RC23347 (W0409-145) September 22, 2004
Computer Science

IBM Research Report

Programming Model Alternatives for Disconnected Business Applications

Avraham Leff, James Rayfield
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Programming Model Alternatives for Disconnected Business Applications

Avraham Leff
IBM T. J. Watson Research Center
P. O. Box 704
Yorktown Heights
NY 10598
avraham@us.ibm.com

James Rayfield
IBM T. J. Watson Research Center
P. O. Box 704
Yorktown Heights
NY 10598
jtray@us.ibm.com

Abstract

We discuss some of the challenges that are inherent to programming business applications for disconnected environments, and outline various programming models that address these challenges. We focus especially on a transparent programming model that uses a "log/replay" approach, and describe our experiences in implementing this programming model as middleware that supports disconnected Enterprise JavaBeans applications.

1 Introduction

This paper describes an ongoing project to build middleware that supports a transparent programming model for building disconnected business applications. To explain why this is useful, we first define "disconnected business applications" and show how technology trends are increasingly motivating the creation of such applications. We then explain the challenges that are specific to "disconnected" in contrast to "connected" business applications, and describe three programming models that attempt to address these challenges. Although there are tradeoffs between these programming models, we explain why the *log/replay* approach is superior. The rest of the paper focuses on our ***dEJB*** (**d**isconnected **E**nterprise **J**ava**B**eans) middleware, which is designed to transparently support the log/replay programming model for disconnected business applications.

1.1 Disconnected Business Applications

A "business" application is characterized by the fact that the application (1) involves updates to state that is shared by multiple users and (2) these updates must be performed transactionally [6]. A business application is "disconnected" if it executes on devices that operate, at least part of the time, without being able to interact with the server's shared database. Mobile devices such as personal digital assistants (PDAs), PDA phones, hand-held computers, and laptop computers are examples of such disconnected devices. Historically, resource constraints (e.g., memory and CPU) have precluded disconnected devices from running business applications. Ongoing technology trends, however, imply that such resource constraints are disappearing. For example, DB2 Everyplace [3] (a relational database) and WebSphere MQ Everyplace [11] (a secure and dependable messaging system) run on a wide variety of platforms such as PocketPC™, PalmOS™, QNX™, and Linux; they are also compatible with J2ME [7] configurations/profiles such as CDC and Foundation. It seems likely that even an Enterprise JavaBeans [4] container can run on mobile devices. As a result, business applications that previously required the resources of an "always connected" desktop computer can potentially run on a mobile device. This paper therefore examines a key issue: what are the implications (if any) of the fact that such devices are only intermittently connected to the server.

Business applications, almost by definition, are structured as application logic that reads from, and writes to, a transactional database that can be concurrently accessed by other applications. Business applications have taken for granted that the transactional database can always be accessed by the application. Even if business applications are structured so as to access locally cached data for "read" application, state changes ("updates") must still applied to the shared, master database [5] [9]. In contrast, disconnected business applications are forced to read from, and write to, a database that is *not* shared by other applications and users. This observation raises the possibility that business applications must be fundamentally restructured in order for them to run in a disconnected environment.

There are three basic programming models that address this fundamental characteristic of disconnected business applications: *Messaging*, *Row-level replication* and

Log/Replay.

2 Programming Model Alternatives

In this section we discuss the various programming-model alternatives for disconnected client applications. We focus on the client-side execution and client-to-server replication of updates. Suitable technologies already exist for server-to-client database-subset replication [12], and the log/replay programming model does not improve upon these.

2.1 Messaging

The messaging programming model takes the approach that a disconnected business application must be explictly partioned into two portions. Developers explictly code one portion to execute on the disconnected device and another portion to execute on the server. On a per-application basis, developers devise a suite of messages that are sent by the mobile device to the server when it reconnects. Upon receiving these messages, the server invokes programs that propagate the disconnected application's changes to the server's database. For example, the portion of an *order entry* application that runs on a disconnected device is responsible for saving enough of the new order information to allow the server to update its database as if the order had been placed by a server-side application program. This might include the name of the agent executing the order, the customer for whom the order is executed, and the set of items in the order. The message suite transmits this state and invokes a server-side program that executes the order on the server using the state that was previously saved on the disconnected device.

