Transaction Processing in PRO-MOTION

Gary D. Walborn and *Panos K. Chrysanthis*
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
{gwalborn,panos}@cs.pitt.edu

Abstract

To provide data consistency in the presence of failures and concurrency, database methods will continue to be important to the processing of shared information in a mobile computing environment. Motivated by the need to migrate existing database applications while supporting the development of new database applications and personal services involving mobile and wireless data access, we have developed PRO-MOTION. PRO-MOTION is a mobile transaction processing system that supports disconnected transaction processing in a mobile client-server environment. In this paper, we present the specifics of the structuring and the management of transactions in PRO-MOTION, which utilizes nested-split transactions to provide different levels of isolation and transaction consistency.

Keywords: mobile transactions, data caching, commit processing, disconnected database operations

1 Introduction

We are in the midst of a mobile revolution. Just as cellular telephones have forever altered the way we communicate, a new generation of mobile (even hand held) computers will change the way that we compute. Laptop computers have become a modern business necessity and smaller, lighter computers are on the way. In a trend started by personal digital assistants (PDAs) like the Apple Newton, handheld computers and "palmtops" are taking mobile computing to the masses. 3com, for instance, has sold over a million of its pocket-sized PDA, the Palm Pilot. The introduction of Windows CE, an operating system designed specifically for handheld computers and embedded systems, will provide a standard API and user interface for a number of various platforms, processors, and form-factors. The sub-$500 street price of these midget marvels will be a justifiable business expense for the executive "on the go" and an attractive option for the hobbyist or homemaker.

Perhaps as important as the advent of small, low-cost, powerful, and energy efficient mobile computers is the emergence of a wireless communication infrastructure to support mobile computing. For example, the industry has finally settled on a Wireless Ethernet standard (802.11), GSM phones deliver digital data in many urban areas, cellular modems have become commonplace, and some form of wireless digital information is available at every level, from the micro-cell in an office to a global wireless network. Modems are still the most common means of data exchange among portable computers, but new radio-based wireless schemes and even infrared light are becoming popular conduits for mobile data communications. There is no sign, however, that any of these wireless communication methods will approach, let alone exceed, the speed of a copper wire or a fiber optic link. In addition, the wireless link is inherently less reliable than a "hard wired" network connection. The wireless connection may be lost because of computer movement, battery exhaustion, or even some planned event, such as deliberate suspension (or "sleep") to conserve remaining battery power.

Therefore, despite improvements in mobile computing technology, the mobile computing environment still includes limitations on communication bandwidth, storage capacity, battery life, and processing speed. Network disconnection, however, is not failure, and, if the data and methods needed to complete a task are already present on the mobile host (MH), processing may continue even though disconnection has occurred. The goal of mobile software research is, therefore, to provide as much of the functionality of network computing as

This material is based upon work partially supported by the National Science Foundation under grants IRI-95020091 and IIS-9812532.

SAC '99, San Antonio, Texas

possible within the confines of the mobile computer's capabilities. This means, in the context of database applications, that mobile users should have the ability to both query *and update* public, private, and corporate databases. Such databases typically utilize transactions to provide data consistency and reliability in spite of concurrent updates and system failures. As a result, transaction processing and efficient update techniques for mobile and, in particular, disconnected operations, have recently attracted some attention.

Most transaction processing systems are built upon the *client-server* paradigm. If the clients are mobile computers, all communications must be completed over a wireless connection. In this environment, a disconnected client is virtually powerless because all queries and updates must be executed at the server. It becomes clear, therefore, that to process transactions while disconnected, we must keep data on the mobile computer and manage database operations locally, a process known as *data shipping* or *caching*.

The first systems that were intended to support processing on a disconnected mobile, cached data at the file level [12, 20, 14] and provided consistency by means of optimistic concurrency control. While many operations can be supported at the file level, database requests often operate on extremely large files. Caching (or hoarding[1]) several database files would require too much storage on the typical mobile host (with a 3 to 5 gigabyte hard disk). Even if a mobile host were able to cache the entire database, file level caching complicates efforts to maintain data consistency. File systems that support disconnection treat the file as the basic unit of caching and consistency, which, due to optimistic concurrency control, would lead to wasted work since any two update operations by different transactions conflict. To prevent wasted work (important in a heavily loaded system), database systems typically do not use the database file as the unit of consistency. To be practical, therefore, mobile transaction processing requires finer granularity for caching and control than that offered at the file level.

