# 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 mobile device applications. 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. This gap 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 provides an initial step in a larger effort on a pattern language for mobile application development; it focuses solely 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 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 or misleading 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 other pattern catalogs and pattern languages for applicable patterns, and documenting our experiences developing mobile applications. Open-source examples and insights into the platforms and frameworks are cited explicitly, and some examples of these patterns in popular consumer applications are also mentioned.

## 2. Pattern Format

The patterns presented here are documented using a variant of the Gang of Four and POSA pattern forms to reflect the types of patterns covered in this document. The following are the sections in our modified 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

### Visual Explanation

A diagram or graphical representation explaining the pattern and a description of the interaction between the different elements of the 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—yet often overlooked—in mobile application design, 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.

The contexts eliciting the use of data synchronization mechanism patterns can be considered from two perspectives: *uploading* and *downloading*. Uploading is the transfer of data from the mobile application to a remote system, whereas downloading is the 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 either directly (with a notification or dialog) or indirectly (in a log or separate part of the application), with appropriate error information if a failure occurs.

Data synchronization mechanism patterns are often architectural patterns. 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."[iii] The following data synchronization mechanism 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 elements that determine how quickly data can be accessed in a mobile environment. A non-responsive or slow-to-respond application yields a poor user experience. 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 following are applicability considerations for asynchronous data synchronization from the uploading and downloading perspectives outlined at the beginning of Section 3:

#### *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.

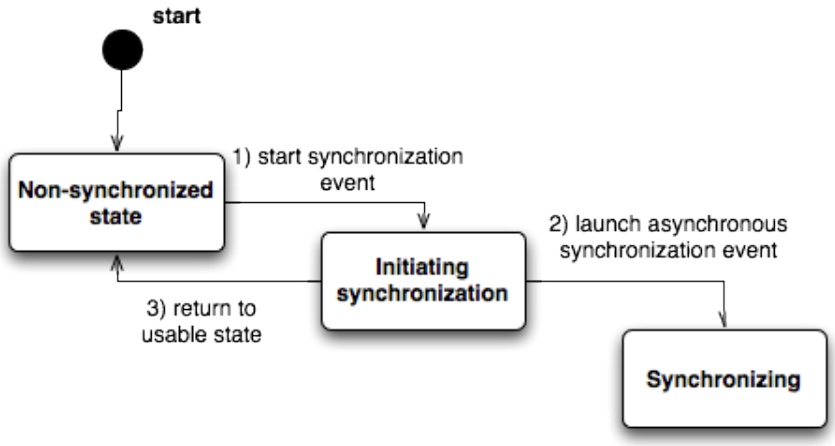
#### *Downloading*

- 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 pre-

ferred, 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 (such as an app that shows the menu for a restaurant) could have a dataset that rarely changes. 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.

In some systems it may not be possible or desired to run a data transfer operation simultaneously with the user experience due to limited resources. This limitation was especially prevalent with older mobile devices, such as those running early versions of Symbian, which led to the development of the Active Object concurrency model within the Symbian OS.[iv]

### Visual Explanation




The *Asynchronous Data Synchronization* pattern is a mechanism pattern, thus it may be best visualized as a series of states. When an application is "usable" (meaning the user can interact with the application) a synchronization event can be started. Rather than executing the synchronization event sequentially, however, the event is launched asynchronously and the application returns to a usable state immediately.

### Solution

Initiate data transfer asynchronously via a trigger (such as a user action, a timer tick, an Android "intent[v]", or a push notification in iOS) and perform the transfer asynchronously. A notification mechanism (such as Android "toasts[vi]" or iOS "UIAlertViews[vii]") can be used to inform 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 inform 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 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" or an iOS push notification) 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[viii], with the data source being some flavor of local storage in this context) 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. Rather than blocking the user interface for the transfer and the transactions in the local data source, therefore, do the (slower) transfer in the background and only block the user interface for the (faster) transactions.
- Data quantity charges unknown to the user
    - What if an application is asynchronously transferring a large quantity of data without the user's knowledge via a pay-by-quantity radio connection? This issue is increasingly important, as many people using smartphones purchase plans limiting their data transfer to 2 or 4 GB per month, with overage charges after they reach their limit. A common solution (e.g., used by Apple's FaceTime app at the time this paper was written) is to disable application usage unless the user has Wi-Fi connectivity.
- Data quantity can congest the network
    - What if many users are using high-bandwidth applications, such as videoconferencing applications or file-sharing applications in the same cell? This issue drives the development of next-generation networks to handle data usage by smartphones and smartphone applications. For instance, AT&T suffered from network congestion after the release of the iPhone.[ix] The solution noted above regarding application restrictions on cell networks also applies to this liability.

