

Gold Rush: Mobile Transaction Middleware with Java-Object Replication

Maria A. Butrico, Henry Chang, Anthony Cocchi,
Norman H. Cohen, Dennis G. Shea, Stephen E. Smith

*IBM Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598*

{butrico,hychang,tony,ncohen,shea,steve}@watson.ibm.com

Abstract. Gold Rush is middleware supporting the writing of Java applications that reside on an intermittently connected mobile client device and access an enterprise database on a central server. While the client is connected to the central server, objects constructed from database entities can be cached in a persistent store on the client. While the client is disconnected, these entities can be manipulated within transactions that are logged on the client. Upon reconnection, the client application can replay these logged transactions to the server, modifying the database. A replayed transaction is checked for conflicts with other database updates that have occurred since the client obtained the input data for the transaction, and the client is notified when such a conflict arises. Communication between the client and the server is optimized to economize the use of a slow or expensive connection such as a radio link.

Introduction

Continued rapid advances in mobile computing are allowing computing and communication technologies to be applied where they are most effective and productive. For a traveling salesperson, this is typically the customer's place of business. For a health-care worker, it is the point of care.

Disconnecting the client platform from a high-speed network, to provide workers with access to enterprise data when and where they need it, presents many difficulties. One such difficulty is the limited capability of the mobile worker's platform (e.g. processor speed, disk capacity, battery life). Another is the nature of the mobile communications link (e.g. low bandwidth, high cost, frequent disconnections).

Application connectivity requirements for data access vary widely. Some applications require high

degrees of connectivity even while mobile. For example, a stock trader requires essentially instantaneous access to the constantly changing prices of stocks while the market is open, and the ability to execute a trade quickly. This requires the trader to pay the high cost of wireless communication, or to be tethered to a phone line while out of the office.

In contrast, a financial planner can perform most tasks while disconnected from the network. The financial planner might begin the day by connecting to a central server and downloading data, such as current market conditions and the portfolios of customers to be visited, from the enterprise financial database to the client device. While visiting with a customer, disconnected from the network, the financial planner might run a Java application on the device to explore various "what-if" scenarios. If it becomes necessary to obtain additional data that was not previously downloaded, the planner could connect to the central server with a brief phone call, download the missing data to the client machine, disconnect again, and resume the use of the program. If the client decides to change the portfolio, the financial planner could execute a local transaction ordering the change. The planner could connect to the central server immediately to replay the local transaction on the central server, or connect at the end of the day to replay all the day's local transactions at once.

In this paper, we describe Gold Rush, middleware that provides lightweight, platform-independent mobile clients with object-oriented, transaction-based access to enterprise information over a weakly connected or primarily disconnected link. Gold Rush includes a client-side persistent store, an object-replication layer to track and minimize data traffic, and an intelligent transaction replay engine. The middleware helps facilitate the development of mobile applications.

These are applications that, like the financial planning application, enable off-line, occasionally-connected workers to execute transactions on enterprise data. Gold Rush enables the financial planner to replicate part of his financial and customer databases, to execute off-line transactions and log them on the disconnected client device, and finally to transmit the logged transactions back to the enterprise's financial and customer databases, checking for conflicting updates.

Gold Rush is not appropriate for every mobile application. Gold Rush is most useful for applications that execute in an environment where client systems are disconnected most of the time. For example, Gold Rush was *not* designed to address the needs of the stock trader who needs continual real-time access to database information. While the device used by the stock trader might in fact be mobile (more precisely, untethered), from the point of view of network connectivity the device is always connected. There are interesting problems that must be solved to keep a wireless communication protocol operational in this environment, but those problems are beyond the scope of the Gold Rush project. In general, applications where the central database changes rapidly, and where the latest version is always needed for a transaction, are not implementable in an occasionally connected environment.

The suitability of Gold Rush for a given application also depends on the application's frequency of conflict. If the application naturally exhibits a low degree of conflict, then it is well suited for the Gold Rush environment, which will allow for conflict resolution. However, if the application typically generates a large amount of conflict, then the application ought to be redesigned, or executed in a connected environment. For example, a set of transactions that always updates a shared counter cannot run in a disconnected environment without conflicting at every transaction replay. A solution in this case is to avoid updating the shared counter at the disconnected client, and to do so later when the transaction is being replayed at the server. That is, the client transaction can be redesigned to specify the amount by which the count should be increased or decreased rather than the value by which the count should be replaced. Given this redesign, the application cannot make use of the actual value of the shared counter. However, if the application cannot be redesigned to avoid using the current value of the shared counter, then the application should connect to commit each transaction.

Off-Line Transaction Requirements

Mobile client applications require access to enterprise data. It is not practical to rely on a constant wireless connection to a server for this access, because radio devices drain batteries quickly and because any-time, any-place wireless links are expensive. Therefore, we replicate enterprise data from the database server on mobile client devices.