New, efficient storage management techniques are not the only requirement of mobile transaction processing. If the mobile is to process transactions while disconnected, there must be some type of transaction management process resident in the mobile host to provide local applications with concurrent database access. And because the structure and behavior of the database "client" has changed, we must make significant changes to the database server or provide an interface between our traditional server and the mobile client.

To simplify the management of disconnected transactions, some proponents of mobile transaction processing advocate new mobile transaction models, e.g., [6, 4], and/or correctness criteria for data consistency that are weaker than the standard serializability, e.g., [17, 22], so that they can cope more effectively with the restrictions of mobility and wireless communication. Even though many applications do not require strict serializability, there are important applications, including existing business applications such as inventory databases [13], that require the data consistency guarantees offered by serializability. Because the traditional techniques for providing serializability (e.g. transaction monitors, schedulers, locks) do not function properly in a disconnected environment, new mechanisms must be developed expressly for the management of mobile transaction processing. For this reason, we have developed PRO-MOTION, a mobile transaction processing infrastructure to support disconnected transaction processing. PRO-MOTION is built upon a generalized, multi-tier client-server architecture. Its architecture and fundamental building block, the *compacts*, were presented in [23, 24]. The focus of this paper is on the specifics of the management of transactions in PRO-MOTION.

In the next section, we elaborate on mobile transaction requirements. In Section 3, we briefly review the fundamentals of PRO-MOTION while in Section 4, we discuss the *nested-split transaction* which is the underlying transaction processing model of PRO-MOTION. Transaction execution and commitment on mobile clients is discussed in Section 5. Finally, in Section 5.4, we discuss the mechanisms in PRO-MOTION which define and enforce the correctness of transactions which involve a number of compacts with varying correctness criteria.

2 Mobile Transaction Requirements

In order to better describe local transaction processing on an MH, it is often helpful to generalize the traditional ACID properties[2] and talk instead about *visibility, consistency, permanence,* and *recovery.* Visibility, for instance, refers to the ability of a transaction to see the effects on data items caused by other transactions. To reduce the cost of recovery, the effects of a transaction are usually not made visible until the transaction commits and the changes are made permanent in the database. Allowing new transactions to see uncommitted changes (dirty data) may result in unwanted dependencies and cascading aborts. But since no updates on a disconnected MH can be incorporated in the server database, subsequent transactions using the same data items normally could not proceed until connection occurs and the mobile transaction commits. By making the results of a transaction visible as soon as the transaction *begins* to commit at the MH, we can allow additional transactions to progress even though the data items involved have been modified by an *active* (i.e., non-committed) transaction, similar to altruistic locking [19]. Making data available at the beginning of transaction commitment leads to the notion of *local*

[1] Hoarding is the caching of data before the actual demand based upon the possible data requirements of the host.

[2] ACID: *Atomicity* - if any of the operations contained in a transaction are executed, all of the operations in the transaction are executed. *Consistency* - any transaction, executed singly against a "correct" database, completes with the database in a "correct" state. *Isolation* - each transaction executes independently of other transactions. *Durability* - once *committed*, the effects of a transaction become permanent in the database, ensured to survive any failure.

visibility and *local* (vs. *server*) commitment which can, in turn, reduce the blocking of transactions during disconnected transaction processing while minimizing, but not eliminating, the probability of cascading aborts.

A *complete* database management system to support transaction processing on a disconnected mobile must include:

- a means of interfacing with stationary database servers,
- a stand-alone transaction processing subsystem to execute on the MH,
- a method to reconcile transactions processed while disconnected with the server database,
- sufficient logging, checkpoint, and recovery systems to mitigate system failures, and
- a way to manage the replication and consistency of needed data.

Such a system should provide serializability when needed, but should also support more relaxed correctness criteria and different degrees of transaction isolation, where appropriate. The system must be frugal with mobile system resources, such as bandwidth, energy, and storage, but powerful enough to adapt to changing conditions, such as remaining battery life or imminent disconnection. It should be extensible enough to seamlessly support new data types and correctness criteria and new recovery and caching schemes, but basic enough to support legacy applications and database servers. Because the mobile network may include multiple servers and various mobile computers, the system should be platform independent and able to support a heterogeneous mixture of clients and hosts. Finally, the system should be written to allow integration into new mobile computers and servers or as an add-on to existing object relational database systems.