### 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 these APIs, notifications can serve as the trigger for asynchronous updates in an application.
- The Facebook and Twitter applications for both Android and iOS allow users to access the applications while the data is being synchronized, thereby applying the pattern by using the opening of the application as the trigger.

## Synchronous Data Synchronization

### Intent

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

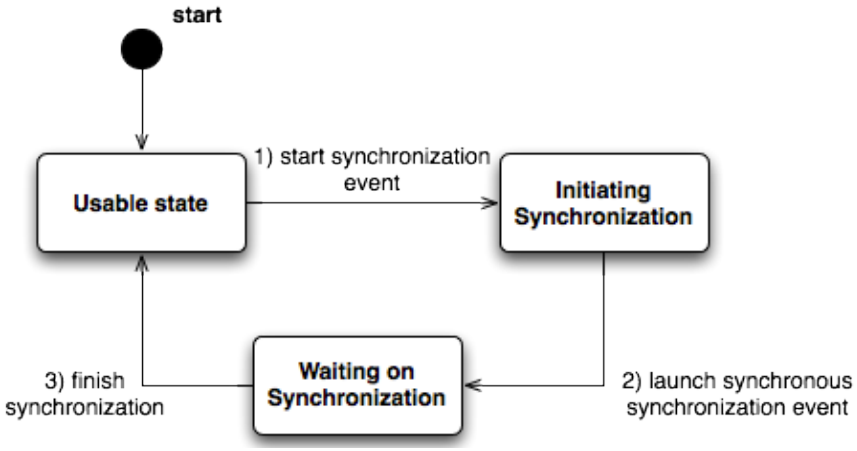
### Problem

Some mobile application systems rely on datasets that must be accurate in real time or have certain time constraints limiting usability of the data. Moreover, allowing users to work with stale datasets may hinder or eliminate productivity for these applications. 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.
  - As mentioned in the introduction, an application that allows users to manage stock portfolios via online services must have fresh data for sensible functionality. If such an application employed an asynchronous synchronization of data between online services and devices, users may unknowingly be viewing stale data.
- The application cannot advance to the next state without knowing the result of a prior synchronization action.
  - To extend the example outlined above, *Synchronous Data Synchronization* may be a necessary pattern from the perspective of an online service. If users place orders, it would be prudent to confirm the success or failure of purchases synchronously and make users wait, rather than submitting orders asynchronously and allowing users to submit more orders without knowing the result of previous ones. The transmissions could arrive out-of-order or be lost, leading to many new errors that could v arise.

## Diagram/Structure




The *Synchronous Data Synchronization* pattern is a mechanism pattern as well, thus we will visualize it as a series of states. When an application is in a usable state, a synchronization event can be started. The event is launched synchronously and executed in a sequential fashion, leaving the application in a waiting state until the synchronization is finished, whereupon it returns to the usable state.

## 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 description, 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 may become unresponsive. 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, thereby not blocking 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.
- A user opens a banking application and is prompted to enter his or her username and password. The user presses submit and the credentials are synchronously validated by the remote system before the application becomes usable.
- The Spotify application on BlackBerry, iPhone, and Android stores the date up to which the current account is paid, and on/after that day, synchronously checks the account status of the current account to update that date or alert the user that his or her account has expired.

## 4. Data Storage and Availability Patterns

Data Storage and Availability patterns address the questions: "how much data should be stored?" and "how much data should be available without further transfer of data?" This question arises in both the design of mobile applications and the design of remote systems with which they interact. 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.

### Partial Storage[x]

#### Intent

Synchronize and store 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 requiring such resources were originally developed on non-mobile platforms (e.g., laptop, desktop, server, cloud, etc.) where the resources are more abundant. These applications can still be developed as mobile platforms, but specific attention must be paid to conservation/utilization of these resources. 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 mobile application developers who must operate within the constraints of their underlying mobile platforms.