The problem is how to write business applications to run seamlessly in both connected and disconnected off-line modes. Will different applications be needed for off-line mode, or can the same application be used with some features disabled? What requirements are imposed by a mobile application, beyond those of a classical client/server connected-mode application? Which data should be replicated, how should it be replicated, and how should the replicated data on the client be synchronized with the data on the main server? How might conflicts arise, and how should they be resolved? To answer these questions, we must examine the characteristics of both business data and off-line business applications.

Off-line business applications are structured into transactions to guarantee atomicity, concurrency control, and durability of the data. A transaction represents a set of read/write operations upon business data. Transactional access enforces integrity constraints: Noncompliant transactions are aborted and accepted transactions are committed into the accumulated state of the database. An off-line transaction commits data into a client's local store while the client is disconnected. The commit is replayed to the master database when the client is reconnected. This approach of a *lazy* commit, consisting of two stages, is necessitated by the occasionally connected nature of mobile applications. In an analytical paper [GH96], Gray et al. compare several transaction propagation strategies and conclude that the lazy-commit approach is the one that scales well and fits into mobile environments.

Conflicts may arise during reintegration with the main database. For example, if multiple clients change the same field while they are disconnected, the conflict can only be detected during reintegration. The previously committed data of the disconnected clients might then be reconciled using a predetermined formula based on the context of the transaction or the data itself.

The flow of the off-line transaction model of application development can be summarized as follows:

1. **Check out:** partial replication of business data from the business server together with its integrity constraints
2. **Access:** off-line transactional access with all the read/write information logged
3. **Check in:** reintegration of off-line transactional data with main database
4. **Conflict handling:** detection and resolution of conflicts with predefined formulas or repair utilities

Transactions provide off-line, occasionally connected business applications with a design model for data integrity and conflict prevention. Without such a model, we would not be able to determine the scope of conflict and to make proper repairs to reintegrate off-line data with the master database. Our model of off-line transactions is similar to the transaction model of network-partitioned databases, but less stringent on the client side, because the mobile client is a single-user, one-application-at-a-time system, and playing a second class role relative to the master database.

Related Approaches

Existing Java remote-database-access products based on the JDBC API [Ja97] are designed for permanently connected clients. These include IBM's VisualAge for Java [IB97] and Symantec's dbANYWHERE [Sy97]. In contrast, Gold Rush supports an occasionally connected client. JDBC provides access to data in terms of relational-database tuples. In contrast, Gold Rush supports manipulation of Java objects.

Several methods have been proposed to allow mobile workers to access information from central data bases. The methods are as varied as the type of information: files, relational data bases, web pages, etc. For mobile file access, the Coda remote file system of CMU pioneered the notion of disconnected operations [KI92], based on file-level transactions isolating groups of changes to files by an application [LS94]. For mobile access to documents, Lotus Notes [Lo96] handles two-way data replication, allowing document-level and field-level propagation, but without grouping changes into transactions. Several approaches have been suggested for access to database information. Of greatest interest to us are those methods which not only make the information available for reading, but also allow changes to be written.

An alternative approach is to download a portion of the central database to a private database on the mobile client. The smaller database is accessible through traditional interfaces, such as ODBC or JDBC,

residing on the client. The application makes all modifications to the smaller database. When the mobile client is able to communicate again with the central database, the changes made to both databases are reconciled. The reconciliation is carried out by software usually called a *replicator*.

If the replicator detects a conflict during reconciliation, it acts according to its configuration. Typically, replicators can be instructed to carry out some default action in case of conflict—for example, merging the changes if possible, or discarding the tuple with the older timestamp—and also allow for custom-programmed actions.

Replicators are often tightly coupled with the implementations of the database systems both in the mobile client and the centralized system. In addition to a complete replicator, that is, one that can incorporate changes from both sides, this approach requires the availability of a suitable small server that can host the smaller database on the mobile client.

This design, and the suitability of the replication and reconciliation mechanism in mobile or other environments, have been studied in depth. [GH96] points out the instability of some replication methods and proposes algorithms that alleviate this problem; [Fr96] addresses scalability and availability issues; [RZ96] discusses the effect on transactions on disconnected operation, and proposes a transaction management model; [YT96] proposes optimistic concurrency control, and addresses migration and replication methods; [Pi96] presents a method for replication in the presence of challenging connections; [ZF96] proposes another replication method and analyzes its performance; [Wo95] evaluates yet another strategy using an application for travel agents; [YW94] describes an algorithm for dynamic allocation of replicas; [AN93] discusses replication organization, and reconciliation methods. Finally, major database vendors offer database-access products for mobile workers based on this approach ([Sy96], [Or95], [IB95]).

A replicated database burdens the mobile client with a database server and with logic to access the data stored on this server. Recently, the increasing popularity of Internet and intranet applications has made lightweight clients desirable. Rather than placing a database server on the mobile client, a three-tier architecture with *mobile transaction middleware* gives the client access to server data without tying the client to a specific database implementation. Three-tier systems move the interface between the application and the database to a central server. The Tactica Corporation has a commercially available product, Caprera, which

supports off-line *long-lived transactions* and three-tiered access to databases [La96].