Several systems have been proposed to support updating data on a disconnected MH. Most of these incorporate some notion of local visibility and local commitment with reconciliation of work done locally with the server database when reconnection occurs. But many such systems implement correctness criteria weaker than serializability and are, therefore, unsuitable for applications which demand the correctness measure that serializability provides. Others require specially constructed applications to deal with failed tentative commits and changing mobile conditions and, therefore, do not support existing database applications. Of the systems that permit disconnected operations on replicated data, Bayou[5], and Odyssey[15] each exhibit some of the qualities that we feel are essential to support transaction processing on the MH but fail to address all of the issues surrounding tentative commitment, server consistency, flexibility, recovery, and transparency. In the next section, we will briefly present PRO-MOTION, our proposed solution to the difficulties associated with disconnected transaction processing.

3 PRO-MOTION

PRO-MOTION is a new transaction processing infrastructure developed to deal with the problems introduced by disconnection and limited resources in mobile client-server operating environments. PRO-MOTION (Figure 1) is built upon a generalized, multi-tier client-server architecture with a mobile client agent called *compact agent*, a stationary server front-end called *compact manager*, and an intermediate array of *mobility managers* to help manage the flow of updates and data between the other components in the system. Its fundamental building block is the *compact* which functions as the basic unit of data replication for caching, prefetching, and hoarding.

A compact is, broadly speaking, a satisfied request to cache data, enhanced with *obligations* (such as a deadline), *restrictions* (such as a set of allowable operations) and *state information* (such as the number of accesses to the object). The compact represents an agreement between the database server and the mobile host. In this agreement, the database server delegates control of some data to the MH to be used for local transaction processing. The MH, in return, agrees to honor specific conditions on the use of the data set forth by the database server so that the consistency of the database is maintained when the updated data items are incorporated back into the server database. As a result, the database server need not be aware of the operations executed by individual transactions on the MH but, rather, sees periodic updates to a compact for each of the data items manipulated by the mobile transactions. Compacts are represented in our system as objects (Figure 2) which encapsulate

- the cached data,
- methods (i.e., code) for the access of the cached data,
- information about the current state of the compact,
- consistency rules, if any, which must be followed to guarantee global consistency of the data item[3],
- obligations, such as a *deadline* which creates a bound on the time for which the rights to a resource are held by the mobile host or restrictions on the visibility of locally committed updates, and
- methods which provide an interface with which the MH may manage the compact.

The management of compacts is a cooperative effort by the compact manager on the database server and the compact agent on each mobile host. Compacts are obtained from the database via requests by the MH when a real or anticipated data demand is created. If

[3]The compact author determines what sharing can be done and what correctness criteria are used. For example, if an aggregate item is involved in escrow transactions[16], it may be possible to create additional compacts with appropriate constraints to allow a distributed, shared access to the data. Or, for example, if the compact is expired, the ownership may be transferred to the the new MH and the expired compact invalidated.

