RC23561 (W0503-057) March 9, 2005
Computer Science

IBM Research Report

Projecting a Connected Programming-Model for Business Applications onto Disconnected Devices: A Practical Approach

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

Projecting a Connected Programming-Model for Business Applications Onto Disconnected Devices: a Practical Approach

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

Programming models usefully structure the way that programmers approach problems and develop applications. Business applications need properties such as persistence, data sharing, transactions, and security, and various programming models exist – for connected environments – that facilitate the development of applications with these properties. However, as developers consider how business applications should run on disconnected devices, we must consider ways to implement these properties in such an environment. In this paper we present a practical approach for projecting a connected programming model onto disconnected devices, explain the advantages of this approach, and show how the connected programming model is useful even in a disconnected environment.

1 Introduction

1.1 Programming Models for Business Applications

A business application is characterized by the fact that the application (1) updates state that is shared by multiple users; (2) must perform these updates transactionally [8] to a shared database; and (3) must operate securely. Business applications therefore have requirements that other applications do not: chiefly, to access persistent datastores securely and transactionally. Programming models can ease the difficulty of developing complex business logic that meets these requirements. This is typically done by abstracting business application requirements as generic services or middleware that the developer can access in as unobtrusive a manner as possible. Good programming models enable a "separation of concerns" through which the application developer can concentrate on the application-specific logic, and assume that the deployed application will meet the business application requirements. Well-known examples of such programming models include CORBA [1], DCOM [2], and Enterprise JavaBeans (EJBs) [5].

1.2 Business Applications on Disconnected Devices

Business applications have traditionally been deployed in *connected* environments in which the shared database can always be accessed by the application. In contrast, when applications are deployed to mobile devices such as personal digital assistants (PDAs), hand-held computers, and laptop computers, these devices are only intermittently able to interact with the shared database. (In a client/server environment, the shared database resides on the server.) 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 [18] (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 [13] configurations/profiles such as CDC and Foundation. It seems likely that mobile devices will even be able to host middleware such as an Enterprise JavaBeans container. As a result, business applications that previously required the resources of an "always connected" desktop computer can potentially run on a disconnected device.

However, resource constraints are not the only issue precluding the deployment of business applications on disconnected devices. Fundamental algorithmic and infrastructure problems must also be solved. The algorithmic problems

stem from the fact that the application executes while disconnected from the server, but the work performed must later be propagated to the server. To see why this is so, consider the fact that business applications, by our definition, are structured as application logic that reads from, and writes to, a transactional database that can be concurrently accessed by other applications. Connected business applications have taken for granted that the transactional database can always be accessed by the application. Even if they are structured so as to access locally cached data for "read" operations, state changes ("updates") must still applied to the shared, master database [6] [15] at the completion of each user operation. Obviously, the shared-database assumption does not hold when business applications are disconnected: they are then forced to read from, and write to, a database that is *not* shared by other applications and users. Also, almost inevitably, the disconnected application will execute against data that is out-of-date with respect to the server's version of the data. How can work performed on the disconnected device be merged into the shared database in a manner that preserves the transactional behavior of both disconnected and connected clients? The lock-based concurrency control mechanisms used in connected environments are not suitable for disconnectable applications because they unacceptably reduce database availability. Also, lock-based concurrency control is simply not dynamic enough; it is typically impossible to know what needs to be locked before the device disconnects from the server.

Infrastructure must also be developed to deal with the lifecycle of an application deployed to a disconnected device. Data must first be "checked out" from the shared database; the data are then used by the application; and the committed work must be merged into the server database when the device reconnects. Without middleware that provides replication (from the server to the device) and synchronization (from the device to the server) functions, each application must provide its own implementation of these features.

A programming model is therefore needed to facilitate business application development for disconnected devices. It must provide constructs that address these algorithmic issues, and must integrate with middleware that provides the services described above. This paper considers whether the projection of an existing programming model, with the appropriate middleware runtime, is sufficient to develop and deploy useful business applications to disconnected devices.

1.3 Projecting a Programming Model

We explain in this paper how the Enterprise JavaBeans [5] programming model can be projected onto disconnected devices. By "projection", we acknowledge explicitly that our approach does require that developers be aware that the application will be deployed in a disconnected environment. However, the programming model enables application semantics that are identical, or similar, to the connected programming model. The algorithmic issues discussed above are solved in a way that is transparent to the developer, who does not have to write more (or different) code than she does for the connected environment. Further, we provide middleware that meets the basic infrastructure requirements.

1.4 Technical Contribution

The implications of disconnection for transactional applications are well known (see [23] for a recent survey of the area of "mobile transactions"). Much work has been done in the area of transactionally synchronizing work performed on a disconnected client to the server. One contribution of our work is that we show how a mature programming model can be projected to disconnected devices in a way that takes advantage of this algorithmic work. This is important because (1) developers can use their existing programming model experience to develop disconnected applications and (2) differences between the connected and disconnected versions of an application are greatly reduced. We also demonstrate that our approach is "practical" in the sense that useful work can be performed on the disconnected device (i.e., few constraints are imposed), while minimizing the likelihood of problems arising during synchronization. Our approach is also practical in the sense that we have built middleware that concretely realizes the programming model on disconnected devices.