Our Approach

In a mobile database application, *mobile transaction middleware* provides mobile connectivity and mobile data management. The mobile middleware provides support for:

- a wire-efficient access protocol
- object caching and replication
- logging of deferred transactions
- a server-side object server to reduce the frequency and duration of slow-link connections

It is not sufficient simply to extend database query capability to the mobile client. There must be services to manage the data for mobile use.

Gold Rush mobile data management is based on Java objects. Java has attracted wide interest because it facilitates cross-platform deployment. Furthermore, the Java Remote Method Invocation (RMI) API [Su96] supports remote method-call and object-shipping paradigms, which are useful for both connected and disconnected operations. Java technology is very well suited for mobile database-access applications.

An objective of Gold Rush is to make enterprise data available to Java applications. Enterprise data is most likely to be found in relational databases, VSAM data sets, or IMS databases; a very small portion of such data is in object databases. Typically, a one-time conversion of these relational data bases into object data bases is not possible because of other existing applications that regenerate and alter the data stored in

them. Our current prototype supports mappings between relational data base tuples and Java objects.

In a connected environment, one can use a remote method call to access an enterprise database or to download an object for temporary caching. In an occasionally connected environment, one must first download Java classes and data to the Java-enabled client. Java applications or trusted applets can then support disconnected operations through locally persistent objects. Application code can manipulate local and remote objects uniformly, through the same object interface.

The Gold Rush three-tier architecture consists of a Java client, an intermediate mobile object server, and a back-end data store (see Figure 1). The mobile middleware resides partly on the client and partly on the intermediate server. The middleware presents the client application with the same transaction API regardless of connection mode, except that the database cannot be queried in disconnected mode. Thus, a method call that would obtain a service directly from the server in connected mode invokes middleware that transparently performs that service locally in disconnected mode, using locally available resources.

This is not to say that the application programmer is oblivious to the mobile nature of the application. The parts of the application that are specifically mobile and must be exposed to the programming interface include the handling of the modes of connectivity, prefetching and downloading objects, controlling the replay of transactions, and resolving conflicts during reconnection.

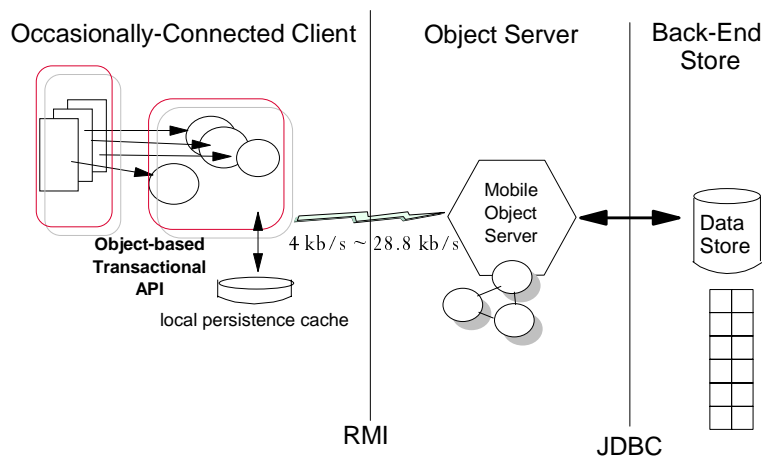


Figure 1. Our three-tiered architecture

The Gold Rush middleware has the following basic components (see Figure 2):

- **Database objects:** A *database object* is a Java object that represents an instance of a database entity.
- **Object caching:** To support disconnected transactions on data base objects, we provide a persistent local object store on the client.
- **Transactions with optimistic concurrency control:** The client works primarily off-line on data objects in the mobile device. Transactions are logged on the client side and replayed to the server when connection is established. An object read by a transaction may be either locked with an optimistic read lock or left unlocked. An object written by a transaction is locked with an optimistic write lock.
- **Communication:** Database objects are transferred between client and server using RMI and Sun's object serialization mechanism. To optimize this communication, we keep track of objects known to be present both at the client and at the server and transmit only the differences between the object version to be transmitted and the version, if any, known to be stored remotely.

The following sections describe the correspondence between database objects and relational database entities, the persistent client object store, the off-line transaction model, and optimization of communication.

Correspondence Between Relational Data Bases and Java Objects

A database object is a Java object that corresponds to a row in a relational database table. Each such object belongs to a subclass of a Java class named `Entity`. Each such subclass corresponds to a table of the relational database.

In a relational database, tuples in different tables are related through primary and foreign keys. We distinguish among 1-1, 1-*n*, and *m-n* relationships. (1-1 relationships are special cases of 1-*n* relationships.) If a 1-*n* relationship exists between two object classes, then the table corresponding to one object class must have a foreign key into the table corresponding to the other class. If there is an *m-n* relationship between two object classes, then there must be a third table with foreign keys into both of the tables corresponding to the related classes.

Our system provides methods to retrieve database objects that satisfy queries, for example, a query on foreign keys, when the client is connected to the server. To allow retrieval of these collections when the client is disconnected, we provide methods to associate names with collections. These named collections are persistent at the client.

