RC23881 (W0602-133) February 15, 2006
Computer Science

IBM Research Report

Efficient Replication for Disconnected Business Applications

Avraham Leff, James T. 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

Efficient Replication for Disconnected Business Applications

Avraham Leff James T. Rayfield

Abstract

Business applications that execute on disconnectable client devices require the periodic replication of server-based data to keep them up-to-date. We propose an efficient replication algorithm that compresses the state that a server must store to perform such replication. Furthermore, the algorithm enables additional compression when a server transmits state to an individual client. We examine the implications of this approach for client-to-server synchronization, and discuss issues related to implementing this algorithm in a prototype.

Index Terms

database replication, database compression, synchronization, method-replay

I. Introduction

A. Disconnected 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 [5] to a shared database; and (3) must operate securely. Business applications therefore have requirements that other applications do not: chiefly, to access persistent shared datastores securely and transactionally. Business applications have traditionally been deployed in *connected* (continuously networked) 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

IBM T.J. Watson Research Center email: avraham@ibm.com
IBM T.J. Watson Research Center email: jtray@ibm.com

database residing 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 [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 [6] 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.

The set of "non-business" disconnected applications is declining in size and importance at the same time that the set of disconnected business applications is expanding. For example, even simple mobile applications typically support synchronization of updates back to the user's personal PC. Since the PC copy of the database may be updated by both the synchronization agent and other PC-based applications (e.g. calendaring), the database is, in fact, shared. Also, users will probably be very disappointed to discover that synchronization of updates did not occur transactionally (e.g., if concurrent updates to the same record were not detected and resolved in some way). Finally, security of PDA databases is certainly a concern nowadays.

Various scenarios motivate the requirement for an application to execute on a disconnected device. A network connection to the server may simply not exist; or the network bandwidth may be so limited, or the latency so great, as to make constant communication between the device and the server impractical. To operate while disconnected, the device must therefore cache sufficient state from the server in order to execute the disconnected applications. This caching requirement, in which the server maintains the "master" database state for a set of clients, and clients independently cache the state they need to execute disconnected aplications, implies that disconnected

applications have a *life-cycle* that distinguishes them from their connected counterparts. Ideally, disconnected applications should be supported by middleware that addresses this life-cycle's requirements. Otherwise, developers are forced to solve these problems on a per-application basis.

B. Life-cycle of a Disconnected Business Application

Disconnected business applications typically have the following life-cycle. First, their initial deployment on a client device requires that an administrator perform a one-time setup of the mobile device's database(s). The key challenge here 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 may 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 sequence:

Server-to-Client Replication: Before disconnecting, the server's updates are propagated to the device. As with the one-time setup tasks, typically only the relevant subset of updates are copied, e.g., by executing a query (or set of queries) against the server database, and storing the results on the device's database.

Application(s) Execution: The user executes one or more business applications on the disconnected device.

Client-to-Server Synchronization: The user reconnects the device to the server, and the client software propagates the work performed while disconnected to the server-side database.

Note that in our terminology, we use "replication" to indicate server-to-client data flow, and

"synchronization" to indicate client-to-server data flow.

C. Replication

The goal of the replication phase is for the device's state to be identical to a subset of the state on the server. To the extent that this goal is *not* met, the device's state becomes increasingly "stale" as it diverges from the server's actual (and "master") state. (We assume that multiple devices concurrently execute the same set of disconnected applications.) Also, when the device's state is "stale", modifications to this state are more likely to be invalidated when the device synchronizes its state to the server.

Current approaches [1][2] recognize that transmitting a complete copy of the server database to each device on every replication request is very inefficient, since the device's database typically has not diverged that much (since the last replication) from the server's state. It is more efficient for the server to transmit only the state needed to transform the device's state to the server's current state. Therefore, the server stores – on behalf of all its clients – sufficient state to transform each client's state to that of the server during replication. Standardized protocols can then be used to pass data between a device and the server. SyncML [14], for example, enables such data exchanges – across wireless and wired networks and over multiple transport protocols – by using the standard format defined in its "representation" protocol. In addition, SyncML provides a "synchronization" protocol which 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.