1.5 Paper Structure

The paper is structured as follows. In Section 2, we discuss how this programming model approach relates to work in the area of mobile transactions. Section 3 explains how the programming model must support the lifecycle of a generic disconnected application; Section 4 introduces two synchronization techniques that we considered using to support the programming model. In Section 5 we explain why our combination of this programming model and middleware is a practical approach for building disconnected applications.

2 Related Work

Our paper focuses on how a connected programming model can be projected to disconnected devices. This part of our work is closely related to the area of mobile transactions [23]. However, our projected programming model and synchronization algorithm (see Section 4) is much simpler

than many proposed in the mobile transactions literature. We assert that this simpler approach is sufficient for several reasons.

First, we address a programming model for environments that are more robust than is typically assumed for mobile transactions. Much mobile transactions research, for example, assumes that these transactions execute in resource constrained environments. They must thererefore address issues related to limited bandwidth capacity, communication costs, and energy consumption. In contrast, we assume that business applications are deployed to (the increasingly more powerful) devices that can locally execute business applications against a transactional database.

Second, we assume that transactions are able to execute *entirely* on the disconnected device, without assistance from a server. This allows considerable simplification compared to the mobile transactions work designed to support transaction processing in which a client may initiate transactions on servers or may distribute transactions among the mobile client device and servers. Such environments require that transaction processing be supported while the mobile device moves from one networked cell to another or drops its network connections. Mobile transaction models such as *Kangaroo Transactions* [4] are explicitly designed to operate in such complex environments, whereas the synchronization techniques discussed here can use traditional transaction semantics. Similarly, the synchronization techniques discussed here do not deal with distributed transactions (between the mobile device and the network), nor do they deal with heterogeneous multi-database systems. Our focus, instead, is to jump-start deployment of business applications to disconnected devices in well-controlled environments.

Finally, we *do* disagree with the assumption made by some research that optimistic (non-locking) concurrency control mechanisms must perform badly for the long disconnect durations typical of mobile transactions. Such research assumes that the classic optimistic algorithms [8] perform well only for short disconnections, and will experience unacceptable abort ratios for long disconnections. Non-traditional transaction models such as *pre-write operations* [17] and *dynamic object clustering* replication schemes [21] are designed to increase concurrency by avoiding such aborts. In our experience, however, business processes greatly reduce the actual occurrence of such aborts by implictly partitioning data among application users. Furthermore, the transform-based approach used by method replay synchronization (Section 4) is designed to reduce the size of a transactional footprint, and thus reduces the probability of aborts during synchronization.

Our synchronization work builds on earlier work using log-replay in support of long-running transactions [19]. These ideas are similar to the approach taken by the IceCube[9] system, although IceCube does not focus on transactional applications.

3 Lifecycle of a Disconnected Business Application

In order to be successful, a programming model for disconnected devices must be compatible with the lifecycle of a disconnected application.

Deployment of a disconnectable business application requires that an administrator perform a one-time setup (lifecycle stage **0**) of the mobile device's database(s). The key challenge in stage **0** is to replicate sufficient data (from the server to the device) such that the application can execute correctly. This can be a difficult task when an application can potentially access a data set that is too large to fit on the device. In such cases, application administrators must determine the subset of data that will actually be used by the application, and replicate that subset to the device. This is often done through *ad-hoc*, but effective, business rules. For example, a salesman does not need to have the entire set of customer data replicated; only the set of customers in her district is typically needed.

After this initial setup is performed, the mobile device repeatedly executes the following lifecycle:

Stage 1 (Propagate server updates): Before disconnecting, the server's updates are propagated to the device.

Stage 2 (Execution): User executes one or more business applications on the disconnected device.

Stage 3 (Propagate client updates): Device reconnects to server, and propagates its updates to the server-side database.

Stages **0** and **1** benefit from middleware that allows devices to define the subset of relevant data that needs to be copied to the device. The subset is typically expressed as a query or set of queries applied to the server database.

Stage **1** ensures that the device's database is as up-to-date as possible before beginning disconnected execution. It greatly benefits from middleware that:

- *Subscribes* to relevant changes on the server (i.e. those changes which fall within the defined subset of relevant data).
- *Propagates* server changes to the client device database, with the result that the device's data is now up-to-date with respect to the server.

Standardized protocols such as SyncML [25] can be used to pass data between the device and the server – across wireless and wired networks and over multiple transport protocols – using the standard representation format defined in

the SyncML *Representation* protocol. The SyncML Synchronization protocol efficiently replicates server-side data to the device, by doing either a "one-way sync from server only" or a "Refresh sync from server only". In the former, the device gets all data modifications that have been committed on the server; in the latter, the server exports all of its data to the device, which then replaces its current set of data.

Assuming that the correct set of data has been replicated to the device, stage **2** simply involves the execution of the disconnected application against the local database.

Consider, for example, an "order entry" application that enables agents to record customer orders. Enabling this application to run on a disconnected device requires, during stage **0**, that a system administrator replicate the stock catalog consisting of items, in-stock quantities, agents, customers, and prior orders, if any. Before disconnection, the device's database is brought up-to-date (stage **1**), so that the recorded stock levels match the server's values. The agent is then able to take new orders (stage **2**) while disconnected from the server database. We assume that the agents prefer to work while out in the field, where connectivity may be unavailable or sporadic.

