Supporting Disconnected Operation in Wireless Enterprise Applications

A Java BluePrints for Wireless White Paper

PRELIMINARY

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A. 650-960-1300

June 26, 2003 (draft)

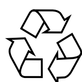
Copyright © 2002-2003 Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, California 94303 U.S.A. All rights reserved.

Sun Microsystems, Inc. has intellectual property rights relating to implementations of the technology described in this publication. In particular, and without limitation, these intellectual property rights may include one or more U.S. patents, foreign patents, or pending applications.

Sun, Sun Microsystems, the Sun logo, Java, the Java Coffee Cup logo, J2ME, J2SE, J2EE, JDBC, Enterprise JavaBeans, EJB, JavaServer Pages, and JSP are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.



Supporting Disconnected Operation in Wireless Enterprise Applications

by Thierry Violleau and Ray Ortigas

This paper describes guidelines for designing wireless Java clients that can operate effectively when disconnected from the network and the enterprise.

Several aspects of networked wireless applications motivate the need to design for disconnected operation. Mobile users may wish to use the network sparingly due to technical factors such as low quality of service (compared to wireline networks). Additionally, many mobile users find themselves working in contexts with frequent interruptions, such as incoming calls or distractions from the surrounding environment.

The design of effective disconnected wireless clients involves strategies such as batching or coalescing client/server interactions, data caching, and data synchronization. However, these strategies are highly constrained by the limited memory, data storage, processing power, and battery life of mobile devices. They are also constrained by poor network quality, because a wireless application needs the network to facilitate disconnected operation in the first place (for example, to download relevant data for a disconnected session). The design must also address the duality of modes of operation (connected and disconnected) which must be kept similar to preserve a consistent user experience.

Another factor to consider is that users often access enterprise applications from a variety of devices, including desktops and wireless devices, depending on their context or location. For example, users typically access a service with their

desktop when at home or in the office, but may turn to a wireless device when travelling or commuting. Enterprise applications must provide multiple application clients and target these clients to different devices. Given this assumption, enterprise applications have to address different operation modalities (offline and online modes), plus how to maintain a coherent user work experience across different devices.

This paper is one of several Java BluePrints for Wireless papers, and it limits itself to those issues that arise for wireless clients and applications that operate offline from the network. It leaves the basics of designing Java wireless enterprise applications, such as communication, session management, security, and so forth, and localization and internationalization issues, to other papers.

Note that some of the concepts and issues exposed in this paper are addressed by emerging specifications (such as SyncML) or commercial products (such as client-side database products). While this paper focuses on solutions that can be directly implemented on top of MIDP 1.0, commercial products may be considered as part of alternative solutions.

1 Background

This paper presents guidelines for end-to-end Java applications. On the client side, they use the Mobile Information Device Profile (MIDP), as defined by the Java 2 Platform, Micro Edition (J2ME). On the server side, they use the Java 2 Platform, Enterprise Edition (J2EE). These applications are typically architected. (See Figure 1.)

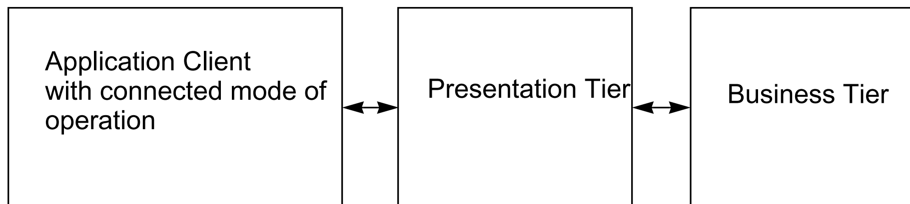


Figure 1 Typical Three Tier Architecture

This paper presents guidelines for designing an offline or disconnected mode of operation into such clients.

1.1 Standalone Wireless Applications Versus Wireless Application Clients

MIDP applications run the gamut from thin wireless clients of end-to-end applications to standalone wireless applications, as shown in Figure 2.

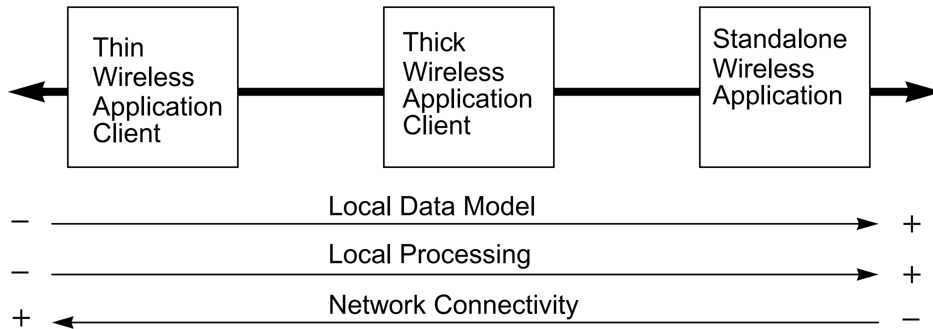


Figure 2 The MIDP Application Spectrum

Generally, applications fall within this spectrum according to the following criteria:

- The network connectivity required
- The local processing capability required
- The part of the application data model that needs to be located locally

Standalone wireless applications are at the self-sufficient end of the spectrum. They perform most, if not all, of the application data processing. They may occasionally connect to the network to synchronize or archive data with the enterprise's repository, usually at the request of the user. Examples of standalone wireless applications include calculator, notepad, and calendar applications.

Thin wireless application clients are at the other end of the spectrum. They are typically generic browsers or browsing applications. Other than for presentation, thin wireless clients perform limited data processing, if any. They mainly work connected to the network, although some thin wireless clients may provide some caching and limited disconnected functions. Examples of thin wireless clients include HTML, WML, and XHTML browsers.

Thick application clients fall somewhere between standalone wireless applications and thin application clients in the MIDP application spectrum. Thick application clients are generally dedicated programs that work closely with their server-side counterparts. Thick wireless clients do a large part of their work connected to the network. However, they also can perform a range of disconnected functions depending on how much of the data model they can locally manage. Often with thick wireless clients, processing is distributed more or less evenly between the client and the server. Client-side processing may include sophisticated data presentation, such as sorting and filtering, as well as data editing. Examples of thick wireless clients include expense reporting and inventory management applications.

A general concern is the partitioning of processing between the client and the server. A client that can operate while disconnected requires a more complex design than a client that operates solely connected. Although disconnected operation reduces network requirements, it does incur costs in memory, persistent storage, and processor usage on the client. Thus, it's important to provide the proper balance between client-side and server-side processing, and at the same time keep the client-side resource requirements within device capabilities. Balancing processing between client and server is a fundamental design decision for client/server applications. This paper addresses the issue only from the perspective of designing for disconnected operation.

Thin application clients and lower-end thick application clients (that is, thick wireless clients whose functionality is closer to that of thin wireless clients) may not require a domain-specific client-side data model and instead may only require simple data exchanges with the application service. However, thick wireless clients—especially those able to operate in a disconnected mode—require both a richer client-side data model and the ability to do data exchanges with the application service. Simple data exchange techniques, such as those implemented by thin application clients and lower-end thick application clients, may not be able to support a rich (hierarchical and type rich) data model required for disconnected operation.

The Java Smart Ticket sample application, from the Java BluePrints for Wireless program, uses a thick application client in the context of a sophisticated mobile commerce (or *m-commerce*) scenario. The sample application is used throughout this paper to illustrate various concepts and guidelines.

1.2 Why Operate in a Disconnected Mode

It may be necessary for clients to operate offline or disconnected from the network for two reasons: the network quality may be poor (compared to wireline networks) and the typical use case of mobile wireless devices may justify it. Often, a combination of these two reasons makes offline operations attractive.

A low quality of network service may be due to one or more of the following reasons:

- High network latency—The extra time required for a packet to go from one point to another, because of router processing and so forth, is a disadvantage for "chatty" protocols.
- Limited network bandwidth—Limited bandwidth restricts data transfer rates and is a factor when transferring large amounts of data.
- Intermittent connectivity—Wireless devices generally do not maintain a dedicated connection to the network, but connect only intermittently due to imperfect network coverage and poor network reliability (such as when the user enters an area where the network cannot be reached).

It's easy to see how a low quality of network service may cause an online user to experience interruptions and frustrating delays when using a wireless application.

Provider pricing schemes may also affect network usage. Schemes that are not flat-rate, but rather charge users per packet, for example, mitigate against "chatty" client-server communications. In fact, users may see such schemes as additional incentives for disconnected operation.

Typical use cases for mobile wireless devices also lend themselves to operating in a disconnected mode. Most mobile device users function in what can best be described as a "highly interruptible" mode. That is, the mobile device work session may be interrupted at any time and must be resumed quickly. Consider, for example, the average cell phone user. The user may be interrupted by an incoming phone call. The user may be using the device in a context rife with interruptions, such as a restaurant, theater, public transportation, and so forth. The user may also be taking advantage of sporadic moments of free time to incrementally complete a task.

2 Disconnected Operation Use Case Example

To illustrate these scenarios where a disconnected mode of operation is useful, this paper examines the Java Smart Ticket sample application. This movie ticketing application addresses a typical use case which requires a wireless client to operate in connected and disconnected modes.

Users are regular moviegoers, usually visiting the cinema weekly or perhaps more often. They tend to go to their neighborhood theater, but may sometimes visit other locations.

With the movie ticketing application, users can:

- Buy tickets ahead of time. For example, users may buy tickets on their way to the theater and avoid standing in line at the ticket counter.
- Download up-to-date movie schedules (these schedules change weekly) and browse various theater schedules offline.
- Personalize the application to their own needs.
- Improve their buying experience by contributing to and browsing a collaborative compilation of movie ratings.
- Interact with the application using different clients, such as desktop workstations or wireless devices.

The movie ticketing application provides both online and offline modes. As a point of comparison, Figure 3 illustrates the use cases for an application client providing only an online mode[1].

[1]. Note that some of the features described from here on are not implemented yet in the current release of the Smart Ticket sample application.

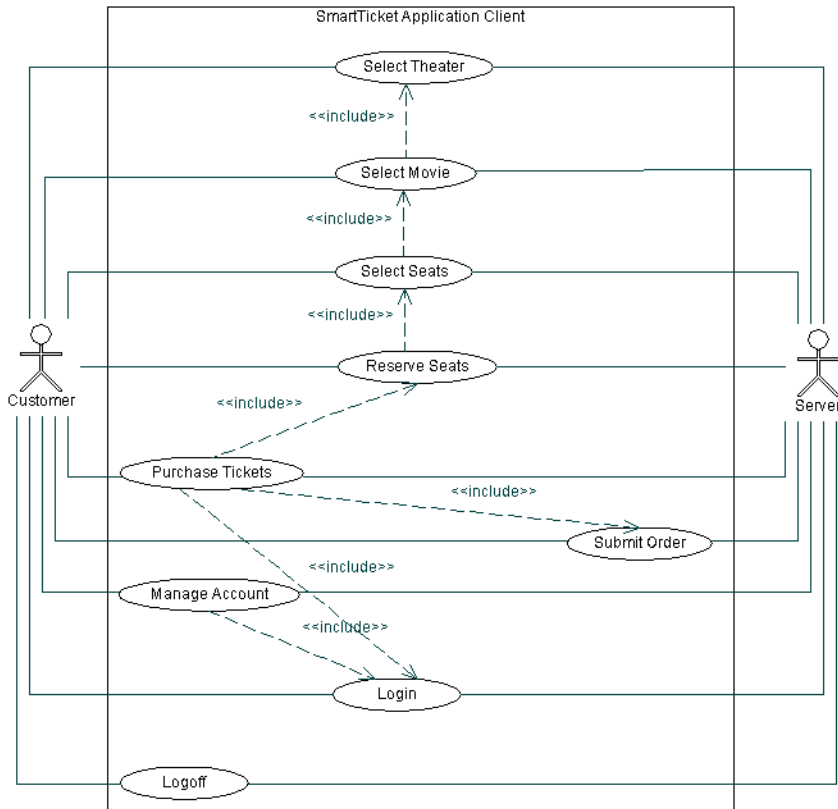


Figure 3 Use Case Diagram of Movie-Ticketing Application Client With Only Online Mode

When the application is in online mode, users can browse complete lists of theaters and movies by zip code. Or, users may browse movies that have been compiled into a "must-see" list based on personal criteria such as critic rating, genre, and so forth. While still connected, users can select a particular movie in a particular theater, see its show times, reserve seats according to that theater's seating plan, and purchase tickets.