The messaging approach has the advantage of (potentially) minimizing the required bandwidth needed to propagate the device's state to the server. This is possible because only the minimum amount of state needed to invoke the server-side program need be transmitted. On the other hand, the messaging approach has important disadvantages. Developers must "hand-craft" a two-part solution (client and server) on a per-application basis. For example, the application itself is responsible for transactionally constructing and transmitting the message from the device to the server, processing the message on the server, invoking the program that executes the order on the server, and returning the results to the reconnected device. The messaging approach, in this sense, is a step backward from the historical trend in software to push as much function as possible into generic middleware. It is also a distraction to the developers, who would be more productive if they were able to focus their efforts solely on the user interface and business logic.

From the standpoint of productivity, as well, businesses would prefer to develop only one version of an application, and deploy that application to both connected and disconnected environments. The messaging approach usually requires that two versions of an application must be developed: the *partitioned* version of the application, described above, and a *connected* version, for machines which are always connected to the server. Thus the partitioned version requires additional development, test, and maintenance effort beyond that required for the standard connected version.

The messaging approach also has difficulty enabling the disconnected device to see locally-applied state changes — that is, state changes made by the application to the cached database. This is because the straightforward implementation of the messaging approach does not actually make changes to the local database; instead, they are saved for eventual transmission to the server. Applying the changes locally complicates the implementation because the changes must be transactionally merged with the updated server state after the server has executed the application messages.

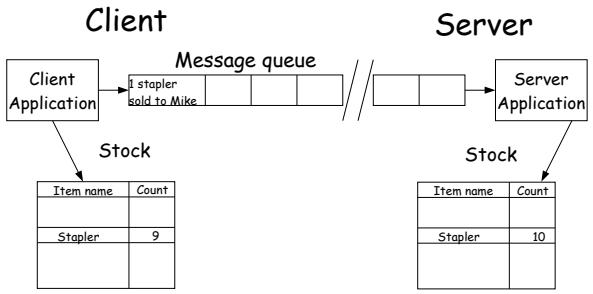

Figure 1. Messaging Programming Model

Figure 1 sketches the use of messaging programming model to implement an "order entry" application for a disconnected environment. It shows the client portion of the application as having decremented the stock level of staplers because one was sold to Mike; it also shows the subsequent message to the server, instructing the server to decrement its stock level so as to process the customer's order on the server.

2.2 Row-Level Replication

In contrast to the messaging programming model, the row-level replication programming model for disconnected business applications takes the approach that the client application reads and modifies the local copy of the database without worrying (directly) about concurrent operations by other clients, and without keeping a separate record of the changes it makes. Instead, middleware running on the disconnected device tracks which database rows have been modified. Upon reconnecting to the server, all modified rows are replicated to the server, thus propagating the application's disconnected activity. One advantage of this approach is that the client application closely resembles a standard server-side (connected) application. Another advantage is that it takes a "middleware" approach in which the fact that the application executes on a disconnected device is (largely) transparent to the application and to its developers.

However, the row-level replication approach has disadvantages of its own. For example, some business logic may be too expensive to run on disconnected clients (e.g. address validation), or may require access to external systems (e.g. credit-card verification). The business-logic for these kind of operations must essentially queue this work somehow to be done later by the server. Other problems arise from the limitations of various row-replication implementations. Typically, unmodified rows are not compared at replication time against the current server state (i.e. only write-write conflicts are detected). If the server state was modified while the client was disconnected, the client may have made decisions based on stale data. Also, many implementations do not record the transactional boundaries of the client application execution or the order of the row updates which were made. If any replication conflicts are detected, the replication session must be manually corrected or discarded.

