on patterns related to *data synchronization*, which involves ensuring consistency among data from a mobile source device to a target data storage service (and vice versa).

With some datasets, such as with Google Maps, it is impossible to store all of the data the application can leverage on the device, so specific strategies must be employed to synchronize the necessary data. With other datasets, such as stock prices, mobile applications that allow users to manage their portfolios are useless without the most recent data, so strategies must be employed to ensure users only see the most recent data. This paper describes common concerns related to data synchronization as a collection of patterns, grouped by the problems they address.

The patterns described here have been collected from examining open-source applications, inspecting the platforms and frameworks that comprise these mobile systems, evaluating of other pattern catalogs and languages for applicable patterns, and documenting our experiences developing mobile applications. Open-source examples and insights into the platforms and frameworks will be cited explicitly, and some examples of these patterns in popular consumer applications will also be mentioned.

2. Pattern Format

The patterns presented here were initially documented using a variant of the Gang of Four and POSA pattern forms, but have been abbreviated to remove the Structure, Participants, and Collaboration sections to reflect the types of patterns covered in this document. The following are the sections in our abbreviated pattern form.

Pattern Name

Intent
State the intent of the pattern

Problem
State the problem(s) that this pattern works to solve

Applicability
Constraints of the specific contexts that would elicit the use of this pattern

Solution
Explain how this pattern solves the problem considering the constraints of the context

Consequences
Anything resulting from the use of the pattern aside from solving the given problem

Examples/Known Uses
Explained examples of the problem and/or known uses of the pattern

3. Data Synchronization Mechanism Patterns
Data synchronization mechanism patterns address the question: "when should an application synchronize data between a device and a remote system, such as a cloud server?" This problem is common in mobile application design, and is often overlooked, but there is no one-size-fits-all solution. Instead, mobile application developers must consider the constraints of many factors, including network availability, data freshness requirements, and user interface design.

Data synchronization mechanism patterns are often architectural patterns. An architectural pattern is described as follows[iii]: "An architectural pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them." The following data synchronization patterns should therefore be viewed as structural organization schemas that address the question of when an application should synchronize data between a device and a remote system in a variety of different contexts.

Asynchronous Data Synchronization

Intent
Manage a data synchronization event asynchronously and without blocking the user interface. Alternatively, allow a data synchronization event to occur independently of the user interface.

Problem

One benefit of using mobile applications is having quick access to data. Responsiveness and waiting time are two key components to quick access of data in a mobile environment. A non-responsive or slow-to-respond application will quickly fall into disuse from frustration. Even if an application responds to user input quickly, however, a user will be frustrated if they must wait for significant periods of time for data to load. It is therefore important to ensure an application does not block when data synchronization occurs (or is attempted).

Applicability

The contexts eliciting the use of an asynchronous data synchronization mechanism can be considered from two perspectives: *uploading* and *downloading*. Uploading is defined as a transfer of data from the mobile application to a remote system. Downloading is defined as a transfer of data from the mobile application to the remote system. In both cases, the success or failure of the operation should be conveyed to the user, with appropriate error information if a failure occurs. The following are two applicability considerations for asynchronous data synchronization from each perspective outlined above.

Uploading

- The next state of the user interface or functionality of the application does not depend on the result of uploading data.
    - For example, in an application that manages many online social media services, a "status update" can be initiated for a number of different services. These uploading operations can happen simultaneously and the user can continue to interact with the application (or other applications on a mobile device that supports multi-tasking) while the transfers are occurring since the state of the application does not depend on the results of the transfers.

Downloading

- While fresh data is preferred, the application can at least partially fulfill its functionality with stale data. This constraint entirely depends on the nature of the data.
    - For example, a mobile application can manage the times, locations, and summaries of talks in a conference. When the user opens the application, it starts to download in the background any updates for the times and locations from a remote system. The nature of the data is such that fresh data (e.g., one of the talks has been rescheduled and this change is reflected on the remote system but not on the device) is preferred, but stale data can still be used (it is still useful to read the summaries and view the titles of the talks).
    - Moreover, an application could have a dataset that rarely changes, such as an application that shows the menu for a restaurant. It makes no sense to block the user interface for a synchronous download every time the application is opened. Rather, this download can be performed asynchronously from the user interface.

Solution

Initiate data transfer asynchronously via a trigger (such as a user action, a timer tick, an Android "intent", or a push notification in iOS) and performed asynchronously. A notifica-

tion mechanism (such as Android "toasts" or iOS "UIAlertViews") can be used to notify the user of the result of the data transfer. Likewise, a callback mechanism (such as an Android "intent" or a general callback function) can be used to notify the system when the data transfer completes.