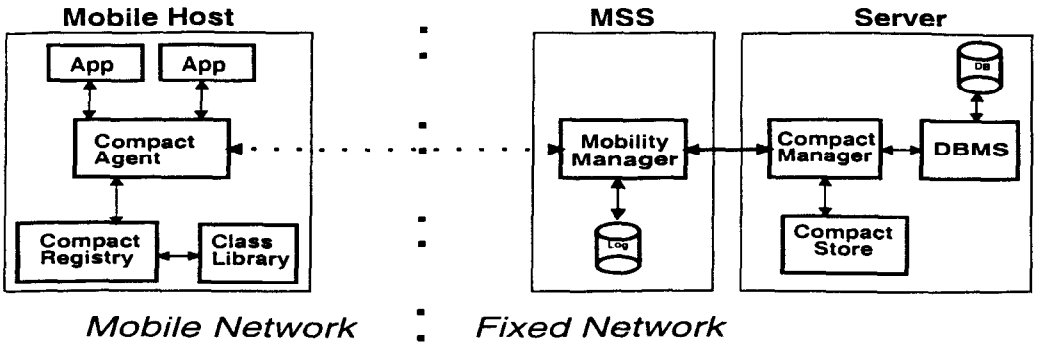

Figure 1: PRO-MOTION System Architecture

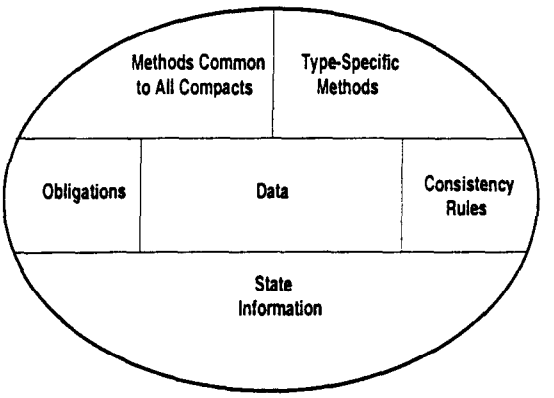

Figure 2: Compacts as Objects

data is available to satisfy the request, the database server creates a compact (with the help of the compact manager) which is recorded in the compact store and transmitted to the MH to provide the data and methods to satisfy the needs of transactions executing on the MH. The request can be tailored to cause only the transmission of missing or outdated components of a compact. In this way, transmitting the compact methods, which may be very expensive, is avoided if they are already available on the MH. Once the MH receives the compact, it is recorded in a *compact registry*, which is used by the compact agent to track the location and status of all local compacts.

Each compact has a common interface which is used by the compact agent to manage the compacts listed in the compact registry and to perform updates submitted by transactions run by applications executing on the MH. The basic set of methods necessary to manage compacts includes,

- inquire(), which retrieves useful information about the state of the compact (such as name, data type and version, cache status, outstanding transaction IDs, and remaining storage),
- notify(), used to notify the compact when the mobile environment changes,
- dispatch(), used to perform operations on the compact on behalf of transactions executing on the MH,
- commit(), to make the operations of a specified transaction permanent on the database, and,
- abort(), to abandon the changes made to the compact data by a given transaction.

The implementation of a common interface simplifies the design of the compact agent and guarantees the minimum acceptable functionality of a specific compact instance. In addition, each compact may contain specialized methods which support the particular type of data or concurrency control specific to that particular compact.

Compacts are managed by the compact agent, which is much like the daemon responsible for cache management in the CODA file system [12] in that the compact agent handles disconnections and manages storage on a MH. It monitors activity and interacts with the user and applications to maintain lists of items which are candidates for caching. However, unlike the CODA daemon, or other cache managers, the compact agent is actively involved in *transaction processing* on the mobile host, acting as a transaction manager for transactions executing on the mobile host. As such, the compact agent is responsible for concurrency control, logging and recovery.

4 Nested-Split Transactions

As discussed earlier, disconnected transaction processing introduces another level of complexity, namely, the need to provide for two levels of commitment and different degrees of visibility or isolation. It is possible, for example, that the MH may want to commit a transaction locally and make the results of that transaction available on the MH before they are actually incorporated in the server database. In order to provide for local visibility and incremental updates to the server database without compromising consistency, permanence, or recovery. PRO-MOTION uses *nested-split* transactions[4, 18] as its infrastructure. In fact, PRO-MOTION considers the entire mobile sub-system as one extremely large, long-lived transaction which executes at the server with a subtransaction executing at each MH. Each of these MH subtransactions, in turn, is the root of another nested-split transaction. Individual transactions on the MH form additional subtrees. The exact nesting and structure of the subtrees is dependent upon the commit semantics imposed by the commit processing on the MH.

When reconnecting to the database after a disconnection the MH identifies a group of compacts whose states reflect the updates of the locally committed transactions. The transactions in this subset are split from

uncommitted transactions and communicated to the compact manager, which creates a split transaction for this very group of updates. The compact manager then commits this split transaction into the database making the updates visible to all transactions (fixed and mobile) waiting for server commitment. All of this should happen without releasing the locks held by the compact manager root transaction.