In addition, while operating online, users can download complete schedules for selected theaters, or for customized lists of movies, into the wireless device. The listings may be refreshed on demand or automatically, such as when a new schedule is available.

The offline use case is somewhat different from the online use case. While offline, users can locally browse downloaded movie schedules directly with no authentication procedures. Such operations are performed quickly, as there are no network delays. When users select a particular movie, they connect to the server and, after logging in, proceed with reserving seats and purchasing tickets.

Although users are kept aware of whether they are online or offline, the user experiences for browsing movies and theaters in both modes of operation are similar. To accelerate the browsing experience, the application tracks movies the user has seen, and provides the ability to locally filter out these movies from the list. Offline, users can also rate movies they have seen and choose to upload these ratings when online.

The movie ticketing application also gives users connecting from a desktop environment the equivalent of a Web portal application. Not only can these users reserve seats and purchase tickets, they can also personalize the application and rate movies. Because users can rate movies either online through the Web portal client or offline using the MIDP client, the movie ticketing application allows users to synchronize the data on the server and the MIDP client, as shown in Figure 4.

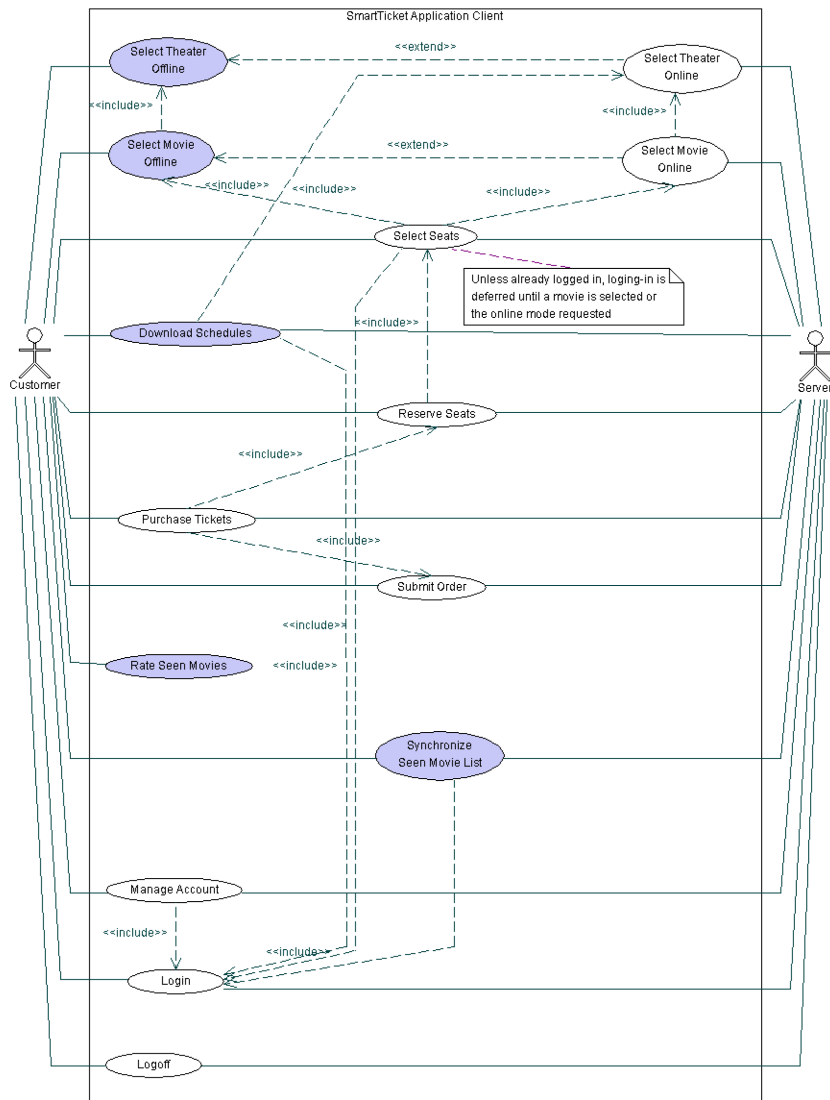


Figure 4 Smart Ticket Offline and Online Use Case Diagram

3 Data Models for Supporting Offline Operation

To support both online and offline modes, the client must maintain a local data model (client-side data model). The Model-View-Controller pattern promotes separating the data model from the presentation (the view and the controller). This pattern isolates the GUI from the data access, which is done via the network from the server or locally from the device's database. This separation is fundamental for the implementation of a disconnected mode of operation.

The Model-View-Controller (MVC) architectural pattern, widely-used for interactive applications, divides applications into three functional components—model, view, and controller—and decouples their respective responsibilities. Each component handles specific tasks and has specific responsibilities to the other two components. The model encapsulates core functionality and data. The view displays the model information to the user and controls the presentation of that information. The controller handles user input (usually forwarded to it from the view) and defines the application behavior in response to the input. Keep in mind that use of this formal architecture on the client may increase the size of the code. You may use this architectural approach but still combine some classes—especially the view and the controller—to achieve optimal code size. However, it's important to keep the model components decoupled to facilitate the implementation of an offline mode.

The client-side data model is supported by a partial or complete replication of the server-side data model. In addition, some of the local data model may be relevant only to the processing that occurs on the device. This might include presentation-oriented data, user preferences, user profile information, or even temporary computation results.

The Facade structural design pattern can be used to hide the complexity of the client-side data model implementation. In a wireless application supporting a disconnected mode of operation, this complexity may arise from the two modes of operation (connected and disconnected) and, more specifically, from replicating, maintaining, and accessing—remotely as well as locally—server-side data, and from accessing client-side-only data. Formally, the Facade design pattern makes it easier to use a subsystem because it provides a higher-level interface that unifies the subsystem's set of interfaces. Often, to reduce its complexity, a system is divided into subsystems. Once this occurs, a common design goal is to reduce the communication and dependencies between the system's various subsystems. Most clients need the services of a subsystem as a whole and do not need access to the

individual classes of a subsystem. Using a facade object, it is possible to provide clients with a single, simplified interface to a subsystem.

Additionally, the Proxy design pattern can be used to abstract the logic that deals with accessing remote data, such as the client/server communication protocol, and any related optimization, such as caching. Formally, a Proxy provides a placeholder to another object to control access to that object. Among its other uses, a remote proxy provides a local representative for an object in a different address space, such as a remote application server. When invoked by the application, the proxy encodes and sends requests to the actual remote object.

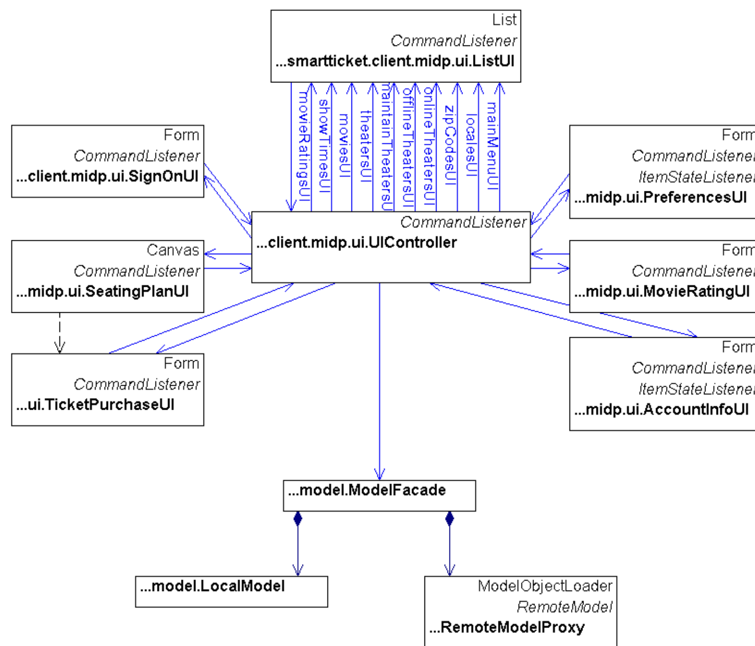


Figure 5 MVC Architecture with Model Facade and Remote Model Proxy in Smart Ticket

Figure 5 shows the implementation of the MVC architecture in Smart Ticket. The user interface (UI) classes, such as `SignOnUI`, `SeatingPlanUI`, and so forth, implement the presentation views and some self-contained, controller-related behavior. The `UIController` class implements a main controller, which controls the overall screen flow and the access to the model. The model is exposed through

a facade, `ModelFacade`, which exposes a simplified interface to both `RemoteModel-Proxy` and `LocalModel`. `RemoteModelProxy` implements locally the same interface as the remote server-side model, and this class handles the communication with the server. `LocalModel` helps manage the local model, both the client-side-only data (such as `Preferences`) and the local replication of the server-side data (such as `Theater`, `TheaterSchedule`, `Movie`). See Figure 8 on page 15 for the breakdown of the model objects.

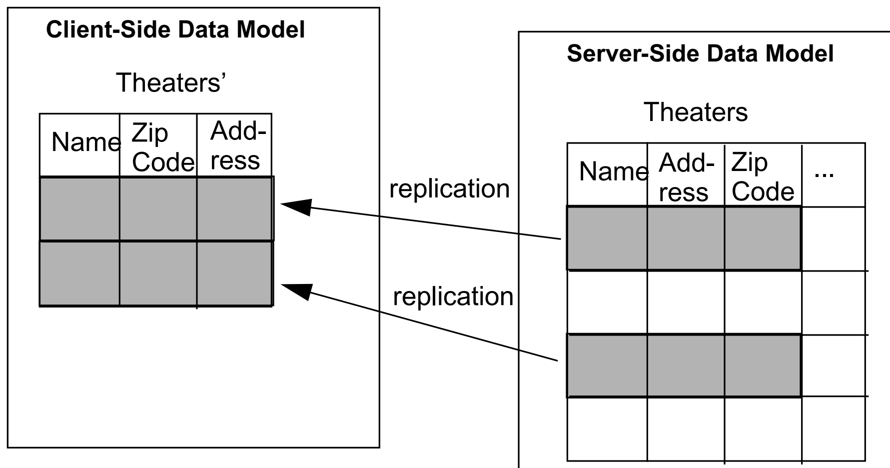


Figure 6 Client- and Server-Side Data Models

The client-side data model that replicates the server-side data model is central to the implementation of a disconnected mode of operation. Because of the memory limitation of mobile devices, the client-side model often replicates only a relevant subset of the server-side model.

Filtering data may be helpful when replicating portions of the server-side model. This is the typical case for an enterprise application where the server-side data model structure may be far too rich for a client to handle. When filtering operations are used, they are applied on the server side. They limit the data downloaded to the client to only what is strictly relevant. Filtering data serves a twofold purpose: it limits the space used for local data on the mobile device and it also reduces the time (and cost) to download the data. Less data on the client side may also improve the client's processing, in terms of speed and complexity. Note that filtering acts on selected data. Selection narrows down the data set to what is

relevant and filtering further eliminates the superfluous pieces of information. In Figure 6, the theater replica contains a subset of theaters from the server-side data model. Note that for the two selected rows, one column has been filtered out.

Often, the client-side data model may not replicate the structure of the server-side model. The client may require a different data representation because of the limitations of the MIDP platform. These limitations include a reduced API and fewer utility classes. When the client-side model uses a different structure from the server-side model, it may be necessary to transform the data.

The client-side data may also be editable or not editable. When the data is read only, it is considered local caching. It is important to identify the proper strategy to keep the locally cached data up-to-date. However, for a client-side model that is editable (read/write) and that can be potentially shared among different users or devices, thought must be given to implementing a more sophisticated and reliable synchronization mechanism. (See Figure 7.)