In order for work performed on the disconnected device to become visible to other applications, the device must transactionally propagate its updates to the server-side database that maintains the master version of the data seen by other applications and users (stage **3**). While propagating the change set from the client to the server, stage **3** must deal with the following issues:

- *Conflict Detection*, in which the application or middleware detects whether the change set conflicts with the current state of the server-side database.

  One important issue is how the notion of a "conflict" is defined. Connected business applications typically define conflicts as non-serializable transaction schedules [8]. Can this definition be used for disconnectable business applications as well? Efficiency is also a consideration: for example, does a detected conflict require that only that one transaction be aborted, or must the entire set of work performed on the device be aborted?

- *Conflict Resolution*, in which (if conflicts were detected), the application or middleware attempts to determine a new server-side state which eliminates the conflict. If no resolution is possible, the synchronization must be (partially or completely) aborted (i.e., the device's state cannot be automatically propagated to the server), and the failure logged and reported to the user. If some resolution is possible, it is performed, and the update is propagated to the server-side database.

  A key challenge here is whether general purpose conflict resolution algorithms can be devised or whether application-specific resolution is required.

- *Transactional Merge*, during which the change set, possibly modified by conflict resolution, is merged with the server-side database. As a result, work performed on the disconnected device is now visible to other applications and users – without violating the transactional guarrantees made by the application.

Note that update propagation between the client and server is asymmetric: updates performed on the server may invalidate transactions performed on the client, but client updates cannot invalidate previously-committed server-side transactions (because the server-side transactions were previously visible to all users and applications).

The programming model discussed here is orthogonal to stage **1**; the existence of middleware such as DB2 Everyplace [3] shows that efficient subscription and replication can propagate server updates to the client. The task of the middleware is to ensure that the application's execution behavior during stage **2** conforms as closely as possible to its behavior in a connected environment. The difficulty of this programming model projection is that we must deal with, and provide middleware for, the stage **3** synchronization process, during which the client's updates are propagated to the the server.

4 Projecting the EJB Programming Model

EJBs are a component model for enterprise applications written in Java. EJBs automatically supply common requirements of enterprise applications such as persistence, concurrency, transactional integrity, and security. Stateless session beans *(SSBs)* are EJBs that enable clients to request a service from the server, and are analogous to a procedure call implementation. Entity beans are EJBs that represent "objects", typically those backed by a persistent datastore. 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 EJB programming model is explicitly concerned with identifying and facilitating the distinct roles that are required to develop and deploy an application. The programming model specifies contracts that separate, for example, the bean-provider role (provider of the application's business logic) from the container-provider role (provider of the deployment tooling and runtime that supply deployed EJBs with functions such as transaction and security management).

We believe that the stage **3** synchronization function can be similarly abstracted as a container-provided function that

bean providers can ignore and that is transparently provided by the middleware. The EJB programming model is thus well suited for projection to disconnected devices: the existing role separation enables us to enhance existing containers without changing the API and semantics used in connected EJB applications.

In the context of projecting the EJB programming model to disconnected clients, we investigate two contrasting approaches to synchronization: *data replication* and *method replay*. We determined that our *EJBSync* middleware should implement the method replay approach. The reasons for this choice are given in Section 5.

4.1 Data Replication

The data replication synchronization technique represents a change set as a log of data modifications that were performed on the disconnected device. (The term "modifications" denotes data creation and deletion as well as data changes.) Data replication is used by both DB2e [3] and Lotus Notes [16]. It also underlies the notion of cached RowSets [22] in which the reference implementation uses optimistic concurrency-control. The synchronization process begins by transmitting the data modification log to the server.

- *Conflict Detection*: the server must track the data it has replicated to individual clients and determine whether activity by a given client – as represented by the data modification log – conflicts with changes that were previously committed by other clients or server-side applications. The standard algorithm is to detect a conflict if the synchronizing client has modified a datum that was concurrently modified on the server while the synchronizing client was disconnected. The server copy may have been modified by a server-based application or the synchronization of another client.

  Note that this algorithm, though commonly used in data replication, does not guarantee detection of all non-serializable conflicts. For example, if client 1 executes A = A + B, and client 2 executes B = B + A, this algorithm does not detect a problem, because the write-sets do not intersect. However, such cases do not seem to arise in practice.

- *Conflict Resolution*: the conflict resolution algorithm used in commercial systems places the burden on the user or system administrator. This is not due to laziness, but reflects the fact that (1) synchronization conflicts at the data level are difficult to resolve automatically, and (2) the cost of a mishandled conflict resolution being merged into the shared database may be very high. This implies that the application itself must specify how conflicts should be resolved.

  In some applications, the cost of a mishandled conflict is not as serious, and/or the probability of an incorrect automatic resolution is not that high. For example, Lotus Notes can be configured to resolve conflicts between document records automatically, either by merging all the modified columns together or by taking the last-modified version of the document. This is adequate for some applications. Also, the resolved documents are typically viewed by users rather than by programs, and the users will tend to see most merge problems. Finally, if an automatic resolution fails, a new "conflict document" is created, which again will typically be seen by users.