Row-level replication often introduces false conflicts at replication time. For example, the middleware must detect a conflict if a modified row has also been modified on the server while the client was disconnected, because this could lead to a transaction serializability violation (lost-write [6]). In the order-entry example, two different clients may decrement the stock level for the same item. At replication time the middleware sees this as a conflict, because two different clients modified the same row of the database. Although this false conflict could easily be resolved by humans (decrement by 2), row-replication middleware does not have enough semantic information about the application and database to resolve this automatically. The semantics (forcibly) introduced by the row-replication middleware are that the stock level upon reconnect must be identical to the stock level at disconnect. However, the desired semantics are merely that sufficient stock exists at reconnect to satisfy the order.

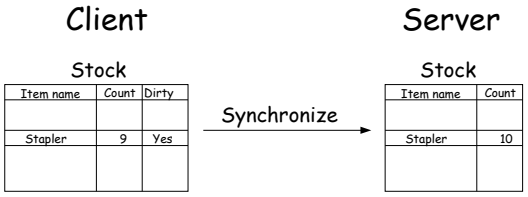

Figure 2. Row-Level Replication Programming Model

Figure 2 sketches the use of the row-level replication programming model to implement the same scenario shown by Figure 1.

2.3 Log/Replay

The log/replay programming model for disconnected business applications first *logs* an application's activity while the device is disconnected and then *replays* the previously logged activity when the device reconnects with the server. We introduce this programming model because it offers the advantages of both the messaging approach (replication is based on the application's business logic) and the row-level replication approach (the application and its developers are unaware that it may execute on a disconnected device). Note that we do not log database state changes, as this would be equivalent to the row-level replication approach. Instead, we log the activity performed by the application itself, or more precisely, we log (and replay) at business transaction granularity.

The log/replay approach can be viewed as a middleware-based version of the messaging programming model. As with the messaging approach, the disconnected application tracks the key business activities that have occurred during the application's execution. Unlike the messaging approach, middleware is responsible for tracking these business activities; the application itself is unmodified, and remains unaware that log activity is occuring. As with other comparisons between hand-crafted and automated solutions, the messaging approach may well (at least initially)

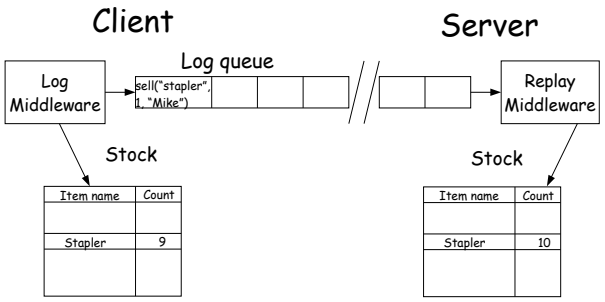

Figure 3. Log/Replay Programming Model

provide a more optimal solution than the log/replay approach. For example, a hand-crafted solution will record the minimal state needed to invoke the server-side part of the application. The usual tradeoff applies, however: development and maintenance costs are considerably cheaper with an automated approach. In the next section we describe how the **dEJB** system provides middleware that supports the log/replay programming model.

Figure 3 sketches the use of the log/replay programming model to implement the same scenario shown by Figures 1 and 2.

3 dEJB Middleware

We are currently working on the **dEJB** project, middleware for implementing the log/replay programming model for disconnected business applications that are coded as Enterprise JavaBeans[4] (EJBs). EJBs are a component model for enterprise applications. EJBs automatically supply common requirements of enterprise applications such as persistence, concurrency, transactional integrity, and security. Bean developers focus on the business logic of their application; when deployed to an EJB *container*, the components are embedded in an infrastructure that automatically supplies the above requirements. The chief design goals of dEJB are that the log/replay function (1) be application-independent and (2) be as transparent as possible. That is, our goal is to enable a business application developed for a connected environment to be deployed to a disconnected environment with no (or few) changes. Although focussing on EJBs, the algorithms and infrastructure used in dEJB apply to other transactional component models[8] such as CORBA[1] and DCOM[2].

dEJB requires that the business logic be implemented as a set of stateless session beans (SSBs), representing the "business tasks" of the application Our middleware logs the top-level SSB methods executed by the application (that is, those methods which initiate a transaction context).