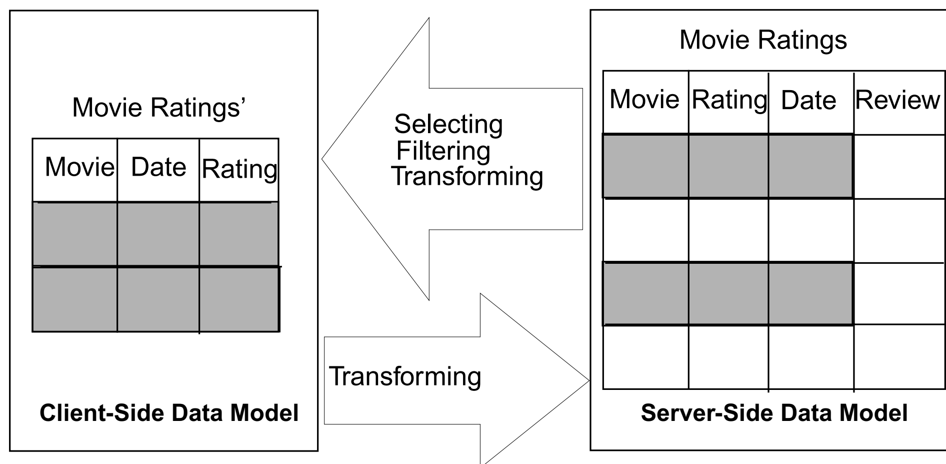


Figure 7 Synchronization of Client and Server-Side Data Models

In addition, the client-side data may be used only to compose or alter the composition of requests to the server. Or, the client-side data may itself constitute the important payload that must be uploaded to the server.

Thus, a client-side data model that replicates the server-side model, regardless of where the data is created, can be characterized as follows:

- Performing partial or complete content replication—How much of the server-

side data is locally replicated?

- Using structural replication or not—How faithfully does the client side replicate the server-side data structure?
- Handling read only or read/write operations on the client-side data—Is the data locally editable?
- Allowing concurrent modification of private or shared data—Is the data shared among users or across devices?
- Handling the expiration of data—Is the data time sensitive?

The movie ticketing application implements to the client-side data model the features summarized in Table 1. The theater movie schedules, for example, are refreshed automatically every week (time sensitivity) and they contain only the schedules of interest to the user (partial content replication). The movie ratings, while not shared among users, is shared through different devices with different connection modalities (concurrent modification) and can be edited by the user (read/write access mode). The shared data must be synchronized with the server.

Table 1 Replication of Server-Side Data Model in Smart Ticket

Features/Constraints	Smart Ticket Examples
Content replication	Partial: theater, theater schedule, movie ratings
	Complete: none
Structure replication	Exact: none
	Transformed: practically all
Access mode	Read only: theater, theater schedule, movie
	Read/write: movie ratings
Concurrent modification	Movie rating
Time sensitivity	Theater schedule, seating plan

Figure 8 shows the relationships between the client-side and server-side data models of the Smart Ticket application. Part of the client-side model replicates the server-side model. For example, `Theater`, `TheaterSchedule`, and `Movie` replicate data from the entity beans `TheaterBean`, `TheaterScheduleBean`, and `MovieBean`,

respectively, and are read only on the client side. Furthermore, `MovieRating` replicates `MovieRatingBean` and, because it can be concurrently edited on the client side, `MovieRating` requires synchronization. Note that `AccountInfo` does not replicate `AccountBean`. `AccountInfo` is actually used to set up an account and to log in. For obvious security reasons, this information is not replicated: it is created locally based on local user input and persisted locally to automate the login procedure.

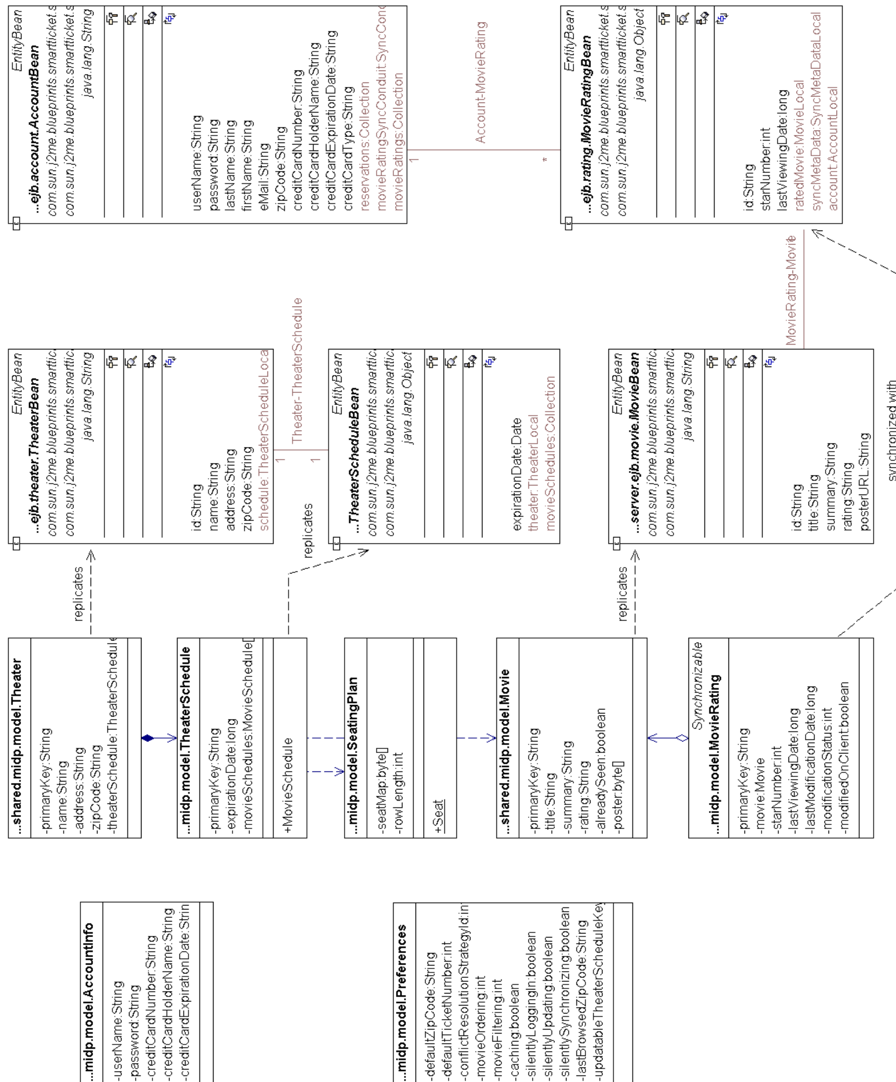


Figure 8 Relationships Between Client-Side and Server-Side Models in Smart

Ticket

Additionally, note that some client-side model objects, such as TheaterSchedule.MovieSchedule, may not only encapsulate local state but may also act as proxies to the remote model. The Proxy design pattern models such proxy use. For example, when getting the seating plan for a particular show time, invoking the method TheaterSchedule.MovieSchedule.getSeatingPlan actually delegates to RemoteModelProxy by means of the ModelObjectLoader interface.

```
public class TheaterSchedule {
  // ...
  public class MovieSchedule {
    // ...
    public SeatingPlan getSeatingPlan(int[] showTime)
        throws ApplicationException {
      try {
          return
              ModelObjectLoader.getInstance().getSeatingPlan(
                  primaryKey, movie.getPrimaryKey(), showTime);
          // SeatingPlan are not cached
      } catch (ModelException me) {
        throw new ApplicationException();
      }
    }
   // ...
}
```

Code Example 0.1 Model Objects Acting as Proxies to Remote Model

As we've seen, replicating the server-side data on the client may require filtering and transformation, driven by the device profile or capabilities, as well as the structure of the client-side model. The server is most efficient for transforming the server-side data structure to the client-side data structure, or for transforming the data structure from the client to the server. Performing transformations on the server reduces the mobile device processor usage and saves battery power. However, since this may tightly bind the server to the client, the presentation layer of a

J2EE application should insulate this transformation process from the business logic. Filtering is subject to the same considerations.

In the Smart Ticket application, the `SmartTicketServlet` and its associated classes, such as `SmartTicketBD`[2], handle in the server's presentation layer the adaptation between the server-side data structure to the client-side data structure. Since the request/response protocol between the client and the server, implemented on top of HTTP, is application-specific and binary-based, the client's and the server's presentation layers are tightly coupled. To ensure the serialization and deserialization compatibility of the request parameters and returned values between both presentation layers, the classes that implement the client-side data model and which support their own serialization and deserialization—such as `Theater`, `TheaterSchedule`, `Movie`, and `MovieRating`—are shared between the client's and the server's presentation layers. These classes are located in the `com.sun.j2me.blueprints.smartticket.shared.model` package. Special care has been taken to ensure that these classes only use APIs that are compatible with both J2ME MIDP and the J2SE and J2EE platforms.

4 Data Persistence

In addition to the ability to process data, an application client device operating in a disconnected mode requires the ability to persist, or store, its data. Persisting data on the client side is central to the typical use cases of wireless mobile devices. Persisting data ensures that a user's work session can be suspended and restored rapidly, thus allowing the user to work productively in the interruptible mode common to these use cases. In addition, it helps with recovery from a temporary connectivity problem, such as a problem caused by a localized lack of wireless coverage.

The MIDP specification provides a mechanism for MIDlets to persist and retrieve data. This persistent storage mechanism, called Record Management System (RMS), is a simple, record-oriented database. The RMS database (also referred to as a record store) consists of a collection of records that remain persistent across multiple invocations of the MIDlet. Each record is stored and retrieved as an array of bytes. The length of each record within a record store can be different, and the format in which the data is stored is not constrained by RMS.

2. SmartTicketBD implements the Business Delegate J2EE design pattern. Hence the suffix BD (ref).

MIDlets can add, retrieve, and remove records from a RMS record store. The RMS API provides methods to compare, enumerate, filter, and monitor records and record stores.

The client side defines two data models: a "working" data model and a persisted data model. (In database terms, a persisted data model is called a database schema.) The mapping between the two models should be task oriented, and must be kept simple yet effective. It also must establish a balance between processor and storage uses.

Because RMS records are stored and retrieved as byte arrays, the application must serialize object state before storing it and deserialize it when retrieving it. Since there is no standard MIDP support for Java serialization, the application must use a custom serialization mechanism.

When persisting data, it is important that the granularity of persistence—the entity or set of entities stored within a single record—be as close as possible, if not equal to, the granularity of usage. That is, you want to have the application's persistence model or schema reflect as close as possible its operation. For example, since the show times for a theater's movies are always browsed together, Smart Ticket stores them in one record as a theater schedule (a theater's movies plus show times) rather than storing each show time in a separate record. However, if the application were to remove movies after they've been seen, then it would be better to store movies along with their show times at individual theaters in separate records.

The same granularity considerations apply to data synchronization. (See "Data Refresh and Synchronization" on page 24.) Going one step further, the granularity of persistence should be kept as close as possible, or equal to, the granularity of synchronization.

Figure 9 shows the Smart Ticket application's persistence of data on the client-side model in the RMS database. The entities depicted in the center of the diagram—namely `INDEX`, `LOCAL_DATA` and `REMOTE_DATA`—are a representation of the record stores in which the different model objects are persisted. Smart Ticket maintains an index (`INDEX`) of the data it stores in the RMS record stores (`LOCAL_DATA` and `REMOTE_DATA`). This index not only maintains the primary keys with which the data may be looked up and its `recordId`, but also additional information about each data item, such as its expiration date, whether it is temporarily cached or explicitly persisted, as well as a `marked` flag that a mark-and-sweep algorithm uses to remove orphaned model objects from RMS. Since the index entries (`IndexEntry`) are small compared to most of the indexed data items, it is possible to effectively manipulate data indirectly from the RMS record stores

without having to actually retrieve the data itself. Note that, depending on the amount of indexed information expected, performance can be improved by partitioning index entries by types over separate record stores. In Smart Ticket, the average amount of indexed information is actually expected to be quite small; therefore, we use only one index.