  For general databases and applications, though, this is not acceptable. Many applications databases are accessed directly by programs, and those programs will not know how to deal with conflict documents or "funny looking" data. For example, many databases are accessed via a JDBC [12] interface. There is no provision in JDBC for calling "user exit" code to resolve conflicts, or for providing a view of conflict records to applications.

- *Transactional Merge*: data deleted on the device must be deleted on the server; data created on the device must be created on the server; and updates performed on the device's data must also be performed to the server's data. For example, in the case of DB2 Everyplace, the server transactionally performs the appropriate sequence of SQL DELETE, INSERT and UPDATE operations. In the case of Lotus Notes, the client's version of the NSF (Notes Storage Facility File) records are copied over the master copy.

If data replication is used to implement the *order entry* example, as orders are placed on the disconnected device, the stock levels are correspondingly modified. During synchronization, those stock table rows that were modified (by reducing the stock level) are transmitted to the server. Figure 1 sketches what happens during a successful data replication synchronization.

4.2 Method Replay

The method replay synchronization technique represents a change set as a log of the method invocations performed on the disconnected device. Each log entry constains the information needed to replay a single method: e.g., the method name, the method's signature, and the method's parameter values. In can thus be seen as the "dual" of the data replication approach which logs the data modifications. A version of log replay is used in the "Field Calls" in IMS Fast Path [11], earlier work on long-running transactions

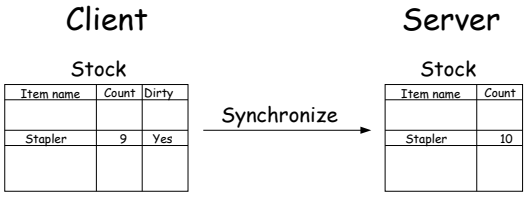

Figure 1. Data Replication Synchronization

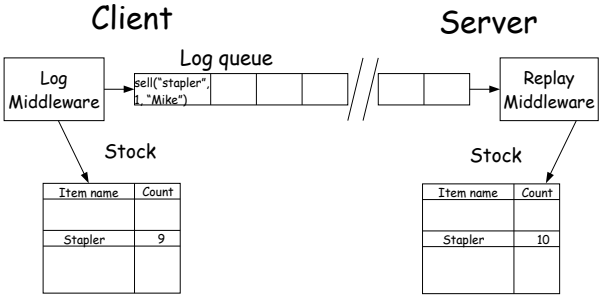

Figure 2. Method Replay Synchronization

[19], and IceCube [9] (see also [10]). The synchronization process begins by transmitting the method log to the server.

- *Conflict Detection*: the method invocations are replayed, in sequence, against the server's current state. This simultaneously propagates the devices's updates to the server (if the method replay is successful) and detects a conflict (if the method replay is unsuccessful). Whether a method replay is successful depends *solely* on the application's business logic: i.e., on whether the method throws an exception when invoked on the server.
- *Conflict Resolution*: the exception that caused the method to fail, when replayed on the server, is logged and the user is informed of the error. The conflict must be resolved manually.
- *Transactional Merge*: the disconnected work is transparently applied to the server through successful method replay.

Figure 2 sketches what happens during a successful method replay synchronization for the disconnectable *order entry* application.

4.3 EJBSync (method replay) Implementation

Because an EJB's state is backed by a datastore (typically a relational database), data replication can be used to synchronize an EJB application that executed on a disconnected device to the server. As the EJBs are modified, they modify the backing datastore, and that datastore can be synchronized with existing middleware such as DB2e. However, as explained in Section 5, the programming model projection is not as successful when using data replication. We therefore built *EJBSync*: method-replay middleware, customized for EJB-based applications.

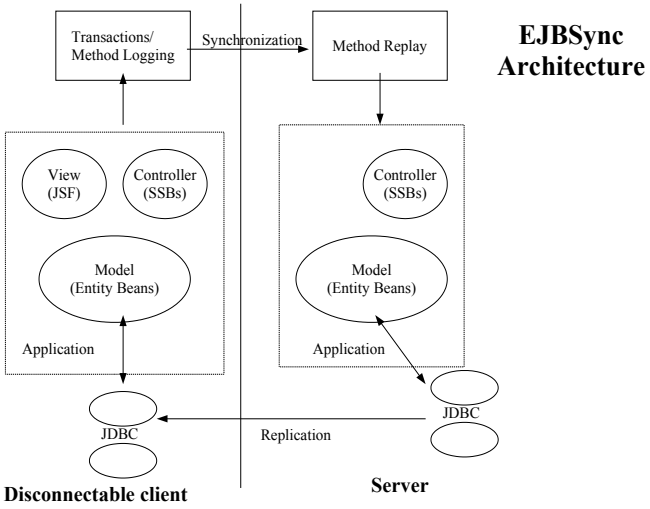

Figure 3. EJBSync: Method Replay Infrastructure

The EJBSync (see Figure 3) middleware is both application-independent and application-transparent. That is, an EJB business application developed for a connected environment can be deployed to a disconnected environment with no (or few) changes.