3.1 Infrastructure

Figure 4 shows the client/server dEJB infrastructure. It consists of a server that maintains a shared database and that executes one or more business applications. Each of these applications consists of EJBs that have been deployed to an EJB container using the dEJB tooling. Note that the business application's EJBs are completely standard; it is the dEJB tooling that adds the hooks to the dEJB runtime that implement the log/replay programming model.

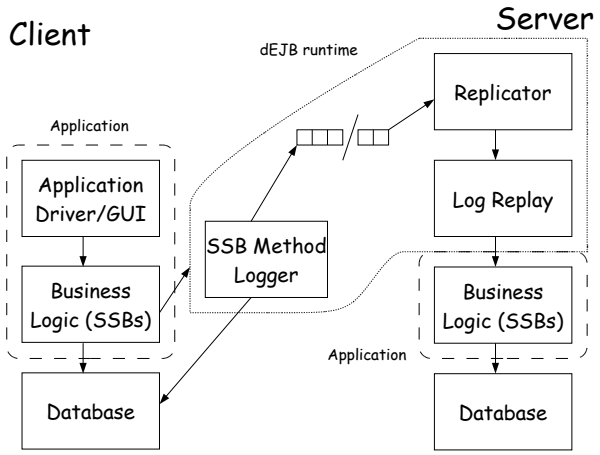

Figure 4. dEJB: Client/Server Infrastructure

Multiple clients are intermittently connected to the server in Figure 4, and each client can run any of the business applications that are running on the server. The two most important EJBs of the dEJB runtime are the *LogRecord* and *Replicator* beans.

The dEJB tooling extends tooling that deploys stateless session beans to a "connected" EJB container in the following way. Wherever the deployed component (e.g., the class that implements *EJBObject*) delegates a client invocation to the bean implementation (e.g., the class that implements *SessionBean*), the dEJB tooling injects a call to a utility method which creates the appropriate LogRecord bean.

3.1.1 LogRecord

Every time that a stateless session bean (*SSB*) method is invoked, the (deployed) code determines whether the method

is executing within an existing transactional scope. If it is, there is no need to log this method's execution since its activity is logged as part of the existing transaction's activity. If no transaction is currently active, a corresponding LogRecord entity bean is created that contains the following state:

- The name of the method being logged.
- The signature of the method being logged.
- The method parameter values used during the (logged) method invocation.
- The JNDI name of the Home which created the SSB

As an application executes on the disconnected device, a set of unique LogRecord instances are created, each corresponding to a single top-level business transaction. At replay time, the LogRecord contents provide enough information to create a new instance of the SSB and re-execute the same method with the same parameter values on the server.

3.1.2 ReplicatorBean

The ReplicatorBean SSB is responsible for the synchronization process that is initiated when a disconnected device synchronizes, at some point, with the server. The replay protocol consists of three phases:

1. The replay protocol is initiated when a client invokes `initiateSync` on the remote Replicator stub. This method specifies a *syncSessionId* that groups the set of LogRecords that are passed to the server.
2. The Replicator SSB implementation on the server iterates over the set of LogRecords, invoking `replay` on each instance. Note that the original parameter values are used when replaying the method.

   A replay is considered to have "failed" if an exception is thrown by the method or (optionally) if the value returned by the method differs from the original value.
3. The server asynchronously contacts the client, and returns the status of the replay operations.

4 Programming Model Semantics

The row-level replication programming model differs significantly from the messaging and log/replay programming models with respect to the *intended* semantics of a synchronization operation. Row-level replication assumes that operations performed by a disconnected client should be considered "committed", and should be transformed "as is" to the server. For example, if an order was placed when items cost $100, the synchronized device will place orders using the $100 price. In contrast, messaging (typically) and log/replay assume that operations performed by a disconnected client are "tentative", and are only as a statement of what the client intends to do when it synchronizes with the server. In the previous example, if the price has changed to $150, the order that gets committed will use that price rather than the $100 price that was used to place the original order.