Because most database servers employ a flat transaction model, the best implementation of PRO-MOTION would include a database server which "knows" about compacts and supports the nested-split transaction model. However, we can mimic the appropriate behavior by limiting *all* database access to the compact manager. If the compact manager is the only means to access the database, we can consider every item in the database implicitly locked by the root transaction. When an item is needed by a MH, the compact manager can read the data value and immediately release any actual (i.e., server imposed) locks on the data item, knowing that it will not be accessed by any transaction unknown to the compact manager. During the reconnection, the compact manager locks the items necessary for the "split transaction", writes the updates to the data items, commits the "split transaction", and re-reads and releases the altered items, maintaining the implicit lock.

5 Transaction Processing in PRO-MOTION

The interaction of the compact manager, compact agents, compacts, and the network connection implicitly suggest four transaction processing activities performed by the compact agent:

- hoarding - the mobile host is connected to the network and the compact manager is storing compacts in preparation for an eventual disconnection,
- connected processing - the mobile host is connected to the network and the compact manager is processing transactions,
- disconnected processing - the MH is disconnected from the network and the compact manager is processing transactions locally, and
- resynchronization - the MH has reconnected to the network and the compact agent is reconciling the updates committed during the disconnection with the fixed database.

Due to space limitations, we will describe only the hoarding, disconnected processing, and resynchronization activities in this paper. Intentional termination of the network connection to save power can force disconnection and a resumption of the network connection begins MH resynchronization. Therefore, connected processing is not necessary to achieve complete functionality of the system, but rather, simply allows for an optimized mode of operation which takes advantage of the network connection to quickly propagate data and status changes to the fixed network. We could easily simulate disconnections following each resynchronization to force the MH to remain in disconnected transaction processing.

5.1 Hoarding and Caching

Hoarding utilizes a list of resources required for processing transactions on the mobile host. Each resource request is implicitly or explicitly associated with a specific compact type. The resource list is built and maintained on the MH by interactions with the compact agent, mobile applications, and the user. The compact agent adds items to the list by monitoring the usage of items by applications running on the MH. If an application attempts to access data not in the list, the compact agent immediately adds the item to the list which initiates an attempt to obtain the data item.

An application can maintain the list directly by passing a request to the compact agent. A program written specifically for PRO-MOTION can have code executed periodically (e.g., when the application is loaded to the MH or when the connection state is about to change) that will update the list. A specific application may make a "best guess" about resources which will be needed during the disconnection based on information provided at the MH, such as the identity of the user or the current MH location. Also, since any program can make additions to the required resource list, an application can be written that allows the user to provide input about what resources must be kept available for disconnected operation. In this way the hoarding behavior may be extended or modified, if desired.

Multiple requests for a single resource may be included in the resource request list. Because various applications may have need for the same item, one entry for each item may exist per application with similar or differing access requirements. For example, perhaps application A requires item x with read and write permissions and application B requires item x with read-only access. If application A is later removed from the MH, the item will still be required, but the level of access may be downgraded to read-only. If, on the other hand, B is removed and A remains, the associated request from B is removed and the compact will be adjusted to comply with the requirements the remaining request(s).

While the MH is connected to the fixed network, the resource request list is scanned for new, unsatisfied requests. The compact registry forwards such a request to the compact manager. The compact manager checks to see if the given resource is currently involved in a compact with the requesting MH. If the requesting MH already holds the resource, the compact manager forwards the new request to the existing server-side compact. The existing server-side compact determines if the access privileges in the new request can be satisfied. If so, an update message is transmitted to the existing client-side compact revising the access rights of the compact.

If the needed resource is currently involved in mobile transaction processing with another MH, the compact manager forwards the new MH request to the

appropriate server-side compact. The related compact determines whether or not the request can be satisfied allowing concurrent caching of the data across two MHs. If the request can be satisfied, a MH-side compact with appropriate constraints and obligations is created and queued for transmission. If not, a null compact is queued to be sent to the MH. In any case, the deadline for the compact, the compact constraints, and the actual data items returned are determined by the nature of the request, the current usage at the server, and pre-programmed parameters of the compact type. Compacts generated at the compact manager are forwarded to the MH and stored in the compact registry where they are accessed by the compact agent.