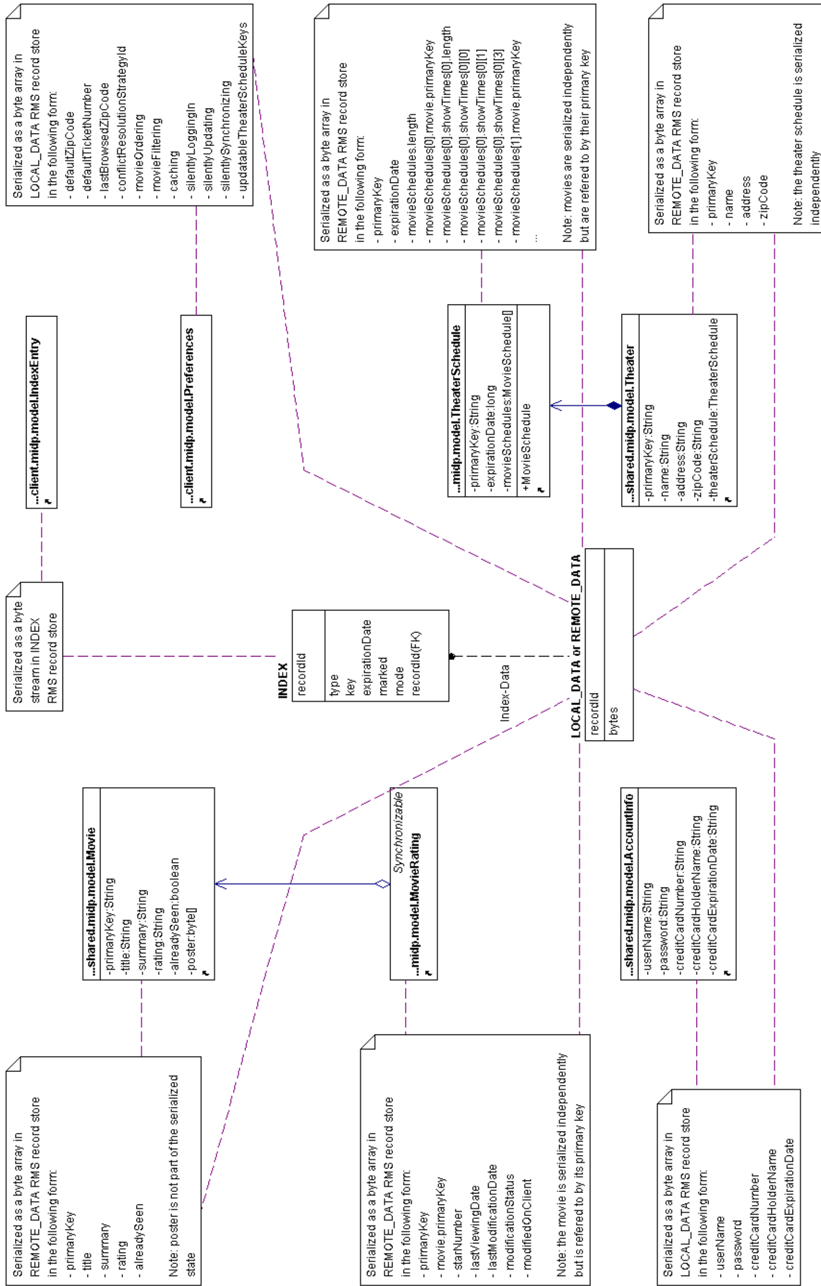


Figure 9 Persistence of Client-Side Data

Overall application size can be reduced and the code simplified by reusing the same serialization mechanism necessary when exchanging objects with the server over the network to perform the persistent object serialization. Smart Ticket, for example, uses the same serialization mechanism for RMS persistence that it uses to download theaters, theater schedules, movies, and movie ratings from the server.

```
public class Movie {
  // ...
  public void serialize(DataOutputStream dataStream)
      throws ApplicationException {
    try {
      dataStream.writeUTF(primaryKey);
      dataStream.writeUTF(title);
      dataStream.writeUTF(summary);
      dataStream.writeUTF(rating);
      dataStream.writeBoolean(alreadySeen);
      return;
    } catch (IOException ioe) {
      throw new ApplicationException(ioe);
    }
  }

  public static Movie deserialize(DataInputStream dataStream)
      throws ApplicationException {
    try {
      Movie movie = new Movie();
      movie.primaryKey = dataStream.readUTF();
      movie.title = dataStream.readUTF();
      movie.summary = dataStream.readUTF();
      movie.rating = dataStream.readUTF();
      try {
        movie.alreadySeen = dataStream.readBoolean();
      } catch (IOException ioe) {
        movie.alreadySeen = false;
      }
      // ...
      return movie;
    } catch (IOException ioe) {
```

```
        throw new ApplicationException(ioe);
      }
    }
  }
```

Code Example 0.2 Serialization and Deserialization of a Movie

To keep memory usage to a minimum and avoid extra in-memory processing, an application should avoid retrieving RMS data that is not relevant to the immediate context. Use the RMS API as much as possible: use `RecordFilter` methods to select relevant records and `RecordComparator` methods to order records to be retrieved. For example, Smart Ticket uses an "already-seen" flag to mark movies that have already been seen. While there is some extra processing to update this flag consistently, it avoids implementing a costly join operation between movie schedule records and records of movies that have already been seen.

The Adapter design pattern is particularly suited for abstracting the details of dealing with RMS and exposing a simplified, domain-oriented interface to its callers. Formally, the Adapter design pattern functions like a wrapper for other classes or interfaces. It is especially helpful for enabling classes with incompatible interfaces to interoperate and work together. A class following the adapter pattern guidelines converts the interface of a second class into a different interface that is more in line with what its clients expect and can use. That is, a class designed with this pattern in mind adapts the incompatible interface of another class in such a way to make that interface usable to clients.

```
public class RMSAdapter {
  // ...
  public Movie loadMovie(int recordId) throws ApplicationException {
    try {
      byte[] data = remoteDataRecordStore.getRecord(recordId);
      return Movie.deserialize(new DataInputStream(
          new ByteArrayInputStream(data)));
    } catch (RecordStoreException rse) {
      throw new ApplicationException(rse);
    }
  }

  public int storeMovie(Movie movie, int recordId)
```

```
        throws ApplicationException {
          try {
            ByteArrayOutputStream stream = new ByteArrayOutputStream();
            DataOutputStream dataStream = new DataOutputStream(stream);
            movie.serialize(dataStream);
            dataStream.flush();
            if (recordId > 0) {
              remoteDataRecordStore.setRecord(recordId,
                  stream.toByteArray(), 0, stream.size());
            } else {
              recordId = remoteDataRecordStore.addRecord(
                  stream.toByteArray(), 0, stream.size());
            }
            return recordId;
          } catch (IOException ioe) {
            throw new ApplicationException(ioe);
          } catch (RecordStoreException rse) {
            throw new ApplicationException(rse);
        }
      }
      // ...
    }
```

Code Example 0.3 Domain-Oriented Methods to Store and Retrieve Movies From RMS

Smart Ticket places most of the RMS-related details within the `RMSAdapter` class—such as determining the record store model into which objects are persisted, error handling, and so forth—while at the same time reusing the same serialization mechanism necessary for exchanging objects with the server over the network. It is possible to change the implementation of `RMSAdapter`, which might be done to address performance-related issues, without impacting its callers as long as the interface is preserved.

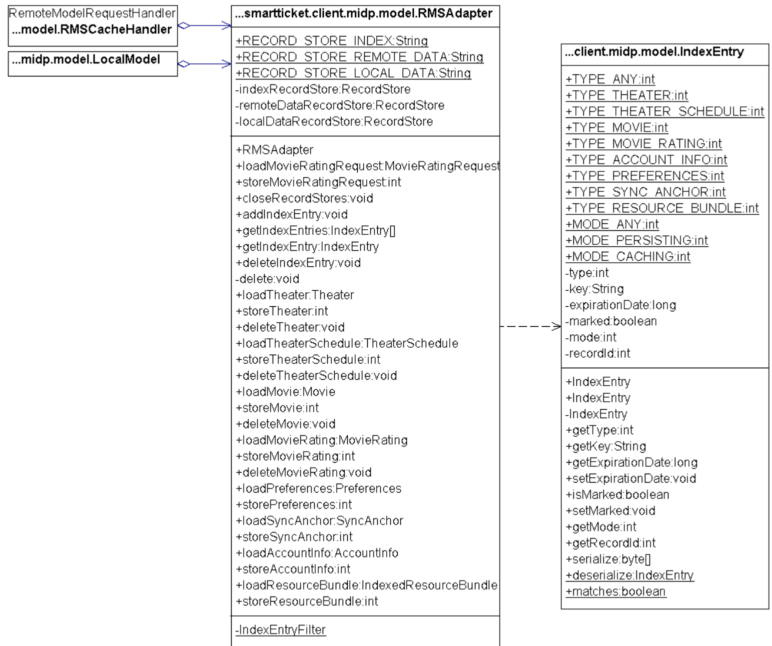


Figure 10 Abstracting RMS-Related Details with the Adapter Design Pattern

5 Data Refresh and Synchronization

Users frequently perform data exchanges with the server. These exchanges are separate from service or processing request submissions. They function to replicate the server-side data on the client-side. As part of these exchanges, a user may retrieve data from the server through the network and store the retrieved data on the mobile device. The user may then disconnect from the network and manipulate the local copy of the data. Later, the user may reconnect to the server and upload to the server changes made to the local data. The user may also retrieve from the server changes made by other users to the data on the server. Depending on the data, the exchange may be a simple data refresh or it may be a more sophisticated data synchronization.

Client-side data refresh is common. Data is read only on the client side but regularly updated on the server side. The client must periodically refresh its local copy even though such data is not considered a replication of the server-side data model. This type of data refresh may be thought of as caching. It is, for example,

implemented by most browsers to improve response time and reduce bandwidth usage.

Server-side data refresh is another common case. The client can edit its data and the data is not shared with others. The data is absolutely private to the user, or it is partitioned in such a way that only one user can access it. This might be a characteristic of a particular business process, such as a sales person owning a particular account, or it might be enforced by the implementation. Such private data needs to be regularly stored on the server—that is, the server copy must be periodically refreshed.

Sophisticated data synchronization is needed in all other cases where data is editable and shared, whether among users or through different devices with different connected or disconnected modes of operation. Overwriting the data is not acceptable.

Some emerging standards, such as SyncML, may address these refresh and synchronization cases under a single data synchronization protocol. This paper addresses data refresh and data synchronization separately because of their different levels of complexity.

Other general issues should be taken into account. For instance, it's important to define the proper granularity of the refresh and synchronization operations so that they take into consideration the limited nature of network connectivity. To account for an unreliable network, perform the commit operation in the local data store when the refresh or synchronization operations have completely downloaded the data, or when adequate granularity is attained (for example, after a complete record is downloaded).

5.1 Data Refresh

Local caching—client-side data that is read only and not editable—requires the proper strategy to keep the cached data up-to-date. An application client may update the local data either automatically (proactively) or when outdated data is accessed (reactively), and usually performs such updates:

- If an expiration date has passed (time based)
- In the absence of notification support, after periodically checking the data's validity on the server (change based)

Data may be retrieved from a local cache or from the server through the network. For each data request made through the model, the application attempts to

retrieve the data from local storage. If the cached data is up-to-date, then the data is directly returned to the requestor. If not, the request is delegated to the server and the returned data is cached locally. It is then returned to the requestor. This behavior is well modelled as a Chain of Responsibilities design pattern.

Formally, the Chain of Responsibilities design pattern is concerned with communication and responsibilities between objects in a control flow. This pattern decouples communicating objects, letting you implicitly send requests to an implicit receiver object through a chain of intermediate objects. With this pattern, a request is sent through a series of multiple objects, and each object may handle the request. The request passes along the chain until it is handled. The first object in the chain receives the request. If it can handle the request, it does. Otherwise, it forwards the request to the next object in the chain. The request is not sent to one specific receiving object for handling—the requesting object has no explicit knowledge of the object that ultimately handles the request. Figure 11 shows how the Smart Ticket application applies the Chain of Responsibilities pattern.