EJB methods are specified in an interface definition that is invoked by clients. EJBSync extends the tooling that deploys EJBs to a connected container in the following way. Whenever the deployed component (i.e., the class that implements *EJBObject*) delegates a client

invocation to the bean implementation (i.e., the class that implements *SessionBean* or *EntityBean*), a *top-level* method invocation is logged by creating a LogRecord EJB. For example, if an application invokes the SSB `placeOrder` method, and that method, in turn, invokes the `OrderHome.createOrder` and a series of `Order.addLineItem` methods, only `placeOrder` (and its arguments) are logged. The EJBSync middleware is responsible for tracking the dynamic method depth at which a given method executes.

Because it has access to the container's transaction mechanisms, EJBSync respects the transaction structure that is dynamically created as the application executes. Thus, LogRecords are scoped with respect to a given transaction; within a transaction, they are ordered in the sequence that the methods were invoked. This allows all methods invoked in a given transaction to be atomically replayed on the server.

The default Java Serializability mechanism enables the LogRecord fields to be transmitted by the client to the server (during synchronization) without much difficulty. One important detail deals with the fact that the default EJB serialization retains information about the address-space (JVM) in which the EJB resides, and creates a remote proxy when the EJB is deserialized in other JVMs. Our solution changes the serialization to remove the address-space information, so that EJB references are deserialized as references to local EJBs with the same primary key, residing in the local Home with the same JNDI name. This information, together with Java's reflection mechanisms, enables server-side reconstruction of the EJB on which the method was invoked as well as the method's parameters. A method is replayed by executing `invoke` on the corresponding `Method`.

This approach enables an application-independent synchronization protocol.

1. The synchronizing client invokes `Replicator.initiateSync` on the client-side remote Replicator stub. All LogRecords created since the last synchronization are transmitted to the server, in addition to the client's id and current synchronization-session id.

2. The server-side Replicator SSB iterates over the set of LogRecords, batching the LogRecords of a given transaction together, and ordering the transactions as they were originally invoked on the client. Each batch is invoked in a separate server-side transaction and replayed atomically. The failure (as indicated through an exception thrown by a replayed method) of one transaction, automatically causes the failure of all subsequent transactions. Finally, the server returns a *SyncToken* to the client which can be used to query the status of the synchronization.

Although EJBSync is customized for EJBs, we believe that this approach can be applied to other transactional component models [14] such as CORBA [1] and DCOM [2].

5 Substantiation

We claim that a connected programming model for business applications can be projected to disconnected devices such that:

- the semantics of the programming model are unchanged regardless of whether the application executes in a connected or disconnected environment;
- the need to synchronize the device's work to the server is hidden from developers by a combination of the programming model and middleware;
- applications developed with this approach can be usefully deployed to disconnected devices because work performed on the device will be committed successfully to the server.

A quantitative evaluation of this claim requires considerable experience with deployed applications over a long period of time. In the absence of such experience, we propose to validate our claim by comparing our approach with some popular alternatives.

5.1 Exotic Transaction Models

Several more complicated programming models have been introduced to address the problems of mobile disconnected business applications ([4], [17], [21]). These attempt to reduce conflicts and/or operate in resource-constrained environments. As discussed in Section 2, business applications (at least in the medium term) will be deployed to sufficiently robust environments that the use of non-standard transaction models is not needed to reduce synchronization conflicts. Also, it seems likely that many of the resource contraints of disconnected clients will be reduced or eliminated in the future. Thus is seems preferable to use a programming model which is familiar to developers of connected applications.

5.2 Message-Based Programming Model

A *message-based* programming model is a common approach to building disconnectable applications. Business applications are explicitly partioned into two portions: one is explictly coded to execute on the disconnected device, and the other is explicitly coded to execute on the server when the device reconnects to the server. The programming

model is "message-based" because, on a per-application basis, developers devise a suite of messages that are transmitted by the device to the server during the stage **3** synchronization process. Upon receiving these messages, the server invokes programs that propagate the change-set to the server's database.

In our *order entry* example, the portion of the 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. During synchronization, this state is transmitted to the server; a server-side program then executes the order on the server using the state that was previously saved on the disconnected device.

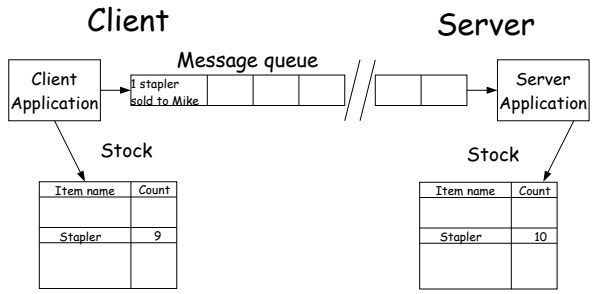

Figure 4. Message-Based Programming Model

Figure 4 sketches message-based synchronization to implement the *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.

In terms of our evaluation criteria, the message-based programming model is less useful than *EJBSync* because it requires two distinct application implementations (for connected and disconnected applications), and it forces developers to be explicitly aware of the synchronization process. On the other hand, because developers can completely customize the message suite and message contents, developers can potentially "tune" the application so as to commit the maximum amount of the device's work to the server. Similarly, message-based synchronization can potentially minimize bandwidth because only the minimum number of methods and the minimum amount of state needed to invoke the server-side program has to be recorded on the client and transmitted to the server.