An application using our system would include a layer to insulate the details of relational database storage, such as foreign keys, from the manipulation of the objects themselves. Such a layer would provide a subclass of `Entity` for each kind of database object, defining the object's properties and its methods. This class would also supply methods for navigation among objects, using the Gold Rush query facility, and associate unique names with collections to allow the

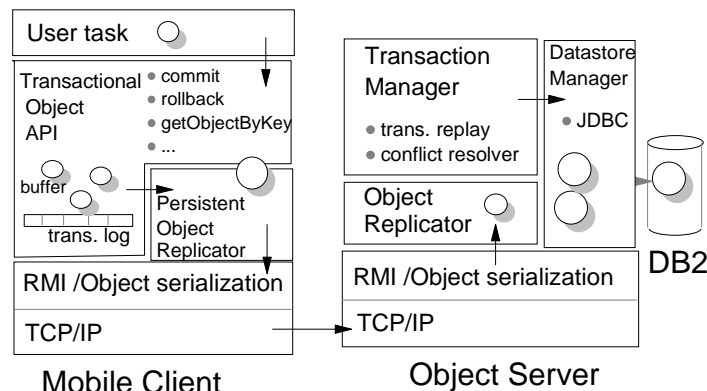


Figure 2. Mobile middleware components

association between objects to persist. Finally, the application would supply a *data manager* class for each subclass of *Entity*, establishing the correspondence between object properties and database fields and implementing the database retrieval and store function.

We have written a tool that allows the application developer to map relational data to object classes. This tool generates the code that defines the classes of database objects; the code needed to instantiate the objects from tuples in the relational database and write object instances into the relational database; and the code that navigates between related object instances.

The mapping tool is a Lotus Notes application. We chose Lotus Notes because it offers flexible storage to represent the association between objects' attributes and fields in tuples, a fast way to develop the user interface through which the programmer establishes this association, and a sufficiently powerful programming language to support the code generation.

It is possible to map a subset of one tuple to a subset of an object. In practice, however, an entire tuple is typically mapped to an entire object. The mapping tool does not support the mapping of an object to multiple tuples, whether from the same table or from multiple tables.

The tool allows the relationships between tuples to be reflected in relationships between object instances, through generated code which allows the application to retrieve related object instances, or establish relations among object instances. Code is generated for `setXXX` and `getXXX` methods (where `XXX` represents an attribute of a database object) in both classes corresponding to a relationship. A `getXXX` method retrieves the associated object instances (one instance in the case of 1-1 relationships); a `setXXX` method establishes such relations.

For each subclass of *Entity*, the tool generates a corresponding JDBC-based data-manager class that provides SQL statements for retrieval, insertion, and update. Each data-manager class provides a method that retrieves data from a particular database table based on an SQL query and returns a Java collection object containing one Java database object for each row in the relational database that satisfied the query. There is also a method to retrieve an object given its unique object ID. The class manager's insertion method takes a Java database object and inserts a new row into the database, checking that uniqueness constraints are not violated. The data manager's update method takes a Java object and replaces one row in the relational database with the data found in the object, checking for update conflicts.

Persistent Client Store

We use Sun's object serialization as the principle means of generating a persistent form of object. To further control the persistent state, and to improve efficiency, we have implemented the serialization methods `readObject` and `writeObject` selectively on certain complex internal objects.

To cache persistent data on the mobile client device, we create a small single-user persistent store. This object store provides object lookup, store, update, and retrieval functions for the objects used by applications in off-line transactions. The main operations of the store are:

- **Caching:** Data is cached into the client's persistent store during connected transaction processing. The latest version of an object is saved in the store in preparation for subsequent disconnected operations. (The communication optimizations to be discussed later prevent the redundant transmission of an object already residing in the client's store.)
- **Retrieval:** The application code residing on the client reads objects from the persistent store while the client is disconnected. An application may retrieve an object by calling a method that retrieves the latest version of an object with a given ID, or by calling a method that returns a collection containing all the locally stored objects of a particular class.
- **Committal:** All mutated objects in the committed transaction are stored and updated. A transaction record is created and the information is stored in the transaction log for future replay.
- **Replay:** The client retrieves the transaction log and replays it to the master database, guided by the dependency relationships among the transactions. After replay, the persistent store cleans up itself by removing stale and old versions of objects as well as old transaction information.

Since the client store is a single-user, one-application-at-a-time persistent store, we choose a file-based design with careful write sequencing to guarantee that the on-disk data is always in a consistent state. The store consists of the following kinds of files:

- **Class files:** Objects of the same class are stored in a single file. This file contains all versions of all objects of a given class resident on the client. A new version of an object is written when a transaction modifying that object is committed.
- **Class index files:** A separate index file is created for each class. The index provides fast look up of object by object ID and transaction ID.