If the resource is freely available in the database (i.e., the resource can be locked by the compact manager), the compact manager obtains modification rights to the data and finds the matching compact type. The compact manager creates an instance of the appropriate server-side compact type with the data obtained and forwards the request to this newly created compact. The server-side compact creates an instance of a matching MH-side compact with the data and needed code and queues the new MH-side compact for transmission to the MH. If the resource is not available (e.g., held by a non-mobile transaction), a null compact is created and queued for transmission to the MH. The null compact will cause operations attempted by an application to return with an indication of a failure.

When a MH-side compact is received by the mobile host, the first message contains only the compact class specification and compact data. The compact registry checks to see if the compact code for the specified class is already resident in the mobile host. If the code is needed, the compact registry sends a request for the MH-side code to the compact manager which returns the required compact code. When the necessary compact code is resident, the compact registry creates a local instance of the desired compact type, makes an entry in the registry database, and calls the initialize() method of the new compact, passing the data received from the server. The new compact stores its initial value into a persistent object store on the MH and does any housekeeping needed to process transactions. The object store always contains at least one copy of the compact which reflects the state of the compact (including the value of the data item) after considering all successfully committed transactions on the MH. By closely following this convention, the MH is always capable of restoring the compact registry (and all associated data) to a consistent state by simply reloading the correct set of objects from the persistent store.

Compacts stored on the MH which are about to expire are also handled by the hoarding process. When a compact is about to expire, the compact registry checks the resource request list to see if the resource held by the compact is still required. If the compact is no longer required, the compact registry sends a cancelation message to the compact manager for the matching server-side compact. If the server-side compact is still valid and synchronized with the MH compact, a cancelation message is returned to the MH where the MH-side compact is removed from the registry. If valid but unsynchronized, the affected MH is notified that resynchronization is required. If the compact is invalid, an invalidation message is returned to the requesting MH. If the MH cancelation came from the last MH holding a matching client-side compact, the lock on the resource is released, the compact is destroyed, and a cancelation message is sent to the MH.

If, based upon the contents of the resource request list, the expiring compact is still needed, the compact registry sends a renegotiation message to the compact manager requesting an extension on the deadline. The compact manager forwards the renegotiation message to the server-side compact. If successful, a status message is queued for the MH with an updated deadline. If not, an invalidation message is queued for transmission.

5.2 Disconnected Transaction Processing

When the network link is lost by the MH, the compact agent begins processing transactions locally. During local transaction processing, the compact agent and the compacts themselves constitute a distributed transaction management system responsible for all local transaction management. Applications running on the MH access data via *events*, such as "begin transaction"(BEGIN), "perform operation"(OP), "commit"(COMMIT), or "abort"(ABORT). In applications written specifically for PRO-MOTION, these events may be sent directly to the compact agent. If legacy applications are to be supported on the MH, there may be a "front end" which intercepts calls from the legacy application and converts those calls to PRO-MOTION events which are sent on to the compact agent.

The compact agent maintains an event log, which is used to manage transaction processing, recovery, and resynchronization on the MH. Events which do not involve a specific compact are sent directly to the event log by the compact agent. Events destined for a specific compact, such as operations to be performed on a data item, are sent directly to the compact. The compact filters these events and returns a log entry along with the results of the event. This filtering allows the compact to discard irrelevant events and create optimized log entries for recovery purposes, saving log storage.

5.2.1 Initiating a transaction

An application on the MH initiates a transaction by issuing a BEGIN event. Upon receiving a BEGIN event, the compact agent assigns a transaction ID which is unique on the MH. The BEGIN event, along with the transaction ID, is written to the event log. PRO-MOTION allows a number of options to the BEGIN event to further specify transaction behavior. Two such options control the transaction's *level of isolation* (discussed in a later section) and *local commitment*. If the BEGIN event contains a LOCAL option, the transaction will be permitted to commit locally and make its results visible to other transaction on the MH, accepting

the possibility of an eventual failure to commit at the server. Transactions which do not have a LOCAL option will not commit locally until the updates have committed at the server. Because the options are attached by the compact agent to each operation sent to the compacts, a specific compact may choose to disallow operations by transactions that wait for server commitment and block local transactions. Or, if it is crucial that updates be propagated to the server database, a compact may be conditioned to accept only traditional (i.e., not LOCAL) transactions.