Neither of these semantics is obviously "right" or "wrong". On the one hand, clients that have run an application without any problems likely assume that their activity is valid and will be committed to the server without incident, and "as is". This can only occur with row-level replication semantics. On the other hand, clients likely assume that synchronization activities execute against the most up-to-date server-side data; this can only occur with the log/replay semantics.

Independent of this basic issue, the log/replay programming model has the advantage of defining valid synchronizations in terms of the application developer's definition rather than in terms of detected data conflicts. Recall that a replay fails when the replayed method throws a (business) exception. In contrast to row-level replication, this will tend to reduce "false positives" as, for example, orders will be rejected only because of insufficient stock rather than conflicts between the stock levels. This will also tend to reduce "false negatives", e.g., a client's desire to place an order only if the price is less than a threshold will not be re-evaluated during synchronization under row-level replication approach since it is a only a function of "read" data. Under log/replay the entire business validation logic will be replayed so that if the threshold constraint is violated at synchronization time, the replay will be rejected.

4.1 Status

We have implemented the dEJB runtime and tooling shown in Figure 4 and deployed a demonstration "order entry" application to that platform. The "order entry" application can be viewed as a "TPC-C lite" application. The application consists of *Agent*, *Customer*, *Item*, *Order*, *OrderLine*, and *Stock* entity EJBs, as well as a *ManageOrder* stateless session bean. These EJBs are standard EJB 2.0 beans, and are not modified in any way so as to enable deployment to the disconnected environment.

Both the client and server portions of the demo execute on the IBM Service Management Framework [13] (SMF), an implementation of the Open Service Gateway initiative [10] (OSGi) framework. OSGi is a specification for lightweight (J2ME-compatible), Java-based containers to which dynamic components are deployed. The container handles interactions between components, provides basic services to applications (e.g., web-services, authentication,

logging), and allows applications to be exposed as services to other applications. Applications must implement the suite of OSGi life-cycle methods: this allows the container to install, start, stop, update, and delete any application programmatically or through an interactive console.

The server portion of the demo uses DB2 as its database, and executes in the J2SE (standard edition) environment. The clients use DB2e as their database and execute in the J2ME environment, specifically using the *jclRM* Java class libraries. The demo runs both on the server in "always connected" mode, and in "intermittently connected" mode in which the client runs the application while disconnected and subsequentally synchronizes with the server. No changes are made to the application code in switching between the two modes.

References

[1] J. Siegel. Quick CORBA 3. John Wiley & Sons, 2001.

[2] F. E. Redmond. DCOM: Microsoft Distributed Component Object Model. John Wiley & Sons 1997.

[3] IBM DB2 Everyplace.
http://www-306.ibm.com/software/data/db2/everyplace/index.html

[4] J2EE Enterprise JavaBeans Technology.
http://java.sun.com/products/ejb/

[5] M.J. Franklin, M.J. Carey, M. Livny. Transactional Client-Server Cache Consistency: Alternatives and Performance. ACM Transactions on Database Systems (TODS), Volume 22 , Issue 3, 315 - 363, 1997.

[6] J. Gray. A. Reuter. Transaction Processing: Concepts and Techniques. Morgan Kaufmann. 1993.

[7] Java 2 Platform, Micro Edition (J2ME).
http://java.sun.com/j2me/index.jsp

[8] A. Leff, P. Prokopek, J. T. Rayfield, and I. Silva-Lepe. Enterprise JavaBeans and Microsoft Transaction Server: Frameworks for Distributed Enterprise Components. Advances in Computers, Academic Press. Vol. 54. 2001. 99-152.

[9] A. Leff and J. T. Rayfield. Improving Application Throughput with Enterprise JavaBeans Caching. May 2003. 23rd International Conference on Distributed Computing Systems.

[10] Open Services Gateway Initiative
http://www.osgi.org/

[11] IBM WebSphere MQ Everyplace.
http://www-306.ibm.com/software/integration/wmqe/

[12] Open Mobile Alliance (OMA), SyncML
http://www.openmobilealliance.org/tech/affiliates/syncml/syncmlindex.ht

[13] IBM Service Management Framework
http://www-306.ibm.com/software/wireless/smf/