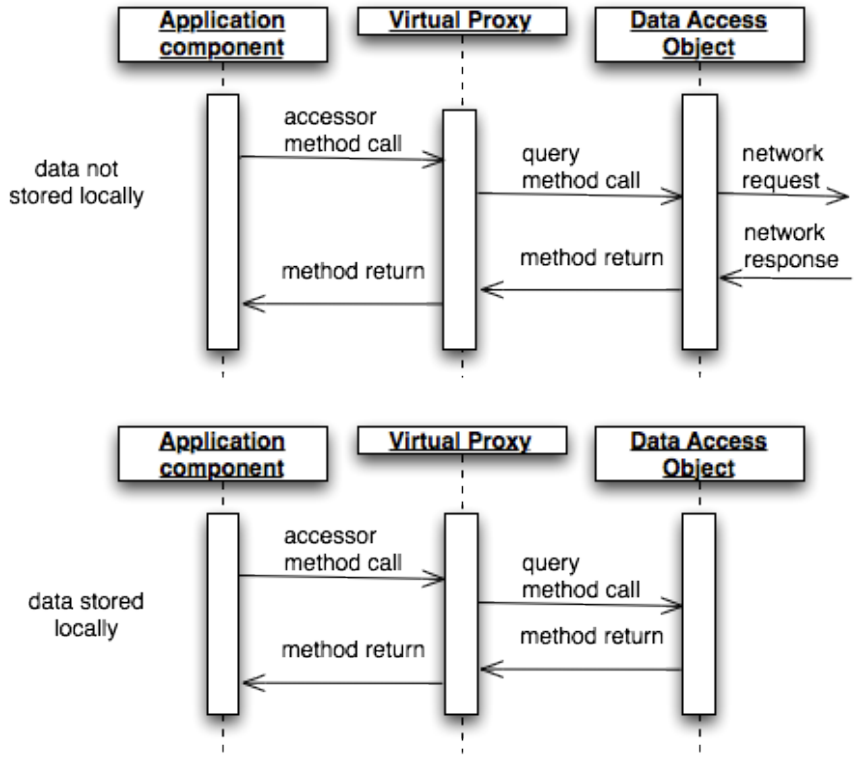
#### Applicability

- 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.
  - An example of this situation is an online bookstore accessed by an e-reader or other mobile device. With memory and battery-efficiency constraints, it is impractical for

e-readers to synchronize and store the entire inventory of an online bookstore. It is therefore significantly more practical to synchronize data as needed.

- The entire dataset is not needed in its entirety for an application to function; rather, parts can be transferred as needed.
  - A mobile application could be used to access an existing geographic information system to manage different systems on a university campus, such as the location of fire extinguishers or available telecom lines in certain buildings. A person responsible for inspecting the fire extinguishers on campus has no need for information about telecom lines or air ducts, so it would be impractical to synchronize all of the available data to his or her particular device.

### Visual Explanation




In the diagram, there are two sequences shown for two separate cases that encompass *Partial Storage*. First, the sequence is shown for when data needs to be accessed and is not stored on the device; second, the sequence is shown for when data is already stored on the device. This diagram uses other patterns described below in the solution section to help visualize the process using existing common patterns.

### Solution

Rather than using pre-fetching, data is synchronized dynamically "on-demand" by triggers in the application, most typically using a variant of the *Virtual Proxy* pattern[xi]. 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. *Partial Storage* can be realized using the *Virtual Proxy* and *Data Access Object*[viii] patterns by creating a vir-

tual proxy that handles the process of on-demand synchronization when data is not in local storage. The Virtual Proxy provides the same interface as the object being accessed by the component and queries the Data Access Object for the data to populate its fields. The Data Access Object determines the local availability of the data and makes network calls if needed. This design minimizes storage needs and does not incur the bandwidth usage of synchronizing the entire dataset.

## 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 *Partial Storage* 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 digital library, a user enters search terms for a particular topic and a data transfer is invoked. The *Virtual Proxy* for the *Data Access Object* is invoked first, which uses the methods of the *Data Access Object* to determine whether or not to invoke a data transfer. Regardless of whether a data transfer occurs or the *Data Access Object* retrieves the data from local storage, the resulting title, author, and availability for each book is returned by the *Virtual Proxy* and displayed. The user chooses one and another such process occurs. 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. This design eliminates the need to store the entire map at every level of detail on the device, but requires network connectivity.