- **Transaction files:** A file is created for each transaction when that transaction is committed. This file identifies the exact version of each object involved in the transaction and also identifies the other transactions upon which this transaction depends.
- **Transaction log:** There is one file containing a record of each transaction in the system. The log entry for a given transaction includes the name and *state* of the transaction as well as information concerning transactions on which the given transaction depends. The major states of a transaction are *locally committed*, *remotely committed*, and *aborted*.

Off-Line Transactional Semantics and Disconnected Transactions

When transactions are run while the client is connected, locks are held in the database and the transaction runs in the traditional way. We will not describe the connected mode of operation in this paper. While the client is disconnected from the server, locks are not held in the database and the system runs in a “lazy” mode similar to that described in [GH96]. Multiple transactions can be run against objects resident in the client’s local store. When the client reconnects, the transactions are *replayed* on the server.

We perform disconnected transactions using the latest version of each object resident in the client’s local store, and save the results in the transaction log. In addition, the changed version of each object modified by the transaction is saved. When the client reconnects, the transaction log is replayed to the server and the final commit to the database is attempted. The initial execution is called a *local commit* and the replay is called a *remote commit*. Locks are granted to the client optimistically before disconnection. These locks are used to detect conflicts during remote commit.

To support conflict detection, each object has an object ID and a timestamp, which are stored in the database. The object ID is generated when the object is created. It is unique, is not modifiable, and is a key of the database table. The object timestamp is unique and is generated locally on the client at the time of commit. It consists of a unique user number concatenated with a local clock reading and a counter to distinguish among objects created during a single tick of the clock.

Gold Rush provides read locks and write locks with the usual semantics (shared read and exclusive write) [Da90]. These locks are not checked during disconnected execution, since transactions proceed in strict serial order on the client, but they are checked

during transaction replay (remote commit). It is also possible to read an object without locking it, and without checking for currency when the transaction is replayed.

Reading without locking reduces the amount of lock contention. If a transaction reads an object without locking it, the transaction can commit successfully even if the version of the object read by the transaction is obsolete at commit time. Reading without locking is useful when it is known that the attributes actually used by the application in a read object (for example, the name and serial number in an `Employee` object) are unlikely to change even though other attributes in the object (for example, the employee’s accrued vacation time) may change. Reading without locking is also useful when an approximate answer is sufficient to satisfy application requirements.

A transaction is started on the client when the application calls a `beginTransaction` method. After that other methods can be called to register particular objects with the current transaction. The application can use and modify any registered objects as it chooses and eventually call a `commit` method, closing the current transaction. When `commit` is called in disconnected mode, each modified object is written to the proper class store. A transaction file is created with references to each of the locked objects in the transaction. The transaction log is extended to include this new transaction file. At this point the transaction’s state is locally committed.

When the client eventually reconnects to the server, it serially reads all locally committed transactions and replays the transactions to the server. Each of these transactions creates an equivalent transaction on the server. The set of locked read objects and the set of write objects are checked for conflicts and the database updated if no conflicts are detected. If no conflicts are detected, the transaction succeeds, and is marked *remotely committed*.

To perform conflict detection the system tracks the following information:

- On the server, each database tuple includes the object ID and the timestamp of the last update. This timestamp is called the *last-modified time*.
- On the client, each modified object includes two timestamps. One is the *last-modified time* and the other is the *local-commit time*.

When an object is first read by a transaction, the *last-modified* timestamp is set to the timestamp of the last local commit that modified the object, or initially to the database timestamp. When the transaction commits, the *local-commit* timestamp is set and the object is written to disk. When the transaction is replayed during remote commit, the object’s *last-modified*

timestamp is compared to the database tuple's timestamp. If they are the same and the object has been modified, the object will replace the database tuple and the *last-modified* timestamp in the database will be changed to the client object's *local-commit* timestamp. If the object has not been modified, the replay proceeds to the next object in the transaction. If the *last-modified* timestamps are different for any object in the transaction, the transaction is rejected. (No global clocks are required, because timestamps are compared only for equality, not order, and each timestamp includes a user number unique to its client.)

Reducing Data Traffic Between Client and Server

When a mobile client connects to the server, the connection may be over a slow and expensive link such as a cellular phone connection. Therefore, it is important to minimize the amount of data exchanged between the client and the server, even at the cost of additional computation and additional storage requirements.

We reduce traffic between the client and server by maintaining mirrored directories of objects known to be stored on both the client and the server. There is one such directory on each client machine and one mirrored directory per client on the server machine. Each directory entry contains an object ID and a reference to a local copy of the corresponding object.

Before an object is transmitted remotely, we check whether its object ID is in the local copy of the directory. If not, we transmit the entire object and—since the object is now stored on both the client and the server—add a corresponding directory entry to each copy of the directory. If the object ID is already in the local directory, we compare the timestamp of the object referenced in the directory with the timestamp of the object to be transmitted. If the timestamps are the same, we transmit only the object ID. If the timestamps are different, we transmit both the object ID and a succinct representation of the differences between the version of the object to be transmitted and the version referenced by the directory entry.