Our paper improves the current approach through a compression scheme through which (1) the server reduces the amount of state it needs to perform replication and (2) the server reduces the amount of state that it transmits to a given client during the replication process itself. Furthermore,

Fig. 1. Uncompressed Server-Side Replication State (DB2 Change-Data)

IBMSNAP_COMMITSEQ	IBMSNAP_INTENTSEQ	IBMSNAP_OPERATION	NAME	VALUE
x'43EBBE8D000000010000'	x'00000000000062744C57'	U	balance	$3000
x'43EBBE8D000000010000'	x'00000000000062744CC7'	U	balance	$4000

our paper examines the implications of this compression scheme on synchronization algorithms.

II. Compression-Based Replication

The following example illustrates these different approaches to representing replication state. Assume that the datum D has an initial value of D_0 at time t_0. This value is replicated to $client_1$ during its initial client setup at time t_1. After $client_1$ has disconnected, a single transaction executes against the server database at time t_2. This transaction sets D to the successive values $\{D_1, D_2, D_3\}$ and then commits. Then, at time t_3, a second client ($client_2$) performs its initial setup, receiving the value D_3. Finally, at time t_4, a second transaction executes, setting D to the successive values $\{D_4, D_5\}$.

Current (non-compressed) approaches [1][2] log the *entire* sequence of changes (D_1 through D_5) on the server. We verified this experimentally with DB2 v8.2.3 by applying this SQL sequence to a sample table:

```
update mvcchip.crudpp set value='$3000' where name='balance'
update mvcchip.crudpp set value='$4000' where name='balance'
commit
```

The resulting change-data table records are shown in Figure 1.

We see that even though the two updates were applied in a single transaction (`x'43EBBE8D000000010000`
the first update record is kept even *after* the second update record is inserted into the log. This is

done despite the fact that, once the transaction commits, only the last update is relevant.

In contrast, in our approach the server logs only D_3 and D_5 as a consequence of using the compression algorithm discussed below. We trade-off the benefit of reduced storage against the processing cost of applying the compression algorithm.

As shown above, compression can reduce the amount of state that the server must store. Compression can also reduce the amount of state that a server transmits when replicating to its clients. Thus, using an uncompressed approach, if both clients perform an incremental replication at time t_5, the server will transmit $\{D_1, D_2, D_3, D_4, D_5\}$ to $client_1$ and $\{D_4, D_5\}$ to $client_2$. The clients, in turn, apply this state change sequence serially to transform their copy of D to its current value on the server (D_5). In contrast, in our approach, the server transmits only D_5 to both $client_1$ and $client_2$. This provides two benefits: less bandwidth is consumed in server-to-client transmission, and client-side processing is reduced.

To understand how the compression algorithm works, we must first explain "what" is being compressed: namely, a *state change record*.

A. State Change Records

A state-change record contains information about a single state change to a single datum. It consists of:

- The *type* of the state change: either a CREATE, a DELETE, or UPDATE. In contrast to the well-known "CRUD" operations, a state-change record only tracks "CUD" operations. Retrieve (query) operations on the data are irrelevant to replication because they do not change a datum's state.
- A sequence number (or timestamp) that specifies where this state change occurred relative to the server's overall state change (or transaction) sequence.
- An identifier that uniquely identifies the datum to which the state change was applied. In

TABLE I
COMPRESSING TWO-RECORD SEQUENCES

$C_{S_1}C_{S_2}$	$\Rightarrow$	Illegal create of existing record
$C_{S_1}U_{S_2}$	$\Rightarrow$	C_{S_2}
$C_{S_1}D$	$\Rightarrow$	Empty set (remove the C record)
$U_{S_1}C_{S_2}$	$\Rightarrow$	Illegal create of existing record
$U_{S_1}U_{S_2}$	$\Rightarrow$	U_{S_2}
$U_{S_1}D_{S_2}$	$\Rightarrow$	D
DC_{S_2}	$\Rightarrow$	U_{S_2}
DU	$\Rightarrow$	Illegal update of nonexistent record
DD	$\Rightarrow$	Illegal delete of nonexistent record

the case of a relational database row, for example, one would typically use (1) the name of the database table containing the row and (2) the row's primary key.