We consider the disadvantages of the message-based programming (compared to EJBSync) to be considerable. Developers must "hand-craft" a two-part solution (client and server) on a per-application basis. 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. EJBSync is a step forward in the way it pushes more function into generic middleware. EJBSync also provides a productivity improvement, in that developers can focus their efforts on the application-specific logic, rather than the infrastructure. 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 message-based 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 message-based approach also requires extra programming to enable 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, the actions 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.

Interestingly, EJBSync can be seen as a middleware-based version of the message-based programming model. As with the message-based approach, EJBSync tracks the key business activities that have occurred during the disconnected application's execution. Unlike the message-based 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 message-based programming model may well (at least initially) provide a more optimal solution than EJBSync. The usual tradeoff applies, however: development and maintenance costs are considerably cheaper with an

automated approach, and automated solutions are typically improved over time.

5.3 Data Replication *versus* Method Replay

As mentioned in Section 4, the techniques used by EJB-Sync can be used to "hide" data-replication synchronization in the same way that method-replay is hidden. Based only on our first two evaluation criteria, these approaches are equally good since middleware is used to capture the device's change set and propagate it to the server. Both approaches facilitate the deployment of disconnected applications since the development, test, and maintenance costs have already been incurred when building the connected version of the application. In fact, DB2 Everyplace provides precisely this synchronization techique together with integrated middleware in a commercial product. The key differentiator between these approaches relates to our third criterion: the way in which the programming model supports *successful* propagation of work from the client to the server. Specifically, method replay has the following advantages compared to data replication:

- Method replay creates a smaller "footprint", in that it is less likely to cause conflicts during the synchronization process.
- Method replay projects a more consistent connected programming model to the disconnected device because of the way conflict resolution logic is specified.
- With method replay, the device's work executes against current data (during synchronization) rather than an out-of-date version of the data.

We discuss these advantages in more detail below, and then discuss their implications for building disconnectable applications.

5.3.1 Smaller Conflict Footprint

The more that the device's database state differs from the server's database, the more likely that synchronization will fail. While we definitely do not want to ignore a "true" conflict, we also want to minimize detection of "false" conflicts since those will cause useful work performed on the device to be discarded, or will require manual intervention to fix. Because data replication performs synchronization in terms of low-level data (where it is difficult to introduce higher-level semantics or application logic), it is more likely to create such false conflicts than method replay.

To see why this is so, consider the *order entry* example, and note that two clients may concurrently decrement the stock level for the same item. Data replication synchronization sees these actions as conflicting: different clients have modified the same row of the database. The middleware *must* flag this as a conflict, because this will lead to a transaction serializability violation (one of the clients' updates will be a *lost write* [8]). From the perspective of the application, however, the desired semantics are merely that sufficient stock exists during synchronization to satisfy the order. The precise stock level value for a given item is irrelvant. These semantics are easily expressed by a connected version of the application. They are also easily expressed for a disconnected version that uses method replay. In fact, only one version of the application need exist, since the identical `purchaseOrder` method executes for both connected and disconnected environments, decrementing the stock levels appropriately, and validating only that `(0 != stockLevel)`. The same method – with the same validating business logic – that executed and was logged on the device, is replayed on the server during synchronization. Disconnected versions of the application that use data replication synchronization must use different semantics: stock levels during synchronization must be identical to their levels during disconnected execution. Because application-independent data replication middleware does not have enough semantic information about the application to resolve this automatically, synchronization will fail.

5.3.2 Consistent PM Projection

Advocates of data replication may counter that the previous argument is naive. The sort of false conflicts described above can be easily avoided by adding business logic as necessary to the synchronization engine. The middleware makes application-specific synchronization "hooks" available to developers in order to explicitly compensate for (and resolve) such false conflicts. The synchronization hooks could (reasonably) assume that applications only increment and decrement stock levels, and thus could determine the stock level that would result from the synchronization of multiple transactions.

However, this starts the synchronization middleware down the path of understanding the application semantics. The separation of synchronization business logic from application business logic is awkward, to say the least. The synchronization business logic is placed in the position of trying to reverse-engineer the changes made by the application logic, so that it can reconcile the data. More importantly, however, the more that such special-purpose logic must be added by *developers* – rather than being performed automatically by middleware – the more the programming model's consistent projection is compromised. While making the application disconnectable, it forces developers to explicitly "code around" the issue of disconnected operation. More pragmatically, rather than maintaining a single version of the application that can be deployed to both con-

nected and disconnected environments, the business must now maintain two (or one and a half) versions of the application.

5.3.3 Execution Against Current Data

Because business applications must behave transactionally, a transaction model is part of any programming model for business applications. At a high-level, both data replication and method replay offer the same transaction semantics. In terms of the taxonomy presented in [6], both use a detection-based algorithm, with deferred validity checking, and invalidation when notified by the server about an update. (This approach is sometimes termed "optimistic", in contrast to pessimistic, lock-based, concurrency control mechanisms.) This enables high server-side availability – despite long periods of disconnection or even device failures – because the server is not forced to avoid potential conflicts.