We rely on RMI for the actual transport of data between client and server. Entities are transmitted from the server to the client only as the function result of an RMI call by the client asking for an object with a particular object ID, or as elements of the function result of an RMI call by the client asking for a vector of objects satisfying a particular SQL query. Entities are transmitted from the client to the server only as

elements of a parameter of an RMI call asking the server to commit a particular transaction.

We do not tamper with the internal mechanisms of RMI to take advantage of our mirrored directories. Rather, we use an abstract class `RemoteEntity` in place of the class `Entity` in the parameters and function results of the RMI call. This abstract class has three subclasses providing concrete implementations:

- `FullRemoteEntity`. This class carries all the information contained in an entity.
- `OidOnlyRemoteEntity`. An object of this class contains only the object ID of an entity.
- `DeltaRemoteEntity`. An object of this class contains only the object ID of a base entity and a succinct description of the changes that must be applied to the base entity to obtain the desired target entity.

The remote method for committing a transaction is called through an interface that accepts vectors of entities participating in the transaction, constructs vectors in which each entity is replaced a remote entity of the appropriate form (based on whether that entity is present in the mirrored directories), and passes these vectors to the remote method. The server converts these vectors back to their original form using its copy of the mirrored directory and performs the commit operation. Similarly, when the client calls a remote method to obtain a particular entity or vector of entities, it does so through an interface that will convert the remote entities returned by the RMI call into ordinary entities, using the client copy of the mirrored directory. The remote method itself, executed at the server, first performs the necessary database operations to construct the result entity or result vector, then constructs a corresponding remote entity or vector of remote entities based on its copy of the mirrored directory and returns that object as the result of the RMI call. (See Figure 3.)

Because we control the amount of data passed in remote calls rather than the mechanisms by which remote calls transmit their data, the fact that we are using RMI is incidental to our approach. The same approach could be used if we were to replace RMI calls with IIOP/CORBA calls. Indeed, by concentrating the bulk of our remote calls in the part of Gold Rush responsible for traffic reduction, we have encapsulated our decision to use RMI. Very little of our code would have to change if we were to decide at some point to use IIOP instead.

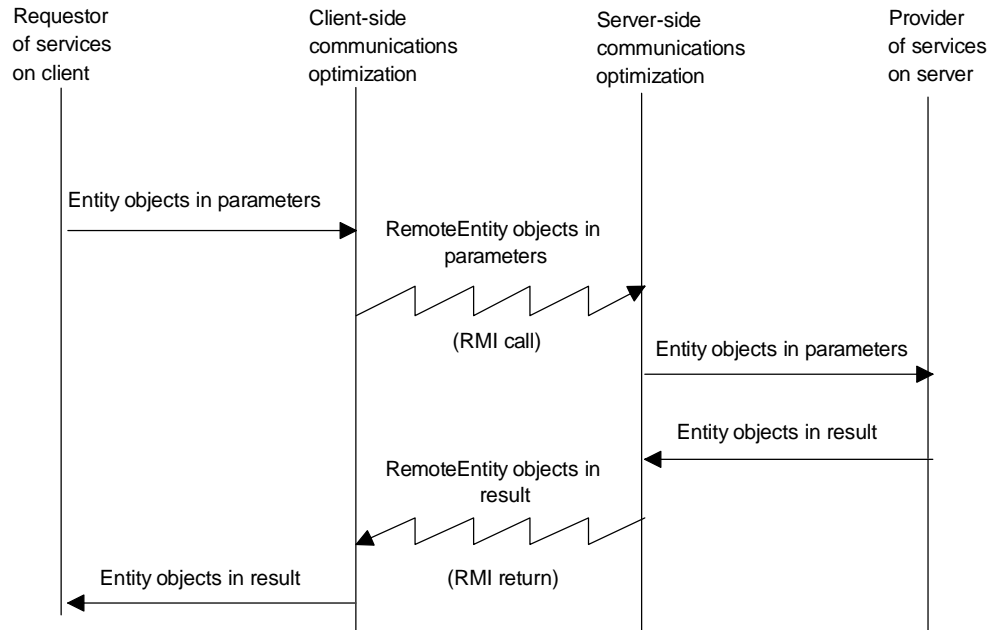


Figure 3. Protocol for optimized RMI calls.

The Role of the Application Code

A system that uses Gold Rush contains application code in both the client and the server. In the server, the application code implements the interface between Gold Rush and the persistent repository, typically a relational database. For this interface the application supplies definitions of the objects and methods for:

- querying the attributes of a single object;
- setting the attributes of a single object; and
- retrieving of a set of objects, all of the same class, satisfying a query clause.

In addition, the application must supply methods to resolve conflicts. We also envision that the application code at the server could run more complex requests, perhaps activating an agent to execute autonomously on behalf of the client, and perhaps yielding a result consisting of objects of various classes.

At the client, Gold Rush provides methods for transaction start, commit and rollback, object creation, and lock upgrade. There are also methods for a connected client to create collections of objects of a given class satisfying a given query. Finally, there are primitives for associating names with such collections and making the collections persistent at the client. The application must supply methods to navigate between related objects.