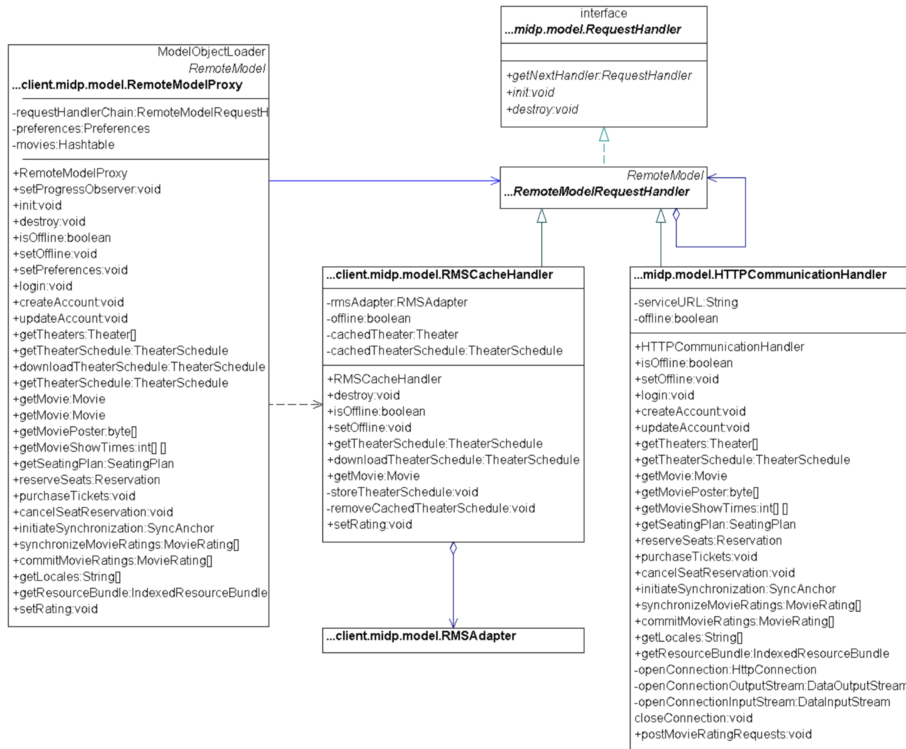


Figure 11 Applying the Chain of Responsibility Pattern to Add Caching Capabilities

To illustrate, the Smart Ticket application lets a user working online choose to download into his wireless device the schedules for selected theaters. The user may then browse these theater schedules when offline. These downloaded theater schedules may be refreshed automatically when they are no longer valid, such as at the regular time new schedules are available, or at the user's request according to pre-set user preferences.

The application persists theater information and movie schedules locally in separate, dedicated record stores. Each movie schedule record contains the name of the movie playing in a particular theater and its show times. Though schedule updates usually occur weekly, individual theaters may update their schedules at

other times. To accommodate this, each theater record contains that theater's schedule expiration date.

When a user launches the Smart Ticket application or accesses the movie schedules, the application checks the locally cached schedule expiration dates. Depending on the user's preferences, the application either updates expired schedules or marks them as out-of-date, leaving it to the user to choose an appropriate time to go online and update the schedules. Thus, the granularity of the refresh operation is tied to the movie schedules for a given theater.

When data is not shared but is editable, the data may be created on the mobile device or it may be downloaded from the server and edited locally. The user may be required by a particular business process to regularly upload the data to the server, or he may willingly choose to do so. A good example might be a data collection application for a warehouse. The user enters into the mobile device inventory for a section of a warehouse. Since only one user is responsible for taking inventory in this part of the warehouse, the data is not shared with any other user. The user may work disconnected from the network and, when completely done, may upload his data to the server, where a batch consolidation of the inventory is performed.

Recall that the UML diagram in Figure 9 shows the Smart Ticket application's persistence of data on the client-side model. The persisted data model includes additional information to manage the expiration of data. For example, the `TheaterSchedule` record contains the field `expiryDate` to handle the time sensitivity of theater schedules.

Figure 11 shows how the `RemoteModelProxy` submits requests to the server-side remote model through a chain of handlers implementing the Chain of Responsibilities design pattern. The first handler in the chain is `RMSCacheHandler`. Apart from the management of a small in-memory cache, its responsibility is to retrieve and maintain up-to-date instances of `TheaterSchedule` (along with the related instances of `Theater` and `Movie`) either from RMS or from the server (by delegating to the next handler). When the `getTheaterSchedule` method is invoked on `RemoteModelProxy`, it in turn calls the `getTheaterSchedule` method on `RMSCacheHandler` (see Code Example 0.4). If it can retrieve an up-to-date theater schedule, either from RMS or from an in-memory cache, then `RMSCacheHandler` returns the schedule directly to the initial caller. Otherwise, `RMSCacheHandler` calls the `getTheaterSchedule` method, which is further down the chain on `HTTPCommunicationHandler`, and `HTTPCommunicationHandler` in turn submits the request to the server using a custom client/server protocol over HTTP. The returned value is then passed back up the chain. If the user has explicitly persisted this theater

schedule in RMS, then `RMSCacheHandler` updates the corresponding RMS records. The `RemoteModelRequestHandler` abstract class, which every handler extends, provides a default implementation for every method that calls the same method on the next handler in the chain, thus allowing a handler to implement only methods it can handle.

```
public class RMSCacheHandler extends RemoteModelRequestHandler {
    private RMSAdapter rmsAdapter;

    public TheaterSchedule getTheaterSchedule(Theater theater) {
        // ...
        IndexEntry indexEntry = rmsAdapter.getIndexEntry
                (theater.getPrimaryKey(),
                IndexEntry.TYPE_THEATER_SCHEDULE,
                IndexEntry.MODE_ANY);
        if (indexEntry != null) {
            if (isOffline() || indexEntry.getExpirationDate() >
                    System.currentTimeMillis()) {
                return rmsAdapter.loadTheaterSchedule
                    (indexEntry.getRecordId());
            } else if (getPreferences().isUpdatable
                    (theater.getPrimaryKey())) {
                return downloadTheaterSchedule(theater);
            } else {
                // delete expired theater schedule which are
                // not updatable
            }
        }
        // Default behavior: call the HTTPCommunicationHandler
        TheaterSchedule theaterSchedule = super.getTheaterSchedule(
                    theater.getPrimaryKey());
        return theaterSchedule;
    }

    public TheaterSchedule downloadTheaterSchedule(Theater theater){
        TheaterSchedule theaterSchedule
            = super.getTheaterSchedule(theater.getPrimaryKey());
         storeTheaterSchedule(theater, theaterSchedule,
            IndexEntry.MODE_PERSISTING);
```

```
            // ...
            return theaterSchedule;
        }

        private void storeTheaterSchedule(Theater theater,
            TheaterSchedule theaterSchedule, int mode) {
            // Store a Theater, its TheaterSchedule and refered Movies
            // into RMS and update or create the related IndexEntries
        }
    }
```

Code Example 0.4 Retrieving Up-to-Date Theater Schedules from RMS or Server

5.2 Data Synchronization

Data synchronization is the process of making the mobile device's locally stored data identical to the data on the server. From the user's point of view, the data on the server may be referred to as the master copy, while the data on the device may be considered the slave or local copy. It's also possible that the master copy is only a subset of the server-side data.

Lack of synchronization between a local and master copy may occur when the client-side data model may be edited and potentially shared either among users or through different devices with connected or disconnected operation modes. In this situation, the concurrent changes made by different users on their local copies may conflict with each other, and this becomes apparent when the different users attempt to update the master copy. Note that the conflict is not one of concurrent access to the master copy. Rather, the conflict is that of multiple users making different or incompatible changes to the same piece of information or data item. When data persistence is directly implemented using RMS, the data items should typically be mapped to records in a record store.

It's important to implement a sophisticated and reliable synchronization mechanism when clients replicate and edit locally shared server-side data. Such a mechanism needs to upload to the server only modified data, with the finest granularity possible, and reconcile that data with the server-side data. A strategy must also be defined to resolve reconciliation conflicts on the server. It must avoid chattiness and be smart enough to keep the user's intervention to a minimum, because of cumbersome user input capabilities and network constraints. Such a strategy must preserve as much as possible of the user's data, as the user may have spent considerable time and energy entering the data.

Data synchronization consists of the following steps:

- **Identify changes**—Identify the changes made to the local data copy, such as inserted, modified, or deleted data items. These changes should be recognized at the proper granularity, typically the RMS record. A change tracking scheme can help identify a list of changed data items. In such a scheme, for example, each data item may include a modification status flag marking its edited status, such as unchanged, inserted, modified, or deleted. On the client side, when the master copy is replicated locally, the flag is set to untouched.
- **Detect conflicting changes**—Detect conflicts between locally changed data items (inserted, modified, or deleted data items) and the master copy's data items. Detecting conflicts between two data items requires a means to uniquely identify both the local and master copies of the data item. This is usually a primary key, with the additional constraint of being globally unique across the server and all involved clients. Such a primary key is often known as a Global Unique Identifier (GUID). Once the two local and master data items are identified, the conflict detection strategy may use a field-by-field comparison of the two items. Or, it may make its comparison based on a modification time stamp. Time-based conflict detection may require a time alignment procedure to account for clock discrepancies between the clients and server. Normally, such conflict detection occurs on the server.
- **Resolve conflicting changes**—Resolve the conflicting changes (that is incompatible or concurrent changes to the same data item both on the client and server) by applying rules to determine which change to choose. Conflict resolution strategy may be automatic (non-interactive strategy) or it may require the user's intervention (interactive strategy). The latter strategy involves both the server and the client. Examples of conflict resolution strategies are:
  - First-modified-wins—Of two discordant data items, automatically elect the item with the oldest modification time (the one modified first)
  - Last-modified-wins—Of two discordant data items, automatically elect the item with the most recent modification time (the one modified last)
  - Client-wins—Of two discordant data items, automatically elect the item modified on the client
  - Server-wins—Of two discordant data items, automatically elect the item modified on the server
  - Ask-user—Present the user with the two discordant data items and prompt

for a decision

  - Application-specific conflict resolution strategies based on business rules

- **Update server-side master copy**—Once all conflicts are resolved, bring the master copy on the server side to an up-to-date status.
- **Identify the updates**—Ideally, the local copy replicates the master copy by downloading only information about its uploaded data items that were concurrently changed and committed or rejected, plus any complementary updates to the master copy, such as non-conflicting changes committed by other users. The downloaded information should also represent only changes made since the last synchronization.
- **Update the client-side copy**—Once the server-side master copy is up-to-date, then the client-side copy can be updated. When the client receives the downloaded information, it blindly applies the updates. The client should also re-initialize change tracking, for example, by resetting the data item modification status flags to `STATUS_UNCHANGED`.

Restricting the exchange of information from the client to the server (and from the server to the client) to only changed data items results in significant improvements in data transfer and response time, particularly when compared to a total data refresh strategy.

Figure 12 illustrates these data synchronization steps.

The Smart Ticket application has addressed data synchronization as follows. A Smart Ticket user can access the application from the MIDP application client or from his desktop, using an equivalent browser-based application. A Smart Ticket user can rate movies online, using the browser application, or offline, using the MIDP client. As a result, the data on the server and the copy on the wireless device must be synchronized and conflicting ratings must be resolved. While the data is not shared among users, it is shared through different devices with different connection modalities. (The devices may operate in connected or disconnected modes.)

Conflicts can potentially arise, as the following scenario demonstrates. Every rating change made through the Web-based browser application is immediately committed to the database. This changes the master copy regardless of what the user has done locally or offline on his mobile device and not yet uploaded to the server. Depending on the user's preferences, Smart Ticket resolves any conflicts

using the last-modified-wins, first-modified-wins, or ask-user conflict resolution strategy.