Similarly, both approaches enable the connected transaction model to be projected to disconnected devices. EJBSync, for example, presents a disconnected device with the J2EE/EJB transaction model: sections of code can be explicitly demarcated with a `UserTransaction`, or the developer can use declarative transactions on a per-method basis. Transactions that are rolled back on the disconnected device are not replayed on the server (since the LogRecords are not committed). During synchronization, the transaction boundaries created during the client's execution are preserved on the server, as is the original transaction ordering. Data replication, as well, can project the connected transaction model to a disconnected environment. Although DB2 Everyplace sync does not maintain an application's transaction boundaries or ordering, this is a limitation of the implementation – not that of the idealized algorithm. Gold Rush, for example [7], shows that mobile transaction middleware can do data replication synchronization and preserve transaction boundaries and ordering. A similar situation holds with respect to validating "read sets" – the set of data read, but not written, by the application. Data replication implementations validate that conflicts have not occurred with respect to the application's "write set", but do not appear to check whether other users have concurrently modified the synchronizing device's read set. Although this can theoretically lead to a violation of transaction isolation [8], in practice, we find it difficult to identify a realistic scenario in which this leads to a serializability violation that was not caused by the application logic. Regardless, enhancing data replication implementations to also validate an application's read sets is relatively straightforward, and is not a fundamental limitiation of the algorithm.

The key difference between the transactional models of data replication and method replay is more subtle, and involves the definition of transactional isolation. Data replication synchronization ideally attempts to ensure that the device's transactions are *conflict serializable* [8] with the transactions that were previously committed on the server. This criterion is defined in terms of read and write operations on data. Method replay attempts to ensure that *transformations* performed on the device are compatabile with transformations previously committed on the server. In effect, transactions executed on the disconnected device are *moved* to a later point in time: namely, the time at which synchronization occurs. They are guaranteed to be serializable with the server-side transactions because, from the shared database viewpoint, they are executed *after* all the connected transactions have executed, and using any data modified by the server-side transactions as their inputs. Thus, strictly speaking, there cannot be a serializability violation. Under model replay semantics, the fact that work executed on a disconnected client is (almost) irrelvant to the transaction model. From a developer's viewpoint, therefore, the programming model is projected more consistently; from the client's viewpoint, the work executed against the most current version of the data.

The only caveat to the method replay transaction model is that any "human input" into the disconnected client transactions may not be replayed accurately, because the human thought processes are not captured in the method implementations. For example, the user may have looked at her checking balance on the disconnected client, seen a balance of $1000, and decided to withdraw $100. When the disconnected transactions are replayed on the server, the balance may have been $101. The replayed method will withdraw $100, leaving $1, but had the user known she only had $101 she may have decided to withdraw $50, or nothing. Also, she thinks her new balance is $900, which does not represent the balance at any time in the shared database. Typically, however, business logic will prevent a withdrawal of an amount greater than the available funds at synchronization time, unless overdrafts are permitted. Thus the synchronization process will not result in database consistency violations, unless the application is flawed. Note also that these "human serializability violation" scenarios occur with data replication as well. Data replication algorithms typically do not verify that data which was read but not modified by the disconnected client is unchanged on the server at synchronization time (in fact we are not aware of *any* data replication system which does this verification).

5.4 Useful Disconnectable Business Applications

We conclude by arguing that only approaches similar to EJBSync enable development of useful disconnectable applications, because of the inevitable issues that arise as server-side function is moved to the client. A continuum ex-

ists with respect to the degree to which server-side function is moved to the client. Browser-based applications exist at one extreme, in which almost all of the application resides on the server. This "thin-client" approach has certain advantages, but the application cannot execute on a disconnected device. To enable disconnectable applications, more of the data and more of the application logic must reside on the client. At the other extreme of the continuum, *all* of the application executes on the device. This may not always be practical, but can be done for certain types of applications (e.g., for the *order entry* example used in this paper). However, as we have shown, this approach causes the maximum amount of data changes to take place on the device (e.g., stock level changes). As more changes are made to the device's database, it becomes more likely that false conflicts will be detected by data replication middleware (false in the sense that human intervention could easily resolve them). Relatively speaking, method replay will cause fewer false conflicts to occur, because the replays happen against up-to-date server data.

Consider the middle of this continuum, in which some application logic is moved to the disconnected device, but some remains on the server. For example, take the usage of a customer's "available credit" balance in validating an order. The credit balance could be replicated to the client, and orders placed only if the customer has sufficient credit. If the balance is sufficient, the credit balance would be decremented by the value of the order. Obviously this requires that *more* data be replicated to the client. For data replication, this also raises a problem similar to the stock-level problem discussed above. Because a customer's credit balance is modified by each placed order, orders placed for the same customer by different clients will always result in a (usually false) conflict during the synchronization of the second client. This conflict is usually false because only the exhaustion of the customer's credit balance is actually a problem. Two debits to the credit balance that do not exhaust the customer's credit could be combined arithmetically during synchronization.