## Complete Storage[xii]

### Intent

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

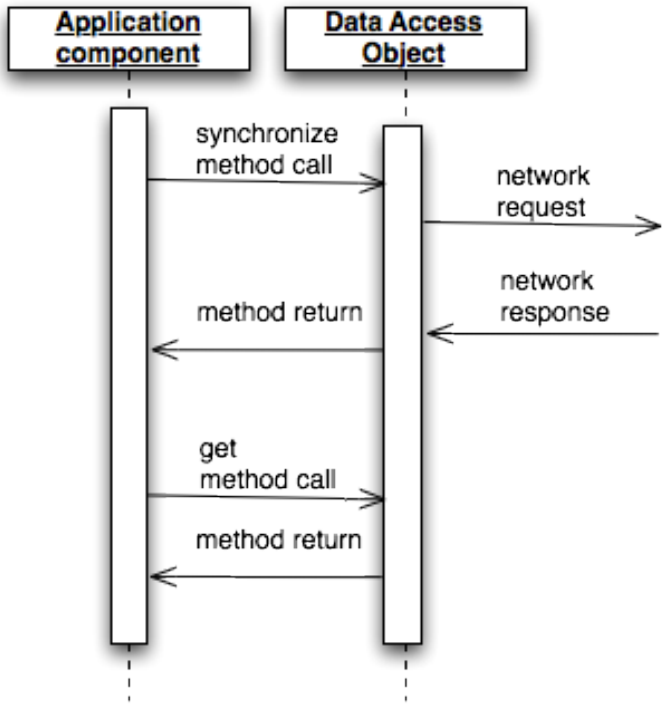
### Problem

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. *Partial Storage* works by loading data on demand, much like a mobile web-site, and will not work in such a scenario. Moreover, the possibility of low network bandwidth can be an issue with respect to application responsiveness.

### Applicability

- The entire dataset should be synchronized between the device and the remote system during a synchronization event.
  - Ideally an application meant to disseminate up-to-date emergency information (such as first-aid stations, food and water stations, or shelters in case of a natural disaster) would have the information stored in the device, since connectivity may be unavailable in such a situation.

### Visual Explanation




This sequence diagram gives a simplified version of *Complete Storage*. It draws a distinction between two types of actions: synchronization and a "get" action. The synchronization action causes a network request and returns data to be stored on the device. In *Complete*

*Storage*, it is this action that synchronizes all of the data. All "get" actions result in local data being returned.

### Solution

Store the entire dataset on the device and keep it wholly (as opposed to partially via *Partial Storage*) synchronized when connectivity is optimal (i.e., when using Wi-Fi where bandwidth is high).

### 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
  - An application must have all of the data it needs stored locally, so the dataset must be able to fit on the device.
- Bandwidth usage is increased
  - Rather than actively synchronizing data as needed, an application transfers all data at once, including possibly not-needed data.

### Examples/Known Uses

- An application should be completely usable offline, such as the emergency preparedness application mentioned in the Applicability section.
- 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.
- Dropbox Mobile (which is a popular app for storing and synchronizing personal files on private cloud storage and between devices) uses this pattern to allow a user to access files when bandwidth is low or a connection is not available. It further optimizes the pattern's liabilities by allowing users to choose which files and folders to synchronize eagerly.

## 5. Data Transfer Patterns

Data Transfer patterns address the problem of transfer quantity in set reconciliation: "how can we synchronize between sets of data such that the amount of data transmitted is minimized?" As discussed above, network bandwidth 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 Transfer

### Intent

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

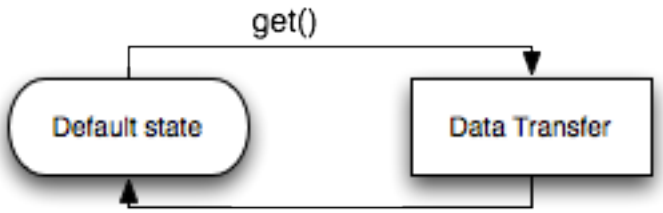
### Problem

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 easily 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.
  - An analytics application that monitors the time a user spends running the application may only need to send a single time value to an online service, and then reset its counter upon acknowledgement of receipt. The dataset involved in synchronization is a single number, thus only a simple reconciliation technique is needed.
- A complex data reconciliation scheme is not needed or provides little benefit over a simple one.
  - An application that shows a different offer, coupon, or deal to a user every day would not benefit from a complex data reconciliation scheme since the data is only being transferred one way (from the service to the application) and the new data has no relationship with the old data.