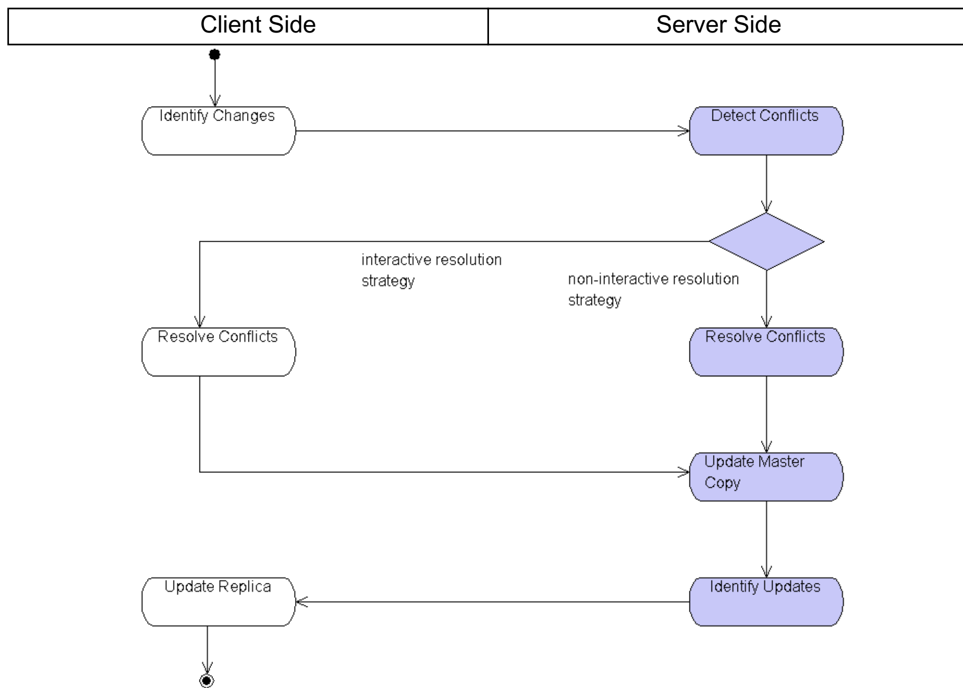


Figure 12 Flow and Partitioning of Synchronization Processing Tasks Between Client and Server

Smart Ticket stores user-submitted movie ratings in a dedicated record store. Each record contains one movie rating, and includes the following information:

- Movie identifier
- Date movie was seen
- Movie rating, from one to five stars
- Modification status flag (`STATUS_UNCHANGED`, `STATUS_INSERTED`, `STATUS_MODIFIED`, `STATUS_DELETED`)
- Date rating was last modified

The movie identifier is the key used for synchronization. It is different from the RMS primary key, `recordId`. Rather, movie identifier is a *remote* foreign key. That is, it is the primary key for movies on the server-side model. The synchronization flag is initially set to untouched. The date the record was last modified is used to implement the first-modified-wins and last-modified-wins strategies.

The `MovieRating` records in RMS include a `modificationStatus` field, a `lastModificationDate` date field, and a `modifiedOnClient` flag. These fields keep track of client-side modifications to achieve synchronization with the server side. (See Figure 9, the client-side persistence model, on page 20.)

In Smart Ticket, the synchronization is handled on the client-side by `SynchronizationAgent`. This class is hidden behind the `ModelFacade`, enabling the facade to expose only a simplified interface to the synchronization mechanism. (See Figure 13.)

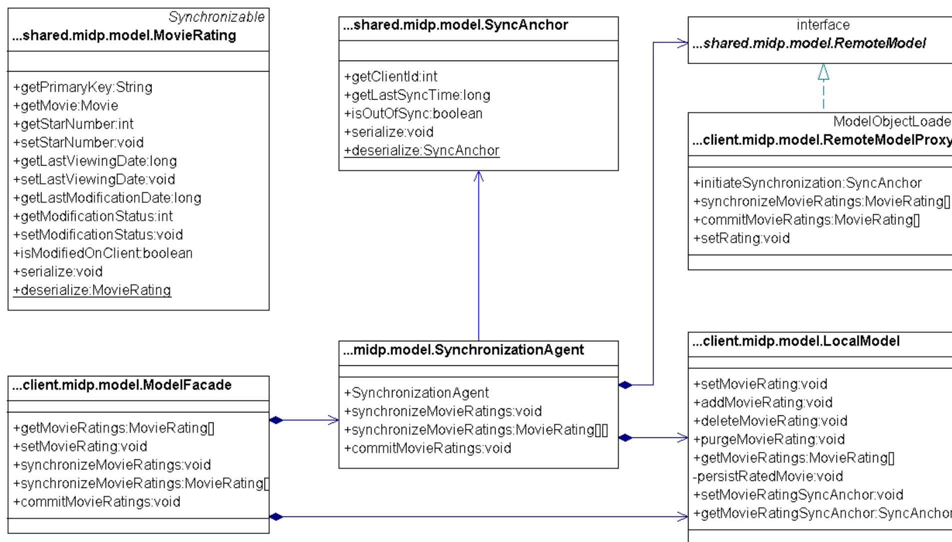


Figure 13 Client-Side Framework for Supporting Synchronization

The diagrams in Figure 15 and Figure 16 show the overall sequence of operations between the client and the server when synchronizing movie ratings. Apart from the implementation details of the generic algorithm presented previously, the initialization and management of the synchronization session are worth examining in detail.

To be able to synchronize different devices, information about previous synchronization sessions must be maintained on a per device manner, both on the devices and on the server. This information includes the identifier of the device and the last synchronization time.

The identifier of the device allows the server to retrieve the synchronization information associated with the device. The identifier is assigned by the server (using a unique identifier generator) to a device the first time the device starts a synchronization. This identifier is saved on the device and used for any subsequent synchronization. If the persistent storage of the device is wiped-out, a new identifier is assigned to the device. When a user uses two different devices to synchronize movie ratings from the same account, the devices are assigned two different identifiers. On the server side, the `SyncConduitBean` entity bean tracks and manages the information about the changes a user makes to the movie ratings—modelled by `SyncMetaDataBean`—and the different clients the user uses to synchronize—modelled by `SyncClientBean`. A `SyncConduitBean` instance is associated with one user account and manages synchronizations with only one set of related entities (in this particular case, `MovieRatingBean`). If we were to synchronize other unrelated entities, an additional `SyncConduitBean` instance would have to handle them (see Figure 14).

The last successful synchronization time is saved both on the client and the server along with the device identifier. The last synchronization time is used to verify that the client and the server are in sync. Since synchronization is an incremental process that exchanges only information about changes that occurred since the last synchronization, the client and the server must absolutely share the same point of reference (also called in this context a synchronization anchor). If this point of reference is not the same—that is, the last successful synchronization time is different—the client and server are not in sync. Such a situation may require a complete refresh of the client-side data from the server-side data, or even a more comprehensive synchronization.

In Smart Ticket, the synchronization process starts with an initialization operation: the client-side `SynchronizationAgent` calls the `initiateSynchronization` method on the server remote model, passing it the instance of `SyncAnchor` that may have been stored locally in RMS or a blank instance of `SyncAnchor` if it is the first time the client attempts a synchronization. This instance of `SyncAnchor` encapsulates both the client identifier and the last synchronization time. The initialization call returns a new instance of `SyncAnchor`, which encapsulates both a client identifier (either the one initially passed or a new one) and, by anticipation, the next "last synchronization time" (assuming that the current synchronization is

successful). The instance of `SyncAnchor` is only persisted in RMS if the complete synchronization process is successful. (See Code Example 0.5 and Code Example 0.6.)

```
public void synchronizeMovieRatings(
        int conflictResolutionStrategyId) {
    SyncAnchor lastSyncAnchor
        = localModel.getMovieRatingSyncAnchor();
    nextSyncAnchor = remoteModel.initiateSynchronization(
        lastSyncAnchor != null ? lastSyncAnchor : new SyncAnchor(),
        new Date().getTime());
    // ...
    // Filter out the unchanged movie ratings
        MovieRating[] updatedMovieRatings
            = remoteModel.synchronizeMovieRatings(
                changedMovieRatings, conflictResolutionStrategyId);
    // Consolidate the updated movie ratings and the
    // local movie ratings...
    // Update local movie ratings...
    localModel.setMovieRatingSyncAnchor(nextSyncAnchor);
    return;
  }
```

Code Example 0.5 Managing and Maintaining Per Device Synchronization Session Information on the Client Side

```
public SyncClientAnchor initiateSynchronization(
        SyncClientAnchor syncClientAnchor, long clientTime) {
    syncConduit = account.getMovieRatingSyncConduit();
    if (syncClientAnchor.getClientId() == -1) {
        // No valid client id assigned to the device:
        // create a SyncClient with a new client id
        syncClient
            = syncClientHome.create(syncConduit.makeSyncClientId());
        syncConduit.getSyncClients().add(syncClient);
        syncClient.setLastSynchronizationTime(0L);
    } else if (syncClient == null || syncClient.getId()
            != syncClientAnchor.getClientId()) {
        // No SyncClient for the current session
```

```
            // or non-matching client id:
            // use the client id to look up the matching SyncClient
            syncClient = syncConduit.getSyncClient(
                syncClientAnchor.getClientId());
            // ...
        }   if (syncClientAnchor.getClientId() == -1
                || syncClientAnchor.getLastSyncTime()
                    == syncClient.getLastSynchronizationTime()) {
            // First synchronization session for a new device or
            // new synchronization session for a registered device:
            // return a new SyncAnchor to the device
            initiationTime = System.currentTimeMillis();
            timeShift = initiationTime - clientTime;
            return new SyncClientAnchor(syncClient.getId(),
                        initiationTime, false);
        }
        // Server and device SyncAnchors out of sync:
        // force a slow synchronization
        syncClient.setLastSynchronizationTime(0L); // to force slow sync
        return new SyncClientAnchor(syncClient.getId(), 0L, true);
    }
```

Code Example 0.6 Managing and Maintaining Per Device Synchronization Session Information on the Server Side

The implementation of synchronization in an end-to-end application, particularly in Smart Ticket, may rely on simplifying assumptions:

- The server is the data repository. It holds the master copy.
- Changes on the server-side may be tracked based on the server's time. The last synchronization time exchanged between the client and the server to check if they are not out-of-sync may be based on the server's time. Changes on the client-side may be simply tracked using a modification flag. In Smart Ticket, when a movie rating is modified on the server, its modification time is updated to the server's current time. When client-side changes are committed on the server, their modification time is reset to the time to be used as the last synchronization time.

- When it exists for the data items to be synchronized, a primary key can be used as the global unique identifier required to identify conflicting changes. To uniquely identify data items within the scope of a particular usage pattern, it is permissible for the primary key to be a compound key. If data items do not have a natural primary key that can be used as a global unique identifier, a primary key can be generated on the server the first time a new data item is created or a client is synchronized (uploaded to the server). In Smart Ticket, a user can maintain only one movie rating per movie; movie ratings can be uniquely identified using the movie primary keys in the scope of a user's account. Since a device may be attached to only one account, movie ratings can be uniquely identified on the client using the movie primary key.