In order to eliminate this false conflict for data replication, the credit-balance check could be eliminated from the disconnected client version of the application. Since the *business* still requires that the balance be checked and updated before fulfillment of the order, a separate server-side process, triggered by synchronization, must be put in place to do the credit-balance check and update. The code itself is not the problem. The problem is that this code is separate from the disconnected and connected versions of the application. It is not embedded in the original order-entry application flow, and it is not necessarily easy to fit it into the post-synchronization workflow. In this scenario, in fact, data replication synchronization begins to resemble the message-based synchronization (with all of its disadvantages) discussed above.

In contrast, with method replay, the `purchaseOrder` method is modified to conditionally perform the credit check and modify the existing balance only when connected (and thus during synchronization as well). This additional application logic is ignored when executing on the disconnected device. All that this approach requires is the ability to determine whether a method is executing in a connected or disconnected environment. The connected version performs the required credit-check; the disconnected version branches around that code (placing the order without the credit-check); and, during method replay, the credit-check will be transparently performed before placing the order.

It seems therefore that the following tension exists with respect to deploying business applications to disconnected devices. If you wish to use a single programming model so as to develop and maintain a single version of the application, the application will modify much transactional state. During synchronization, data replication then is more likely than method replay to abort the work performed on the device because of false conflicts. This problem can be reduced, but only by creating connected and disconnected versions of the application. Here too, method replay is more useful than data replication because the branch logic is more easily packaged within a single method or business process than as extraneous processes that must be hooked into the business workflow during synchronization. Executing the bulk of the application on the server, with only a minimal application executing on the client, mitigates this problem but only by drastically limiting the usefulness of disconnecting the application in the first place.

The use of method-replay synchronization does constrain the application to take extra care when accessing data that is outside the shared datastore. If such data becomes part of the client-side datastore, it may not be used correctly during replay because the external value may have changed since the original disconnected execution. For example, if an application sets fields based on the current time (e.g., timestamps) or using unique identifiers [26], the replayed application will use the *current* values of the external data (e.g. the current time), not the values that were originally used on the device. This may or may not be a problem. If the UUID is used to set an EJB's primary key, and that EJB is referenced (and logged) by the application, method replay synchronization will fail because that EJB's identity on the server will be based on a new UUID value.

Data replication experiences a similar issue when dealing with unique identifiers. Because the locally unique identifier (LUID) client ID may be different from the globally unique identifier (GUID) server ID, the server must maintain an ID mapping table for all items exchanged between itself and the client. Otherwise, the client's datastore that references a set of LUIDs cannot be translated (or identified

as referring to the data on the server by a different name) by the server. SyncML [25], for example, uses a *Map* operation to send the LUID of newly created data to the server. This allows the server to update its mapping table with the new LUID, GUID association. However, this is not sufficient to address all possible problems. If the LUID gets incorporated directly into application data, the synchronization process may not know how to modify the application data, or even detect that it should be modified during synchronization.

6 Summary and Conclusion

We showed that a programming model for business applications can be usefully projected to disconnected devices – despite the fundamental algorithmic and infrastructure issues that must be addressed. This approach enables developers to apply skills acquired in connected applications to disconnected environments. It also enables businesses to maintain only a single version of an application. We described the programming model and the *EJBSync* prototype middleware, and showed how this approach compares favorably to other approaches for developing disconnected applications.

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] Dunham, M. H.; Helal, A.; Balakrishnan, S. Mobile Transaction Model That Captures Both the Data and Movement Behavior. Mobile Networks and Applications Vol 2, No 2, 1997, 149-162.

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

[6] 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.

[7] M. A. Butrico et al. Mobile Transaction Middleware with Java-Object Replication. Proc. Third USENIX Conference (COOTS), 1997.

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

[9] A.-M. Kermarrec et al., The IceCube approach to the reconciliation of diverging replicas. Proc 20th annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC) Aug. 2001.

[10] IceCube http://research.microsoft.com/camdis/icecube.htm. 2005.

[11] IBM IMS Family. http://www-306.ibm.com/software/data/ims

[12] Java Database Connectivity (JDBC) http://java.sun.com/products/jdbc/. 2005.

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

[14] 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. 99-152. 2001.

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

[16] B. Benz, R. Oliver. Lotus Notes and Domino 6 Programming Bible. Wiley, 2003.

[17] Madria, S. K.; Bhargava, B.; A Transaction Model to Improve Data Availability in Mobile Computing. Journal of Distributed and Parallel Databases, Vol. 10 No. 2, 127-160, 2001.

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

[19] B. Bennett et al. A Distributed Object Oriented Framework to Offer Transactional Support for Long Running Business Processes. ACM Middleware 2000. 331-348.

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

[21] E. Pitoura, B. Bhargava. Maintaining consistency of data in mobile distributed environments. 15th International Conference on Distributed Computing Systems (ICDCS'95).

[22] JDBC Rowset Tutorial http://java.sun.com/developer/Books/JDBCTutorial/chapter5.html. 2005.

[23] Serrano-Alvarado, P.; Roncancio, C.; Adiba, M; A Survey of Mobile Transactions. Journal of Distributed and Parallel Databases, Vol. 16 , Issue 2, 193-230, 2004.

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

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

[26] Universal Unique Identifier,
http://www.opengroup.org/onlinepubs/9629399/apdxa.htm