### Visual Representation




This flow chart shows the simplicity of a *Full Transfer*. The application simply initiates a transfer to obtain the entire dataset.

### 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 are then assured of having 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 Transfer* 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 Transfer*.

### Timestamp Transfer

#### 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.

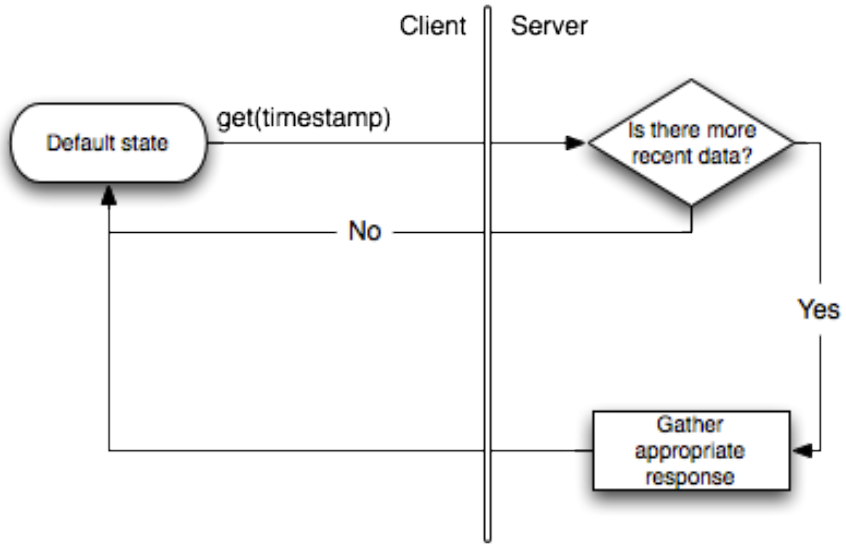
#### Problem

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 Transfer* wastes too many resources and the dataset does not fit the more strict requirements for *Mathematical Transfer* (described below). 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.
  - An application with versioned data that is stored in an online service could choose which specific files or pieces of data to upload or download based on a comparison of the last-updated timestamp on the device and the created/modified timestamp on the file or data.
- The pieces of data must have a field to store a timestamp denoting the last time it was modified.
  - If an application and its corresponding online service synchronize tables of information, each table should have a timestamp column to facilitate the comparisons needed for a *Timestamp Transfer*.

### Visual Representation




This flow chart shows the increased logic of a *Timestamp Transfer*. The client initiates a request and attaches a timestamp to the request, which is processed by the server to determine whether or not to return any data.

### 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 Transfer*.
  - By comparing a "last-update" timestamp submitted by an application to a service, only the data created or modified since that timestamp must be sent back to the device. Likewise, if an application stores the timestamp of the last time it uploaded data, it can then decide on the next synchronization event to only upload data the target has not yet seen.

#### *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 may not be 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 Transfer*, as the deleted data does not exist for a timestamp com-

parison. A common solution to this problem is to add a boolean field 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 Transfer* 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, thereby supporting *Timestamp Transfer*.