Future Directions

We have implemented a prototype version of Gold Rush. Gold Rush is now being considered for integration into a large business-object framework. Its interfaces were designed to facilitate this integration. Issues that we did not address in the prototype but are important in a production system, such as data security, would be addressed during this integration.

We are aware of a number of ways in which we can improve the performance and flexibility of Gold Rush. We expect to incorporate these improvements in our future work.

Currently, the client is notified when a conflict is detected during an attempt to perform a remote commit. However, any actions to recover from the conflict, for example by merging updates or retrying a transaction with fresh data, must be programmed explicitly. Future versions of Gold Rush will include a framework for specifying conflict-resolution strategies. It will be possible to specify strategies both on a class-by-class basis and on a scenario-by-scenario basis.

Our current mapping between relational data bases and objects entails the addition of columns to the relational database tables to hold object ID and time-stamp data for use in conflict detection. This is inconvenient and sometimes unacceptable when dealing with legacy data bases. We are formulating techniques that will allow access to legacy data bases without changing the format of those data bases.

The current system requires that all data be preloaded before disconnection. This requires careful planning by the user to avoid being stranded without the data needed to complete one's work. We are investigating techniques to dynamically connect to the server and fetch missing data.

Our prototype presumes that the client middleware is invoked by only one application at a time, and that all of the application's invocations of middleware methods are performed by a single thread. Thus there is no concurrency control in the local-commit logic. This is not an inherent limitation in our approach, merely a simplifying assumption made for the prototype version. We plan to rewrite the client middleware to make it thread-safe, so that the mobile worker can run several applications at a time and so that a client application programmer can take advantage of Java threads.

Our current mobile object server utilizes one active remote-object-server object for every client regardless of whether the client is connected or disconnected. This naive approach supports high concurrency but requires large number of thread resources, potentially over a long period of time with huge number of clients. We plan to investigate simple activation approaches such as the one described by Wollrath [WW95] to reuse remote-object-server objects if possible and to activate and deactivate persistent remote objects with low overhead.

Several of our strategies for reducing traffic over a slow or expensive link entail a large amount of computation. Over a sufficiently slow link, the time saved by reducing traffic more than makes up for the time expended to perform the computation. However, on occasions when the client is connected to the server over a fast and inexpensive link (as when a mobile user returns to the office and connects the client machine directly to a LAN), the time saved by reducing traffic is negligible, and the computational cost is no longer worthwhile. Therefore, we plan to provide controls for disabling our computationally intensive traffic-reduction strategies. Ultimately, it may be possible to monitor the client-server connection and switch modes automatically based on the speed and cost of the connection and the user's current level of urgency (measured as the amount of extra money the user is willing to pay to speed up transmission).

In our present architecture, all application-specific algorithms (except for the formulaic methods that our tools generate to translate between object-oriented and relational data bases) reside on the client. These algorithms communicate with the server through simple-minded requests to fetch an object with a given ID, to fetch a collection of objects satisfying a given

query, or to commit a transaction remotely. Some algorithms (executable only in connected mode) may involve an extended dialogue between the client and server, in which the client requests some data and, based on the contents of that data, issues further requests. In such cases, traffic between the client and server could be substantially reduced by allowing the client to issue high-level application-defined requests to the server. These requests would invoke application-specific algorithms at the server and deliver results to the client. A server-based algorithm might entail a long series of database queries and updates, but these would all be performed locally on the server. Only the initial high-level request and the final result would have to be communicated, producing substantial savings when the connection is over a slow or expensive link. The server-based algorithm could even be executed autonomously by an agent acting on behalf of a disconnected client. The client would retrieve the result of the autonomous computation upon reconnection.

Conclusion

Mobile applications in Java can easily be ported to other platforms, and can exploit Java's strong support for distributed applications. Gold Rush allows mobile client code written in Java to access data stored in enterprise relational data bases. These applications deal with Java objects corresponding to rows of relational database tables, belonging to classes that correspond to tables. Attributes of these objects reflect the 1-1, 1- n , and m - n relationships among relational-database entities. We have developed tools that automatically generate the required Java classes and translate between the object and relational views of the data.

Unlike other systems allowing relational-database entities to be manipulated as Java objects, Gold Rush allows users to cache objects off-line on the mobile client and then disconnect, obviating the need for a continual, potentially expensive, link to a central server. A client also has the option of running in disconnected mode when a slow link is available, to avoid communication delays. Unlike systems that replicate a subset of a relational database on the client, our architecture confines all manipulation of relational data bases to central servers. The client deals purely with Java objects and can remain lightweight, using the same object interface for both connected and disconnected transactions.

To guarantee atomicity of updates and the integrity of both the central database and the data stores on individual clients, we group updates into transactions.

While the client is connected, transactions can be run directly on the server. While the client is disconnected, transactions are constructed and saved locally on the client and replayed to the server upon reconnection. Objects participating in transactions can be locked optimistically, which allows other clients, or back-office applications, to use the same data, but makes it necessary to check for conflicts when client transactions are replayed to the server. In case of conflict, the central database is not updated and the client is notified of the failure. A conflict-resolution mechanism currently under design will allow the client to take appropriate actions to recover from the rejection of a transaction that had been tentatively committed.