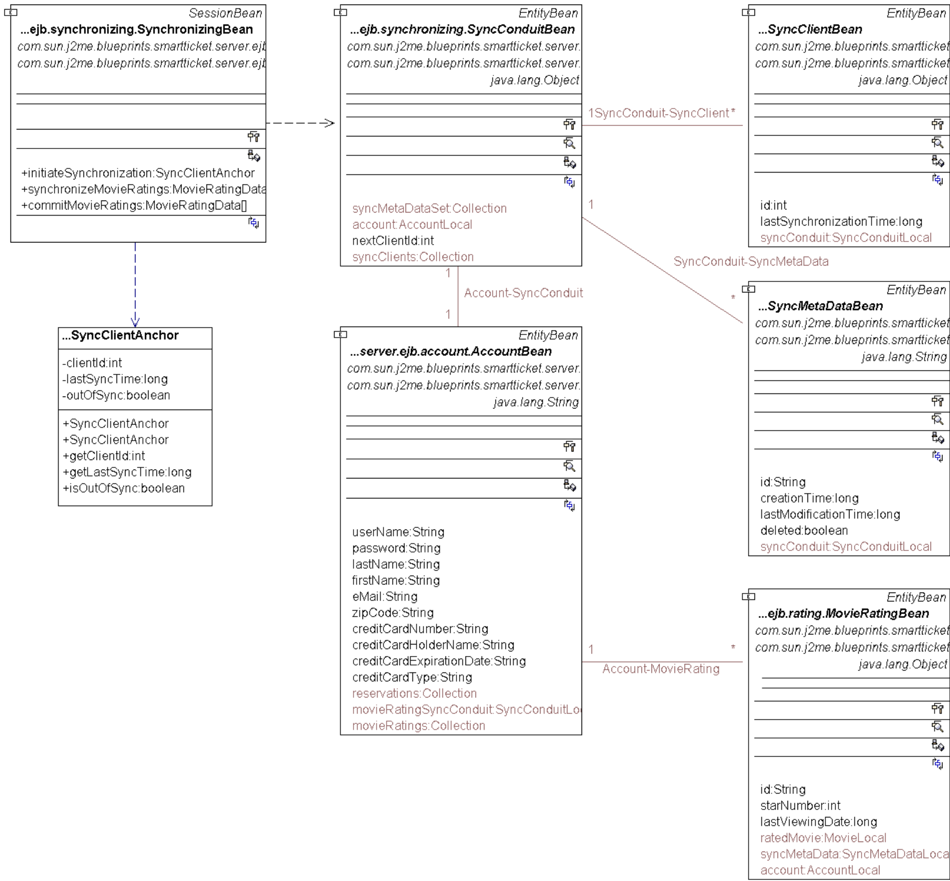


Figure 14 Server-Side Framework For Supporting Synchronization

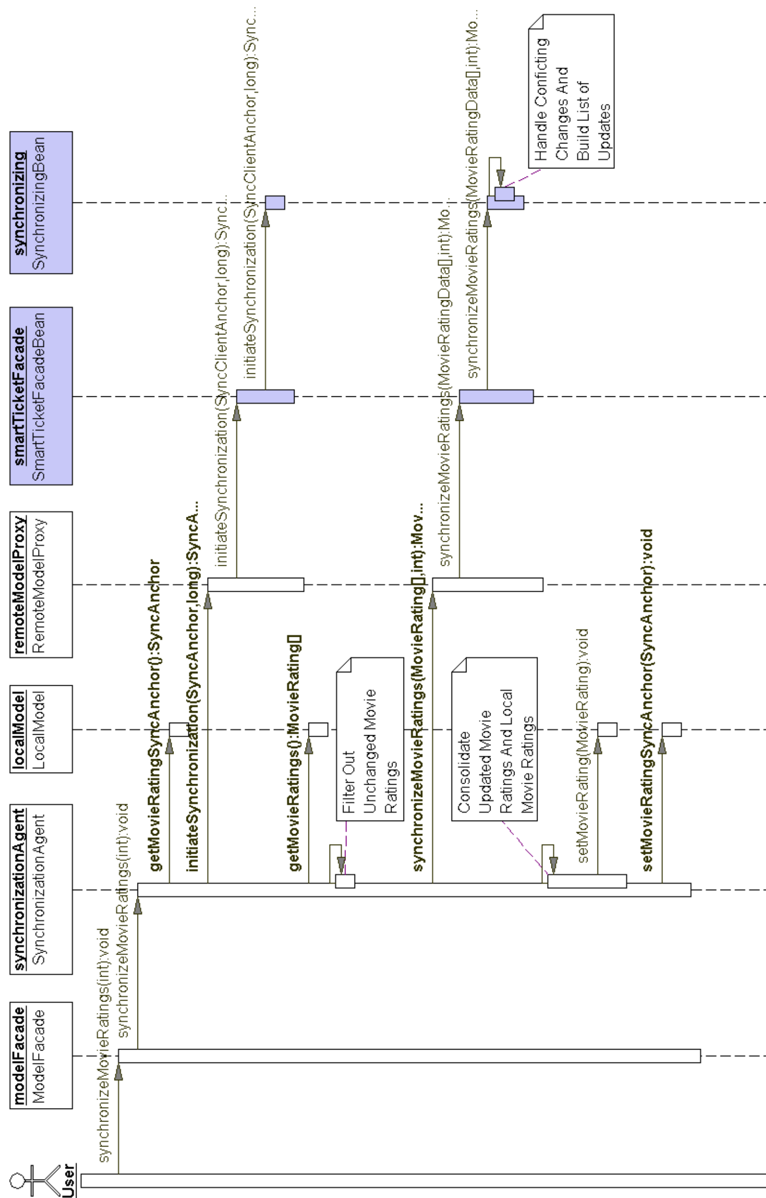


Figure 15 Sequence Diagram of Synchronization With Non-Interactive Conflict Resolution

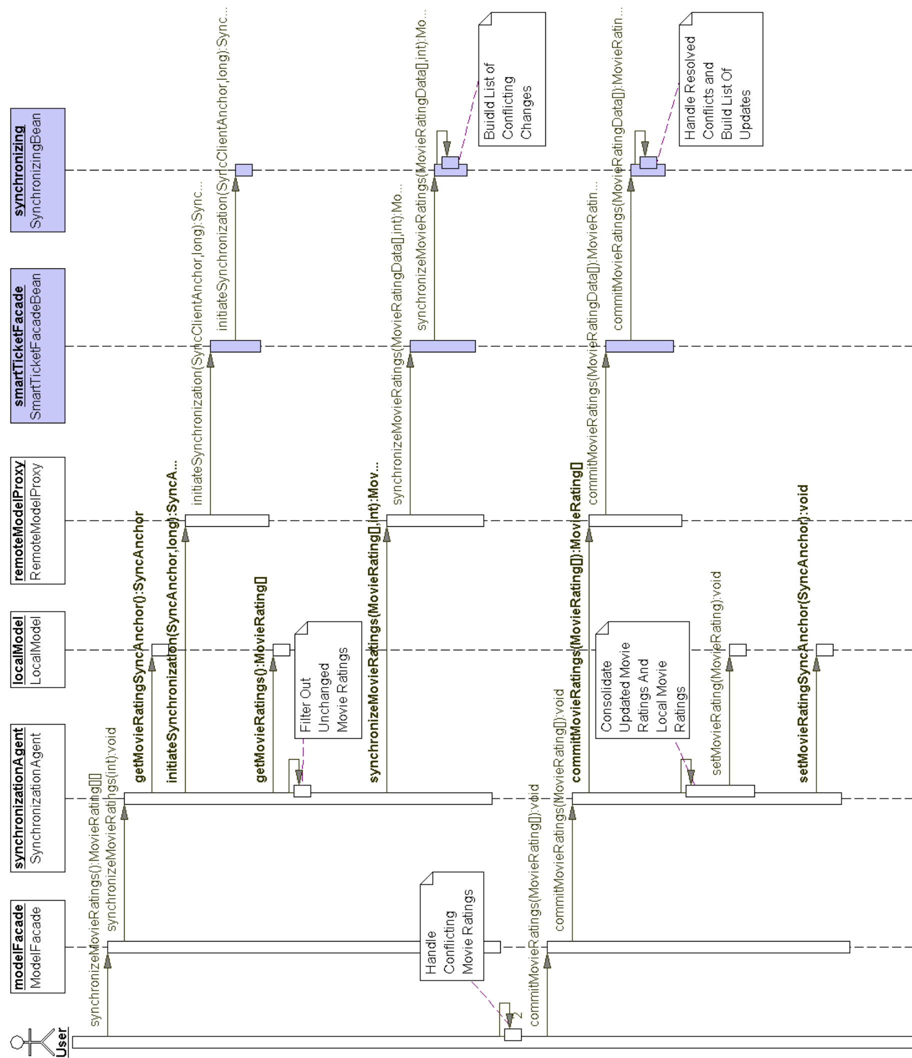


Figure 16 Sequence Diagram of Synchronization With Interactive Conflict Resolution

6 Client/Server Interaction for Supporting Offline Operation

Note: This section will address request queueing and coalescing, as well as store-and-forward messaging, which are two additional techniques that may be used to implement disconnected operations.

7 Security Considerations

To improve the user experience and avoid fastidious authentication procedures, MIDP application clients may provide an automatic login feature. This feature uses a locally stored identifier and password to automatically authenticate the user when the user issues requests to the server. This should be done only with the consent of the user and should be configurable.

However, a disconnected mode of operation requires the persistence of data on the local device. If this data is sensitive, special care should be taken to protect its integrity and confidentiality. Since RMS restricts access to a record store to only the application that owns it, a simple security mechanism is to protect the application itself with an authentication procedure, such as with a password. In certain cases, the business environment may require that the automatic login feature be disabled to protect inappropriate access.

Since data synchronization and data refresh may require exchanging confidential data between the client and server, those exchange procedures should at a minimum rely on a secure protocol such as HTTPS.

8 Usability Considerations

The user experience is an important consideration when designing a disconnected mode of operation, and different usability issues must be taken into account. While some disconnected usability issues are the same as for a connected mode of operation, they take on added significance in a disconnected mode.

8.1 Personalization

A mobile device client has limited memory, persistent storage, and processing power resources. Because the client is limited in the data it can store and manipulate, it is important to optimize the use of its resources. One way to optimize these

resources is to download to the client only the most relevant information to the user and present the information effectively. Taking into account the user's preferences and history, along with the task at hand, helps to narrow down the required information. This process is often referred to as personalization of an application.

Personalization is critical for wireless applications, not only because of their limited resources and network limitations, but also because of their reduced interactivity. Wireless applications have to make do with limited input and display capabilities. Personalization tries to ensure that the most relevant data available is available locally on the device, while less relevant data remains available online. Personalization also tried to shape the data presentation so that the most relevant data is accessible first. For example, proper ordering of information improves browsing and reduces the number of user actions—such as the number of key clicks—required to access desired information.

Smart Ticket filters the downloading of theaters and displays them by the user's preferences. For example, it might download and display only those theaters in the user's zip code. It identifies and displays first the user's preferred theaters—theaters for which the user locally stored schedules—and theaters based on the user's history, in this case theaters at which tickets have been recently purchased.

For movie listings, Smart Ticket again looks at the user's history and removes or displays last those movies that have already been seen. It displays movie show times so that the cursor is pre-positioned on the next available show time, essentially pre-selecting that time. Smart Ticket might also implement more sophisticated browsing optimizations. It could have the server filter out (and not download) movie schedules for movies that have already been seen within some time period, such as the last three months. When offline, the Smart Ticket application client may filter out or order accordingly movies seen recently, such as during the previous week.

8.2 Improving Responsiveness

Poor network connectivity impacts applications operating in a connected mode: they must limit or reuse connections to the server and keep transferred data to a minimum. A disconnected mode of operation tries to compensate for the low quality of network connectivity. A work session operating in a disconnected mode implements application features locally and eliminates the need to connect to the server. It also stores required data locally and thus reduces to zero the amount of data transferred.

Users of that offline work session process and access data locally, resulting in a significant response time improvement.

Although network connection and data transfer may be eliminated during an offline work session, a disconnected mode of operation must still download data to store it locally and upload changed data to commit it to the server. Care must be taken to keep these data refresh and synchronization operations effective, reliable, and as fast as possible. These operations, unlike some other mobile device application scenarios, do not have a direct link with the data repository. (For example, a PDA that can be synchronized with the desktop data repository by placing the device on a cradle.) They occur remotely across the network and are subject to high latency, intermittent connectivity, and limited bandwidth constraints.

While it is possible and often desirable to implement background data synchronization and refresh operations, the choice must still be left to the user. This is because these operations may take a significant part of the available bandwidth at an inappropriate time for the user.

8.3 Balancing Connected and Disconnected Mode Features

The connected mode of operation is the default mode of operation for many mobile applications. This mode implies that all features are available to the user. Resource-intensive processing, and particularly business logic processing, takes place on the server side, although the client side may enforce some business assertions. Business-related assertions on the client side (such as the requirement that a credit card number be twelve characters in length) try to improve the user experience by screening out inconsistent requests to the business logic.