### Mathematical Transfer

#### 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.

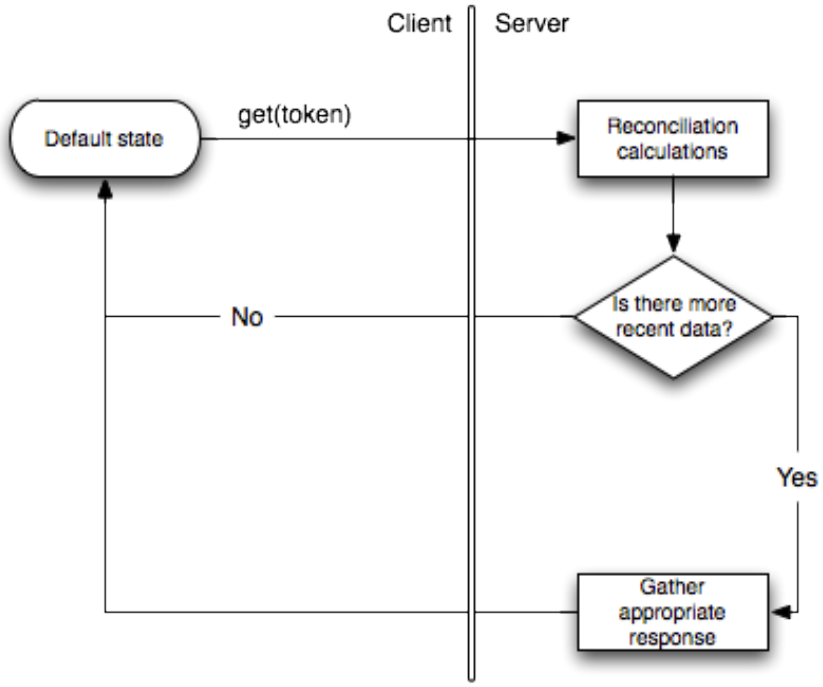
#### Problem

Network bandwidth and speed are concerns, so *Full Transfer* 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 *Timestamp Transfer*, or alternatively, a mathematical method can reconcile changes more easily or efficiently than timestamps can.

#### Applicability

- The dataset of an application can be synchronized using a mathematical method.
  - For example, checksums of large pieces of data could be compared or an algorithm could be used (such as the Characteristic Polynomial Interpolation-based Synchronization developed by Trachtenberg et al.[xiii]).
- The application should use the absolute minimum bandwidth required to synchronize datasets, and time can be spent developing a complex mathematical method.
  - In limited-bandwidth situations (such as military or aeronautical applications that synchronize positional data between multiple real-time or low-power devices) unnecessary bandwidth usage slows down or prohibits application functionality.

## Visual Representation




This flow chart is similar to the one in *Timestamp Transfer* with the addition of a process to calculate the differences in the data sets. In *Timestamp Transfer*, this is simply a comparison, but in *Mathematical Transfer*, this could be a more significant calculation that should be considered a separate process.

## 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 compared with the *Timestamp Transfer* and *Full Transfer* patterns.
  - For synchronizing something such as a very large binary file where only a few bits are changed, *Timestamp Transfer* and *Full Transfer* 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.
  - Code reuse may not be applicable here since 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 Transfer* or *Timestamp Transfer* since they will at least have more steps involved in the process of reconciliation.

### Examples/Known Uses

- Internet video conferencing applications (such as Skype and Facetime) on mobile devices use video compression techniques based on human-perceivable differences in the overall image[xiv]. The remote system stores the previous frame and uses the "sum of absolute differences" or the "sum of squared differences" methods to determine the optimal encoding scheme for the new frame.

---

[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] Aapo Haapanen. "Active Objects in Symbian OS." MS thesis University of Tampere, 2008, http://www.cs.uta.fi/research/theses/masters/Haapanen_Aapo.pdf

[v] http://developer.android.com/reference/android/content/Intent.html

[vi] http://developer.android.com/guide/topics/ui/notifiers/toasts.html

[vii] http://developer.apple.com/library/ios/#DOCUMENTATION/UIKit/Reference/UIAlertView_Class/UIAlertView/UIAlertView.html

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

[ix] Jenna Worthham, "Customers angered as iPhones overload AT&T." New York Times, 2 Sept. 2009, http://www.distributedworkplace.com/DW/News/2009%20News%20July%20-%20December/Customers%20Angered%20as%20iPhones%20Overload%20ATT.doc

[x] 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

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

[xii] 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

[xiii] Ari Trachtenberg; D. Starobinski and S. Agarwal. "Fast PDA Synchronization Using Characteristic Polynomial Interpolation". IEEE INFOCOM 2002. http://people.bu.edu/staro/infocom02pda.pdf

[xiv] Zhao, David. Video Coding. U.S. Patent 8,213,506, filed Sep. 8, 2009, issued Jul. 3, 2012.