Consequences

Benefits

- Availability of the application during data synchronization
  - The application is still usable during synchronization. Intuitively, the latest data will not be available, but for many situations, a user can take advantage of being able to interact with stale data while data synchronizes. In the case where the data is already up-to-date, user experience is not degraded by waiting on data to load.
- Background synchronization of data
  - If the system triggers a synchronization event (e.g., via an Android "intent", an iOS push notification, etc.) while the application is not in the foreground, the application can synchronize data in the background so it is conveniently up-to-date next time it opens.

Liabilities

- Inconsistencies stemming from concurrent access to a shared dataset
  - What happens when the *Data Access Object* (a pattern defined as an object that abstracts and encapsulates all access to the data source[iv], here with the data source being some flavor of local storage) is performing transactions on the dataset and the user is exposed to user interface components that are backed by elements of the same dataset? One solution is to perform the transfer asynchronously and perform any transactions synchronously. Thus, rather than blocking the user interface for the transfer and the transactions in the local data source, do the (slower) transfer in the background and only block the user interface for the (faster) transactions.

Examples/Known Uses

- Crash data for an application is collected on a remote system. Every time the application detects a crash, upload crash data asynchronously to a remote system.
- Usage statistics for an application are collected on a remote system once a week. A timer on the device ticks and the usage statistics are uploaded asynchronously.
- An application receives push notifications when a remote dataset is changed, and synchronizes the dataset on the device asynchronously.
- An application polls a remote system for updates on a given time interval. A timer on the device ticks and the data is synchronized asynchronously.
- Both Android and iOS have push notification systems built into the operating system (Android as of [check version number] and iOS as of [check version number]). They provide developers an online API to issue push notifications for their applications where the transactions are serviced through Google and Apple's infrastructures. By following the rules of the API, these notifications can serve as the trigger for asynchronous updates in an application.

- The Facebook and Twitter applications for both Android and iOS both allow a user to access the application while the data is being synchronized, thus using the pattern with opening the application as the trigger.

Synchronous Data Synchronization

Intent

Manage a data synchronization event synchronously; blocking the user interface while it occurs.

Problem

Some mobile application systems rely on datasets that must be constantly up-to-date or only have a very small window in which data is considered up-to-date. It would be useless for such applications to allow users to work with stale datasets. Similarly, some mobile application systems may rely on the response for an action from a remote system before another action can be performed. To prevent an application from entering unknown, non-functional states, a developer must ensure an application blocks until data synchronization is complete (successfully or unsuccessfully).

Applicability

- Fresh data is crucial to the functionality of the application.
- The application cannot advance to the next state without knowing the result of a prior synchronization action.

Solution

Initiate data transfer via a trigger (such as a user action, timer tick, etc.) and perform the transfer synchronously. The application only proceeds to the next state when a result (positive or negative) is reached.

Consequences

Benefits

- The state of the mobile application system can be managed more easily
  - Consider the difference between writing a simple multithreaded program and a simple single-threaded program. A single-threaded program executes its instructions in order and a state machine can easily be built from the code. A multi-threaded program can follow many paths of execution, thereby significantly increasing the number of states needed to build a state machine. By using the *Synchronous Data Synchronization* pattern, therefore, the extra states caused by asynchronous events can be eliminated.

Liabilities

- User interface thread blocking
  - As explained in the *Asynchronous Data Synchronization* pattern, user experience suffers when user interface thread blocking occurs. If the synchronization of data occurs on the same thread as the user interface, blocking for a network call or a lengthy database operation could occur and the application could become unrespon-

sive. It may be more practical to perform the transfer asynchronously on another thread, but treat it as a synchronous action in the application by using a loading dialog, thus not to block the user interface. This approach has the added advantage of allowing the option to cancel the synchronizing event.

Examples/Known Uses

- A user presses a "submit" button to submit a work order to a digital maintenance system. A loading dialog is shown and the order is transmitted synchronously (the user is made to wait until a result is reached).
- A user opens an application to view the current stock of a warehouse. A loading dialog is shown and the dataset is downloaded synchronously before the application continues.

4. Data Quantity Patterns

Data Quantity patterns address the question: "how much data should be transferred?" This question arises in both the design of a mobile application and the design of a remote system with which it interacts. Often, mobile application projects have constraints (such as network speed/bandwidth or capacity) both remotely and locally. Likewise, the capacity of local storage is often limited relative to the whole dataset needed for a computation.

Lazy Synchronization[v]

Intent

Synchronize data only as needed to optimize network bandwidth and storage space usage.

Problem

Network bandwidth and storage space are two vital concerns for mobile application design. Many applications that would be possible with non-mobile (e.g., laptop, desktop, server, cloud, etc.) computer storage capacity and a broadband connection are still possible as a mobile application, but specific attention must be paid to these issues. While a solution for larger systems would be to increase the network bandwidth or increase the storage capacity for each device, this is impractical on mobile devices, especially from the perspective of a mobile application developer.