- The state (value) of the datum (e.g., the database row) *after* the given state change was applied. Thus C_{S_1} refers to a create operation with value S_1, U_{S_2} refers to an update with value S_2, and D refers to a delete (deletes have no state associated with them).

B. Server-Side Compression

Replication-record compression is the process in which a sequence of state-change records, during one or more transactions, is replaced with (at most) *one* state-change record containing only the state needed for replication to a client. Because clients are interested only in the end result of the sequence, the server need not store the sequence itself. Table I shows how compression is applied to valid two-record sequences. A sequence $C_{S_1}U_{S_2}$ indicates that a datum was first created with value S_1, and subsequently updated to have value S_2. Sequences such as $C_{S_1}C_{S_2}$ are logically impossible (a record cannot be recreated after being created without first being deleted). We do not show the sequence number and identifier components of the replication records as they are irrelevant to the compression algorithm.

While the basic compression idea is straight-forward, care must be taken in the details. In the case of DC, for example, we cannot simply compress to a C because (from a client's perspective)

the datum could *not* have been created. Since the datum already exists on the client, attempts to "create" the datum on the client, during replication, in order to bring its state up-to-date with the server will fail. Instead, the sequence is compressed to a U operation, coupled with the datum's state after it was created. This allows the client to replicate from the server by simply updating the already existent datum to its most recent state on the server.

Since Table I shows how to convert all valid pairs of state-change records into a single state-change record, it is also clear that sequences of three or more state-change records may be compressed by repeated application of the compression rules. First, compress the first two records into a single record; then, compress the result of that compression with the (former) third record, and so on until only a single record remains. Table II shows the results for sequences of three state-change records.

TABLE II
COMPRESSING THREE-RECORD SEQUENCES

$C_{S_1}U_{S_2}U_{S_3}$	$\Rightarrow$	$[C_{S_1}U_{S_2}]U_{S_3}$	$\Rightarrow$	$[C_{S_2}]U_{S_3}$	$\Rightarrow$	C_{S_3}
$C_{S_1}U_{S_2}D$	$\Rightarrow$	$[C_{S_1}U_{S_2}]D$	$\Rightarrow$	$[C_{S_2}]D$	$\Rightarrow$	Empty set
$C_{S_1}DC_{S_3}$	$\Rightarrow$	$[C_{S_1}D]C_{S_3}$	$\Rightarrow$	$[]C_{S_3}$	$\Rightarrow$	C_{S_3}
$U_{S_1}U_{S_2}U_{S_3}$	$\Rightarrow$	$[U_{S_1}U_{S_2}]U_{S_3}$	$\Rightarrow$	$[U_{S_2}]U_{S_3}$	$\Rightarrow$	U_{S_3}
$U_{S_1}U_{S_2}D$	$\Rightarrow$	$[U_{S_1}U_{S_2}]D$	$\Rightarrow$	$[U_{S_2}]D$	$\Rightarrow$	D
$U_{S_1}DC_{S_3}$	$\Rightarrow$	$[U_{S_1}D]C_{S_3}$	$\Rightarrow$	$[D]C_{S_3}$	$\Rightarrow$	U_{S_3}
$DC_{S_2}U_{S_3}$	$\Rightarrow$	$[DC_{S_2}]U_{S_3}$	$\Rightarrow$	$[U_{S_2}]U_{S_3}$	$\Rightarrow$	U_{S_3}
$DC_{S_2}D$	$\Rightarrow$	$[DC_{S_2}]D$	$\Rightarrow$	$[U_{S_2}]D$	$\Rightarrow$	D

Use of compression-based replication gives two benefits:

- The server stores at most one state-change record per datum per transaction. Compression is performed dynamically by the server as transactions create, update, and delete data. As a result, the server stores the minimum number of records required to replicate its data on behalf of *all* of its disconnected clients. In this use of compression, the server cannot further optimize by compressing state-change records *across* transactions. Because the server must service multiple clients, each of whom may have (1) disconnected at different times, and

(2) may reconnect at different times, the server can compress only within a transaction's boundaries.

- The server transmits at most one state-change record per datum when replicating to a specific client. This suffices to replicate state that was modified during *all* transactions that executed while the client was disconnected from the server. In this use of compression, the server first determines that, while disconnected, a given client "missed" transactions tx_i, ..., tx_n, so that the server must transmit the state-change records from these transactions. The same compression algorithm originally used for intra-transaction compression is used to perform further inter-transaction compression over the set of records maintained by the server for transaction tx_i, ... tx_n.