In addition to reducing the need for communication between client and server to the initial caching of objects and the replaying of transactions, we streamline those communications that are necessary. By reducing the amount of data that must be transmitted, we make the use of mobile clients more economical and practical.

Acknowledgement

We thank Professor I-Chen Wu of the National Chao-Tung University at Taiwan for his help with the persistent client store.

References

- [AN93] Adly, N., Nagi, M., and Bacon, J. A hierarchical asynchronous replication protocol for large scale systems. Proceedings, 1993 IEEE Workshop on Advances in Parallel and Distributed Systems, Princeton, New Jersey, October 6, 1993. IEEE Computer Society Press, Los Alamitos, California, 1993, 152-157
- [Da90] Date, C.J. *An Introduction to Database Systems*. Addison-Wesley, Reading, Massachusetts, 1990
- [Fr96] Froemming, G. Moving forward with replication. *DBMS* **9**, No. 4 (April 1996), 83-84+
- [GH96] Gray, J., Helland, P., O'Neil, P., and Shasha, D. The dangers of replication and a solution. Proceedings, 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Canada, June 4-6, 1996. *SIGMOD Record* **25**, No. 2 (June 1996), 173-182
- [IB95] IBM Corporation. *An Introduction to DataPropagator Relational, Release 2*, 1995
- [IB97] IBM Corporation. VisualAge family web development directions, 2nd ed.. <http://www.software.ibm.com/ad/vajava/wp.htm>, March 1996
- [Ja97] JavaSoft. The JDBC(tm) database access API. <http://splash.javasoft.com/jdbc/>, February 4, 1997
- [KI92] Kistler, J.J., and Satyanarayanan, M. Disconnected operation in the Coda file system, *ACM Transactions on Computer Systems* **10**, No. 1 (February 1992)
- [La96] Lardear, J. Taking transactions off-line: Tactita's Caprera transaction monitor demonstrates emerging transaction-processing model. *MIDRANGE System* **9**, No. 12 (August 1996), 18
- [Lo96] Lotus Notes Release 4.5: A Developer's Handbook, IBM Corporation
- [LS94] Lu, Q., and Satyanarayanan, M. Isolation-only transactions for mobile computing. *ACM Operating Systems Review* **28**, No. 2 (April 1994)
- [Or95] Oracle Corporation. Oracle 7 distributed database technology and symmetric replication, April 1995
- [Pi96] Pitoura, E. A replication schema to support weak connectivity in mobile information systems. Database and Expert Systems Applications: 7th International Conference, DEXA '96 Proceedings, Zurich, Switzerland, September 9-13 1996. Springer-Verlag, Berlin, 1996, 510-520
- [RZ96] Rasheed, A., and Zaslavsky, A. Ensuring database availability in dynamically changing mobile computing environment. Proceedings, ADC'96: Seventh Australasian Database Conference, Melbourne, Australia, January 29-30, 1996. Australian Computer Science Communications **18**, No. 2 (1996), 100-108
- [Su96] Sun Microsystems. RMI—remote method invocation. <http://java.sun.com/products/JDK/1.1/docs/guide/rmi/>, 1996

- [Sy96] Sybase Corporation. SQL Remote: replication anywhere, 1996
- [Sy97] Symantec Corporation. Evaluating network database architecture.
http://www.symantec.com/dba/wp_evalnda.html, January 28, 1997
- [Wo95] Wolfson, O. Mobile computing in a reservation application. Information and Communication Technologies in Tourism: Proceedings of the International Conference, Innsbruck, Austria, January 18-20, 1995. Springer-Verlag, Vienna, 1995, 43-45
- [WW95] Wollrath, Ann, Wyant, G., and Waldo, J. Simple activation for distributed objects. Conference on Object-Oriented Technologies, June 26-29, 1995, 1-11
- [YT96] Yoshida, T., and Takizawa, M. Model of mobile objects. Database and Expert Systems Applications: 7th International Conference, DEXA '96 Proceedings, Zurich, Switzerland, September 9-13 1996. Springer-Verlag, Berlin, 1996, 623-632
- [YW94] Yixiu, Huang, and Wolfson, O. Object allocation in distributed databases and mobile computers. Proceedings of the 1994 IEEE 10th International Conference on Data Engineering, Houston, Texas, February 14-18, 1994. IEEE Computer Society Press, Los Alamitos, California, 1994, 20-29
- [ZF96] Zaslavsky, A., Faiz, M., Srinivasan, B., Rasheed, A., and Lai, S. Primary copy method and its modifications for database replication in distributed mobile computing environment. Proceedings, 15th Symposium on Reliable Distributed Systems, Niagara-on-the-Lake, Ontario, Canada, October 23-25, 1996. IEEE Computer Society Press, Los Alamitos, California, 1996, 178-187