Applicability

- The entire dataset is either too large to store on the device or it is impractical to transfer the entire dataset because much of it is not needed.
- The entire dataset is not needed in its entirety for an application to function; rather, parts can be transferred as needed.

Solution

Data is synchronized dynamically "on-demand" by triggers in the application, most typically using a variant of the *Virtual Proxy* pattern[vi]. A virtual proxy is an object with the same interface as the object used by the system that "intercepts" method calls, allowing initialization of fields only when they are needed. In *Lazy Synchronization* the *Virtual Proxy*

pattern can be combined with the *Data Access Object* pattern by creating a virtual proxy that handles the process of on-demand synchronization.

Consequences

Benefits

- Storage space is reduced
  - Rather than storing an entire dataset on the device, the system retrieves data as it is needed and used, thereby allowing mobile applications to use far larger datasets than can be stored on the device.
- Datasets can be synchronized at various levels of granularity
  - By synchronizing data only as needed, portions of the dataset can be used and modified at a fine-grained level in parts, even if the whole dataset cannot be loaded at that granularity. This approach operates similarly to the fine-grained control of task scheduling in an operating system, without needing to know the same level of detail about memory management or hardware addressing.

Liabilities

- Network connectivity
  - More network calls are involved, and if connectivity changes during operation, lazy loading can fail and render the application useless.
- Network bandwidth/speed
  - If the on-demand network operations used by *Lazy Synchronization* run for a significant amount of time, user interface responsiveness and waiting time can cause user experience to degrade quickly.

Examples/Known Uses

- In an application for a library, a user enters search terms for a particular topic and a data transfer is invoked. The resulting title, author, and availability for each book is returned and displayed. The user chooses one and another data transfer is invoked. The remaining details of the particular item are returned and displayed.
- The Google Maps application works by displaying a relevant map at a certain level of granularity and downloading new tiles as needed when the map is zoomed in or out, thus eliminating the need for storing the entire map at every level of detail on the device, but requiring network connectivity to be used.

Eager Synchronization[vii]

Intent

Synchronize data before it is needed so the application has better response or loading time.

Problem

While wireless connectivity is widespread and highly available in many places, there are times when network connectivity is not available or not desired, but an application must still function. *Lazy Synchronization* works by loading data on demand, much like a mobile website, and will not work in such a scenario. Moreover, the possibility of low network bandwidth/speed could be an issue for application responsiveness.

Applicability

- The entire dataset should be synchronized between the device and the remote system during a synchronization event.

Solution

Store the entire dataset on the device and keep it wholly (as opposed to partially via *Lazy Acquisition*) synchronized.

Consequences

Benefits

- Reliance on network connectivity is decreased
  - If the network is not available, the data used by the application cannot be synchronized, but the application can still be used if stale data allows certain functionality.

Liabilities

- Device storage usage is increased
  - The device must have all of the data it needs stored locally, so the dataset must be able to fit on the device.

Examples/Known Uses

- An application should be completely usable offline.
- An application has a relatively small dataset that will not change often/significantly.
- An application suspects that its dataset has become corrupt and needs a new copy of the entire dataset.

5. Reconciliation Patterns

Reconciliation patterns address the problem of set reconciliation: "how can we synchronize between sets of data such that the amount of data transmitted is minimized?" As discussed above, network bandwidth/speed is often a concern of mobile applications, so developers should write their applications to minimize resources to accomplish the task of synchronizing data. Some types of data (such as records with a timestamp to keep track of changes) lend themselves to more efficient methods of reconciliation than others, while other types of data (such as files containing compound documents) have less efficient methods of reconciliation without becoming overly complex.

Full Reconciliation

Intent

On a synchronization event, the entire dataset is transferred between the mobile device and the remote system.

Problem

Some types of data (such as static files or datasets that change entirely after a synchronization event like a "Message of the Day") either cannot use or will not benefit from a complex reconciliation scheme. Moreover, some situations (such as a corrupt dataset) are more eas-

ily handled by a simple reconciliation scheme since a more complex scheme that uses fewer resources may take more time and effort to develop.

Applicability

- The dataset of an application is small enough that it can be downloaded/uploaded in one piece.
- A complex data reconciliation scheme is not needed or provides little benefit over a simple one.

Solution

Reconcile data between a device and a remote system by transferring the entire contents of one to the other and making any appropriate changes when data is received.

Consequences

Benefits

- It is the simplest solution
  - At the most basic level, either the device or the remote system sends all of its data to the other one, who replaces his dataset with the received one. Both the device and the remote system, then, are guaranteed to have the same data.

Liabilities

- Redundancy of data being sent
  - If the data only changes partially or not at all, sending data that will not be changed wastes bandwidth.