III. Implications for Synchronization

Disconnected business applications can use compression in yet another way, depending on the method used to synchronize the work done by disconnected clients to the server.

Broadly speaking, synchronization may be *data-based* or *method-based* [10]. Data-based synchronization propagates the disconnected client's work by copying the client's *data* to the server. For example, assume that an insurance agent uses her "customer application" to issue a new policy. The application creates new rows in the "customer" and "policy" tables, and updates rows in the "agent" table. Data-based synchronization will copy the new and updated rows to the corresponding tables in the server's database: by doing so, the work performed while disconnected is propagated to the server. Method-based synchronization propagates the disconnected client's work by replaying (on the server) the methods that executed on the client. In our example, therefore, the "new policy" method of the "customer application" is replayed against the server's database. In one sense, the end-result is the same: new rows are inserted in the database tables, and existing rows are appropriately modified. However, the "propagation units" are methods rather than

database rows.

The relative advantages of data-based *versus* method-based synchronization are complex, and discussed elsewhere [10]. Assuming, however, that a client does successfully synchronize with the server, the data-based approach has an advantage over the method-based approach. Successful data-based synchronization automatically makes the client's state consistent with the server. The method-based approach must do more work even after a successful synchronization to reach this consistent state.

To see why this is so, consider the following subsets of the client and server databases.

- D_C: the set of data that was modified on the client (including creates and deletes) and *not* concurrently modified on the server.
- D_S: the set of data that was modified on the server and *not* concurrently modified on the server.
- D_B: the set of data that was modified on both the client and the server (while this client was disconnected).

Data-based synchronization is concerned only with resolving conflicts in D_B, such that the client and server databases are identical. Assuming that synchronization has made the value of D_B and D_C identical on the client and the server, D_B and D_C can be ignored during replication since we know *a priori* that the server's state is identical to the client's. Similarly, the processing of D_S during replication is straightforward: copy it to the client and overwrite the corresponding client data. Thus, synchronization followed by replication immediately results in consistent client and server states.

This attractive feature is not present when method-based synchronization is used. This is because the method-replay approach does not lend itself to simple definitions of D_C, D_S, and D_B. The fact that a datum was not modified on the client does not mean that it will not be modified

on the server during a method replay (although it strongly suggests it). Conversely, server-side method replays may leave data unmodified on the server that was modified during the original execution on the client.

This disadvantage of method-based synchronization can be viewed as the mirror-image of its advantage with respect to reducing synchronization conflicts. Consider a "new order" application that decrements stock from an "items" database table as an agent places the order. Assume that the stock count was five items when the server last replicated to the client and that one order was placed by the disconnected client. If no other orders were concurrently placed on the server, data-based synchronization can then simply copy its data (stock = *four*) to the server, and its database will match the server. But what if another order *was* concurrently placed on the server? In this case, data-based synchronization will typically fail because the middleware will detect a conflict between the client's and server's versions of the data. Method-based synchronization, in contrast, will succeed because the server's table will be serially modified to stock = *four* items (by the server-side application) and then to stock = *three* items (as the client synchronizes to the server). Conflicts are reduced precisely because the specific state of the data is unimportant: only successful application execution is important. However, the tradeoff is that successful synchronization may still require further work to ensure a consistent client database (e.g., the stock = *three* data must now be replicated to the client).

One approach is to perform a consistency check on each datum in D_B. However, we think that this effort is prohibitive. Instead, we prefer an approach in which – after synchronization – a client's database is:

1. Rolled-back ("undone") to its state immediately after the last server-to-client replication.
2. Rolled-forward ("redone") to the server's current state by replaying all state-change records created on the server since the client last replicated, including the changes made by the

replay of this client's methods.

We have already discussed how compression can be used to improve the performance of the "redo" step. We now discuss how systems that use method-based synchronization can also use compression to improve the performance of the "undo" step.