The disconnected mode of operation implies that only a subset of features are available to the client. The application client is not a standalone wireless application, but it does provide a reasonable set of features to the user that should allow for meaningful work to be performed without going online. However, the disconnected mode should not attempt to emulate while offline the resource-intensive processing and business logic processing of the server side. If it does, it should do so with great care. Implementing server features on the client side may degrade their quality, require significant amounts of code, and end up breaking user expectations. This should be avoided. (See Figure 17.) A rule of thumb is to only migrate to the client as offline mode features some of those features implemented in the presentation layer. On the other hand, business logic should not be migrated over to the client.

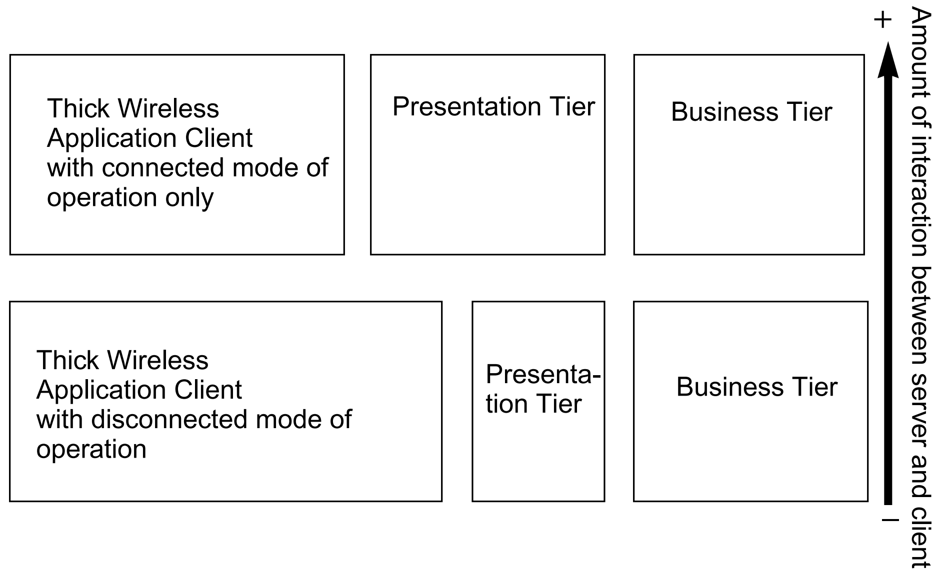


Figure 17 Migration of Presentation Tier Features to Clients for Implementing Disconnected Mode of Operation

It is also important to avoid performing offline operations that are difficult to reconcile on the server side. In particular, avoid over-optimistic locking of server-side resources. For example, Smart Ticket does not persist locally a theater's seating plan or allow it to be browsed offline when close to that theater's show time. Such seating plans are highly volatile as users reserve seats for the upcoming show. Smart Ticket requires a user who wants to reserve a seat to connect to the server and make the reservation against a seating plan with up-to-date availability.

8.4 Awareness of Connected and Disconnected Mode Boundaries

While it is important to keep the same look and feel for the connected and disconnected modes of operation, it is equally as important to not hide the modality from the user. Until network connectivity quality allows for transparent pervasive computing, the user should know whether he is operating in an online or offline mode. Otherwise, the user may be confused and disappointed.

The previously mentioned mobile device constraints (high latency, limited bandwidth, intermittent connectivity, relatively high packet costs) combined with the device's particular use cases make it all the more important to give full control to the user. The user should be completely aware of the boundary between offline and online modes. (See Figure 18.) This boundary may be defined in terms of:

- Server-side data replicated on the client side versus data not replicated on the client side
- Expiration of server-side data that has been replicated on the client side
- Processing requests to be submitted to the server

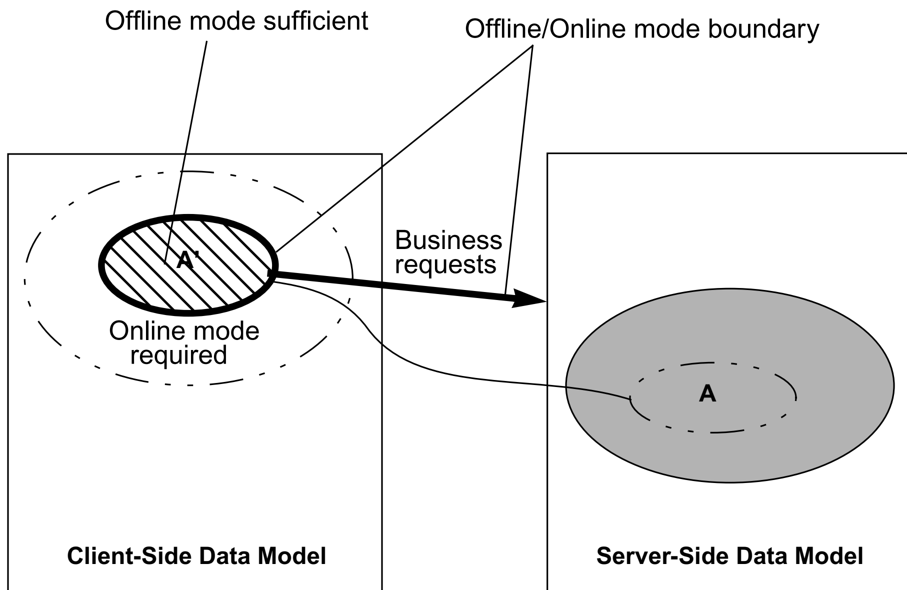


Figure 18 Online and Offline Boundary Awareness

Thus, it's important to warn a user when a particular action may change the user from an offline mode to an online mode. The user should have the opportunity to cancel the action and avoid connecting through the network. It's even better if the user can know beforehand that a particular action may require connecting through the network. An application can achieve this by visually encoding

or labelling actions or items in the user interface whose selection might require changing to an online mode. Such encoded actions can be coupled with dialogs warning the user of the change to an online mode and allow him to cancel the operation. (Note that MIDP 2.0 provides a security feature which prompts the user when connecting through the network.)

9 Conclusion

This paper described the advantages that thick wireless application clients may have by operating in both online and offline modes, rather than operating in online mode only. A well-designed offline or disconnected mode of operation compensates for some of the problems of an online mode of operation, especially high network latency, limited bandwidth, and intermittent connectivity. The paper detailed the design effort that must be made to achieve a workable offline mode. In particular, a wireless application client must address various data management issues, such as the amount of data to persist on the local device, the portion of server data to model on the client, when to refresh such data, and how best to achieve data synchronization.

The paper described these design questions and offered guidelines for effective solutions. Following these guidelines enables an application to achieve a suitable balance between client and server processing. An application can then offer a high-quality offline mode for an overall better user experience.

10 Useful Design Patterns

There are a number of design patterns that can be helpful when designing an application that relies on a disconnected mode of operation. This section summarizes these patterns.

The **Model-View-Controller (MVC)** pattern separates the data model from the presentation (the view and the controller). This pattern isolates the GUI from the data access, which is done via the network from the server or locally from the device's database. This separation is fundamental for the implementation of a disconnected mode of operation. The MVC architectural pattern divides interactive applications into three functional components—model, view, and controller—and decouples their respective responsibilities. Each component handles specific tasks and has specific responsibilities to the other two components. The model encapsulates core functionality and data. The view displays the model information to the

user and controls the presentation of that information. The controller handles user input (usually forwarded to it from the view) and defines the application behavior in response to the input.

The **Facade** structural design pattern hides the complexity of the client-side data model implementation. The Facade design pattern, by providing a higher-level interface that unifies a subsystem's set of interfaces, makes it easier to use that subsystem. Often, to reduce its complexity, a system is divided into subsystems. Once this occurs, a common design goal is to reduce the communication and dependencies between the system's various subsystems. Most clients need the services of a subsystem as a whole and do not need access to the individual classes of a subsystem. Using a facade object, it is possible to provide clients with a single, simplified interface to a subsystem.

The **Proxy** design pattern can be used to abstract the logic that deals with accessing remote data, such as the client/server communication protocol, and any related optimization, such as caching. A Proxy provides a placeholder to another object to control access to that object. Among its other uses, a remote proxy provides a local representative for an object in a different address space, such as a remote application server. When invoked by the application, the proxy encodes and sends requests to the actual remote object.

The **Adapter** design pattern is particularly suited for abstracting the details of dealing with RMS and exposing a simplified, domain-oriented interface to its callers. The Adapter design pattern functions like a wrapper for other classes or interfaces, enabling classes with incompatible interfaces to interoperate and work together. A class following the adapter pattern guidelines converts the interface of a second class to a different interface more in line with what its clients expect and can use. That is, a class designed with this pattern in mind adapts the incompatible interface of another class in such a way as to make that interface usable to clients.

The **Chain of Responsibilities** design pattern is concerned with communication and responsibilities between objects in a control flow. This pattern decouples communicating objects so that you can implicitly send requests to an implicit receiver object through a chain of intermediate objects. With this pattern, a request is sent through a series of multiple objects, and each object may handle the request. The first object in the chain receives the request. If it can handle the request, it does. Otherwise, it forwards the request to the next object in the chain until it is handled. The request is not sent to one specific receiving object for handling—the requesting object has no explicit knowledge of the object that ultimately handles the request.

A **Factory Method** design pattern defines a generic interface that can be used to create an object. However, a class using this pattern defers to its subclasses the decision as to which particular class to instantiate.

A **Session Facade** design pattern provides a unified interface to a set of interfaces in a system or subsystem. Often, a Session Facade is a session bean that provides an interface to a set of entity beans. A Session Facade is a higher-level interface that makes it easier to use the subsystem.

A **Business Delegate** design pattern is useful in distributed applications, where remote component look up and exception handling can be complex. A Business Delegate is an intermediate class that decouples application code from business components used by the application. Not only does the pattern manage the complexity of distributed component handling, it may also facilitate simplifying the business component interface.

11 Resources

For more information on designing wireless enterprise applications using Java technology, and to download the Java Smart Ticket sample application, visit the Java BluePrints for Wireless Web site at:

`http://java.sun.com/blueprints/wireless/`

To get started building wireless enterprise application using Java technology, use the following tools and development kits:

- The J2ME Wireless Toolkit, available at
  `http://java.sun.com/products/j2mewtoolkit/`
- The J2EE Software Development Kit, available at
  `http://java.sun.com/j2ee/download.html#sdk`
- Various Java IDEs, available through
  `http://java.sun.com/tools/`

For introductory information on wireless and enterprise Java technologies, consult the following Web sites:

- *The Java Tutorial, Third Edition: A Short Course on the Basics.* M. Campione, K. Walrath, A. Huml. Copyright 2000, Addison-Wesley. Also available as

<http://java.sun.com/docs/books/tutorial/>

- J2ME technology Web site <http://java.sun.com/j2me/>
- J2EE technology Web site <http://java.sun.com/j2ee/>
- *Programming Wireless Devices with the Java 2 Platform, Micro Edition.* R. Riggs, A. Taivalsaari, M. VandenBrink. Copyright 2001, Addison-Wesley.
- *The J2EE Tutorial.* S. Bodoff, D. Green, K. Haase, E. Jendrock, M. Pawlan, B. Stearns. Copyright 2001, Addison-Wesley. Also available as <http://java.sun.com/j2ee/tutorial/>

The following references cover advanced topics in wireless and enterprise Java application design:

- *MIDP Style Guide.* Copyright 2002, Sun Microsystems, Inc. <http://java.sun.com/j2me/docs/alt-html/midp-style-guide/style-guideTOC.html>
- *Designing Enterprise Applications with the Java 2 Platform, Enterprise Edition.* I. Singh, B. Stearns, M. Johnson, Enterprise Team. Copyright 2002. Also available as <http://java.sun.com/blueprints/enterprise/>

Also consider joining the following interest lists hosted at Sun:

kvm-interest@java.sun.com
j2eeblueprints-interest@java.sun.com

Details on how to subscribe to these lists are available at:

http://archives.java.sun.com

Readers may send comments on this paper to:

j2ee-j2me-blueprint@sun.com