Examples/Known Uses

- An application detects an error in its dataset. Rather than using a complex reconciliation scheme, it uses *Full Reconciliation* to easily replace the faulty dataset.
- An application displays the top ten news articles for a newspaper issued daily (so that the top ten news articles do not change). The dataset changes every time the application is updated, so it uses *Full Reconciliation*.

Timestamp Reconciliation

Intent

On a synchronization event, only the parts of the dataset changed since the last synchronization are transferred between the mobile device and the remote system using a last-changed timestamp.

Problem

With the issue of network speed and bandwidth on mobile devices, the amount of data transferred to reconcile datasets between a device and a remote system should be minimized. *Full Reconciliation* wastes too many resources and the dataset does not fit the more strict requirements for *Mathematical Reconciliation*. It is still imperative to synchronize data, but another method is needed to minimize data transfer.

Applicability

- The dataset of an application can be downloaded/uploaded in specific pieces.
- The pieces of data must have a field to store a timestamp denoting the last time it was modified.

Solution

A timestamp provided by the remote system from the last successful update is bundled with a request for changed data. The remote system returns only data that has been added or changed after that timestamp. For submitting data, the device only submits data that has been added or changed since the last successful submission.

Consequences

Benefits

- Lower bandwidth utilization than *Full Reconciliation*

Liabilities

- Careful attention must be given to the source of timestamps
  - It is important to keep the source of timestamps consistent, as synchronization can become inconsistent if different timestamps are used. It is common to use the remote timestamp for any downloaded data and the device timestamp for any uploaded data.
- It is not immediately apparent how to handle deletion of data
  - A timestamp will not do any good if data is deleted on a remote system and a device tries Timestamp Reconciliation, as the deleted data does not exist for a timestamp comparison. A common solution to this problem is to use a Boolean filed on each piece of data to signify whether or not it has been deleted.

Examples/Known Uses

- An application stores routes for public transportation. It uses *Timestamp Reconciliation* to only transfer new, changed, or removed routes when it updates.
- Twitter and Facebook's public APIs each offer developers the ability to retrieve the latest posts using a "since" value, thus supporting *Timestamp Reconciliation*.

Mathematical Reconciliation

Intent

On a synchronization event, only the parts of the dataset changed since the last synchronization are transferred between the mobile device and the remote system using a mathematical method.

Problem

Network bandwidth and speed are concerns, so *Full Reconciliation* cannot be used to reconcile a dataset between a device and a remote system. The structure of the dataset also may not be able to keep track of changes efficiently using timestamps, or alternatively, a mathematical method can reconcile changes more efficiently than timestamps can.

Applicability

- The dataset of an application can be synchronized using a mathematical method.
- The application should use the absolute minimum bandwidth required to synchronize datasets, and time can be spend developing a complex mathematical method.

Solution

A mathematical method or algorithm decides what is transferred between a device and a remote system to synchronize a dataset.

Consequences

Benefits

- This method potentially uses the least bandwidth
  - For synchronizing something such as a very large binary file where only a few bits are changed, *Timestamp Reconciliation* and *Full Reconciliation* perform the same actions. With a mathematical method, such as dividing the file into blocks, computing checksums, and comparing checksums before transferring data, bandwidth used can be reduced.

Liabilities

- Mathematical methods are often highly context-dependent
  - An important paradigm of programming in general, code reuse, is unlikely to be applicable here, as the mathematical method of reconciliation will be different for different types of data.
- Mathematical methods often require more time to develop
  - Most mathematical methods will be more complex than Full Reconciliation or *Timestamp Reconciliation*, as they will at least have more steps involved in the process of reconciliation.

Examples/Known Uses

- An application synchronizes an image taken from a webcam periodically between a device and a remote system. The remote system stores the previous image and uses the "sum of absolute differences" method to determine whether to send a whole new frame or just the difference between the two.

---

[i] http://developer.android.com/design/index.html

[ii] http://developer.apple.com/library/ios/#documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/Introduction/Introduction.html

[iii] Frank Buschmann, Régine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stahl 1996. *Pattern-Oriented Software Architecture— A System of Patterns*, New York, NY: John Wiley and Sons, Inc.

[iv] Deepak Alur, Dan Malks, and John Crupi. 2001. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

[v] M. Kircher, Lazy Acquisition Pattern, European Pattern Language of Programs conference, Kloster Irsee, Germany, July 5-8, 2001, http://www.cs.wustl.edu/~mk1/LazyAcquisition.pdf

[vi] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides: Design Patterns – Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995

[vii] M. Kircher, Eager Acquisition Pattern, submitted to European Pattern Language of Programs conference, Kloster Irsee, Germany, July 4-7, 2002, http://kircher-schwanninger.de/michael/publications/EagerAcquisition.pdf