A. Client-Side Compression

The client-side compression algorithm is similar to the server-side algorithm. The key difference is that compressed server-side state-change records are used to replicate work performed on the server to the client. In contrast, compressed client-side state-change records are used to undo work all work performed on the client since the last replication. The client-side state-change record is almost identical to that described in Section A – except that it stores the state of the datum *prior* to the state change operation, rather than the state after the completion of the operation. Thus, U_{S_1} denotes an Update where the state prior to the update was S_1, and D_{S_2} denotes a Delete where the state prior to the delete was S_2. C denotes a Create operation; the state prior to the Create operation is undefined (the record did not exist). This allows the client to undo the effect of create, update, and delete operations.

As state-change operations are performed, they are logged on the client. If only a single operation was performed on a given datum, only a single corresponding client-side state-change record is created. When a subsequent state-change is performed on a given datum, the client compresses the two operations into one record, using the algorithm shown in Table III. The result of the client-side compression algorithm is that at most one state-change record, per datum, is stored on the client across *all* transactions that executed since the server last replicated to this client. (This differs from the server-side result, because a server supports multiple clients, but a client interacts with only one server).

The client-side algorithm is simpler than the server version in that, for *any other* state-change

TABLE III
CLIENT-SIDE COMPRESSION OF TWO-RECORD SEQUENCES

CC	$\Rightarrow$	Illegal
CU_{S_1}	$\Rightarrow$	C
CD_{S_2}	$\Rightarrow$	Empty set
$U_{S_1}C$	$\Rightarrow$	Illegal
$U_{S_1}U_{S_2}$	$\Rightarrow$	U_{S_1}
$U_{S_1}D_{S_2}$	$\Rightarrow$	D_{S_1}
$D_{S_1}C$	$\Rightarrow$	U_{S_1}
$D_{S_1}U_{S_2}$	$\Rightarrow$	Illegal
$D_{S_1}D_{S_2}$	$\Rightarrow$	Illegal

sequence, the client simply keeps the existing record. For example, for the sequence $U_{S_1}U_{S_2}$, the U_{S_2} is simply discarded. Because the undo algorithm reverts a datum to its state *before* the first operation was performed, the algorithm only needs the first state-change record.

Again, care must be taken in the details. In the case of $D_{S_1}C$, for example, we cannot simply keep the D_{S_1} because the undo algorithm will later try to recreate the ostensibly deleted – but still existent – datum. If *no* state-change record is kept at all (because the operations cancel each other out), the undo algorithm will not be able to restore the datum's state to that obtaining immediately before the delete operation.

The following *undo* algorithm uses the state-change records to undo all work performed since the last replication. For every state-change record:

- If the record is a C, delete the corresponding datum.
- If the record is a D_{S_1}, (re)create the datum, setting its state to S_1.
- If the record is a U_{S_2}, retrieve the existing record and set its state to S_2.

IV. COMPRESSION IN COMPONENT-BASED ARCHITECTURES

The prior discussion assumed that replication and the corresponding compression of state-change records is done to with respect to relational database tables and their records. We observe now that that the same concepts and algorithms also apply to component-based architectures such

as EJBs [4], CORBA [13] and DCOM [12]. Component-based replication requires only that (1) the component model has a well-defined API for creating, deleting, and modifying components and (2) that users of the cannot change component state except through this API. Transactional component architectures typically meet these requirements [9], and efficient replication can therefore also be applied to such component architectures. The algorithm presented in Section II is modified only with respect to how the state-change record identifies the associated datum. In the case of a row in a relational database, one would typically specify (1) the name of the row's table and (2) the row's primary key. In the case of (entity-bean) EJBs, one would would typically specify (1) the JNDI [7] name of the Enterprise JavaBean's Home and (2) the EJB's primary key.

Our experience with the *EJBSync* middleware [10] confirms this claim. EJBSync is a prototype system that projects the EJB programming model to disconnected devices using compression-based replication. Because EJBSync uses method-based synchronization (Section III), EJBSync clients also do state-record compression (Section A) to improve the performance of the undo algorithm. While building EJBSync, however, we found that important system issues must be addressed in order for compression-based replication to be practical. We describe these issues – and the way that we addressed them – below.

A. Commit-Sequence Number

The inter-transaction compression algorithms discussed in this paper require the system to know the order in which transactions committed. Although the system knows the order in which Prepare and Commit calls are made to the Resource Manager(s) [8], the multi-threaded nature of the system means that the transactions do not necessarily commit in the same order. Only the Resource Manager(s) (databases) know the order in which a transaction's results are actually written to the log, and this information is typically not exposed through any public API.

We addressed this problem by creating a (very short) critical section inside the commit method.

Within the critical section, each transaction is assigned a "commit sequence number" (CSN) before the Resource Manager is committed. The critical section prevents the Resource Manager commits from occurring in a different order than the assignment of CSNs. This solution is not ideal, but better solutions require access to the internals of the Resource Manager(s).

B. Reducing Hot-Spots

The data-structure that maintains the state-change log is a potential "hot-spot" that can greatly degrade performance. All server-side transactions must append state-change records to this data-structure; at the same time, replication requires that the data-structure be read (so as to perform inter-transaction compression). The naive approach that explictly serializes all access to the data-structure (a database table, in the case of EJBSync), precludes the possibility of significant concurrent server-side activity.

We solved this problem by storing the *set* of state-change records created by a transaction, using the transaction's CSN as the set's primary key. We then take advantage of the fact that CSNs define a monotonically increasing range of integers, so that the system can maintain a transient copy of the largest CSN. Clients that read the data-structure do so by initiating a processing loop in which each step accesses a single transaction's set of state-change records (a SELECT by commit sequence number) rather than accessing the records associated with a range of transaction activity. A well-defined range of transaction activity is thus read during the loop, terminating at the most recently committed transaction. This approach (at least with well-designed databases such as DB2) allow concurrent transactions to append to the table, since their writes involve commit sequence numbers that are not involved in "read" activity.

V. Conclusion

Business applications that execute on disconnected client devices require servers to replicate state to clients so that the client databases remain up-to-date. We have shown in this paper that the standard approach, in which a server stores all state-change activity, can be improved. Specifically, we presented an efficient replication algorithm that compresses the state that a server must store to perform such replication. The same algorithm, when applied to inter-transaction activity enables further compression of the state transmitted by a server to an individual client. The paper shows that a similar algorithm can be applied to improve the performance of systems that use method-based synchronization of client state to the server. Finally, we discussed issues encountered while implementing this approach in a prototype system.

In our approach to synchronization in method-replay systems, we asserted that it was easier to rollback the client state to the prior replication (disconnection) point, rather than trying to determine which parts of the modified client state match the post-synchronization server state. Future work may explore alternate approaches to this problem. Also, for method-replay systems, the existing algorithms require that the client database be quiesced during the synchronization/rollback/replication process that takes place at reconnection. Future work will examine algorithms which do not have this restriction.

References

[1] IBM DB2 Information Integrator: SQL Replication Guide and Reference (SC27-1121-02). `ftp://ftp.software.ibm.com/ps/products/db2/info/vr82/pdf/en_US/db2e0e82.pdf`, 2006.

[2] IBM DB2 Everyplacesync Server Administration Guide (SC18-7186-03). `ftp://ftp.software.ibm.com/software/data/db2/everyplace/doc/enu/sag82.pdf`, 2006.

[3] IBM DB2 Everyplace. http://www-306.ibm.com/software/data/db2/everyplace/index.html, 2006.

[4] J2EE Enterprise Javabeans Technology. http://java.sun.com/products/ejb/, 2006.

[5] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Francisco, CA, USA, 1993.

[6] Java 2 Platform, Micro Edition (J2ME). http://java.sun.com/j2me/index.jsp, 2006.

[7] Java Naming and Directory Interface. http://java.sun.com/products/jndi/docs.html, 2006.

[8] Java Transaction API. http://java.sun.com/products/jta, 2006.

[9] 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*, 54:99–152, 2001.

[10] A. Leff and J. T. Rayfield. Programming models and synchronization techniques for disconnected business applications. *Advances in Computers*, XX:XX–XX, 2006. accepted for publication.

[11] IBM Websphere MQ Everyplace. http://www-306.ibm.com/software/integration/wmqe/, 2006.

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

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

[14] Open Mobile Alliance (OMA), SyncML. http://www.openmobilealliance.org/tech/affiliates/syncml/syncmlindex.html, 2006.