The BEGIN event may also contain a group ID. If no group ID is present in the BEGIN event, the compact agent assigns a group ID to the transaction and returns the group ID to the new transaction. The application can pass this information to additional transactions, allowing the additional transactions to "join" the transaction in progress. Additional transactions that include this group ID receive the same group ID but a unique transaction ID. All transactions with the same group ID are commit-dependent upon all other transactions in the group (i.e., all transaction in the group commit together or abort together). Compacts may be written to honor the group ID to allow visibility of uncommitted data to a group of transactions. This allows the data engineer to create compacts which are capable of supporting cooperative transactions without relying on local commitment. It is important to note that not all compacts need honor the group ID. Compacts must be specifically designed to allow group access to uncommitted data.

5.2.2 Transaction execution

As an application executes and needs to perform data access, an OP event is sent to the compact agent with the ID of the compact, the operation to be performed, any parameters for the operation, and the transaction ID and group ID. The compact agent uses the ID of the compact and invokes the compact's dispatch() method with the OP event as a parameter. The compact determines if there is a conflict with the new operation and any pending (i.e., non-committed) operation. If the operation conflicts, the OP event is returned to the compact agent with a CONFLICT return code and a list of conflicting transaction ID(s). The compact agent then queues the OP event for dispatch to the compact when the compact state changes. The information saved in the queue is sufficient for the compact agent to detect and resolve transaction deadlocks.

If no conflict with pending operation(s) is detected, the compact performs the operation against the memory-resident value of the data item and returns an event with:

- the value returned by performing the operation, which must be communicated to the appropriate application,
- an indication that the operation was performed successfully, and
- a record (possibly null) to be written to the event log with enough information to undo or redo the operation.

The compact agent writes the returned event to the event log and returns the result of the operation to the calling application.

5.2.3 Transaction completion

Because a number of compacts may be involved in a single local transaction, the commitment of a transaction is performed using a two-phase commit protocol where all participants reside on the MH. Transaction commitment is initiated upon the receipt of a COMMIT event from the application. Unlike most transaction processing systems, PRO-MOTION allows a *contingency procedure* to be attached to each COMMIT event and logged with the COMMIT event. Because each transaction committed during disconnection represents a local, tentatively committed transaction, there exists the possibility that the transaction will never commit at the server. The contingency procedure is saved to be executed in the event that a locally committed transaction cannot be incorporated in the server database state. When a COMMIT event is received from an application, the compact agent coordinates a two-phase commitment, sending and logging PREPARE and COMMIT messages as appropriate. Each participating compact, upon receiving the COMMIT event, invalidates the previously saved object state in the persistent store, validates the newest committed state, and resumes accepting events. Upon receiving an ABORT event, a compact reloads an appropriate consistent state[4] from the persistent object store and resumes processing with the next event.

5.2.4 Failure recovery

Should the MH fail without warning (e.g., shut down due to low battery voltage) during disconnection, a recovery protocol will be executed. Our recovery protocol assumes that any transaction pending (i.e., not committed) when the failure occurs will be aborted. When the compact agent is restarted after the failure it examines the log and removes records from uncommitted transactions. The connection state is determined and appropriate processes are restarted. Because a set of compact states that constitute a consistent state is assumed to be present in the compact store, transaction processing resumes. If the compact agent attempts to dispatch an operation to a compact that is not in main memory, the compact registry creates an instance of the compact and calls the recover() method. The compact attempts to reload the state from the persistent store and finds that it was in the process of commit processing. The compact then queries the event log to determine whether the commitment was successful and reload the correct stored state. Compacts not involved in commit processing at the time of the failure reload the last consistent state from the object store and resume processing.

[4] It is possible to write a compact that supports multiple versions, which may result in a number of states in the store.

5.3 Resynchronization

When a disconnected MH finally reconnects to the network, resynchronization begins. Resynchronization brings the server database into agreement with the changes performed on the MH while disconnected. Resynchronization requires two, three, or four stages, depending on the validity of the compacts cached on the MH. Every attempt is made to incorporate changes from all transactions locally committed by the MH into the database. Nonetheless, it may not be possible to commit every eligible transaction at the server. The contingency procedure of each transaction that cannot ultimately be committed is executed to compensate. The process begins when the compact agent sends a BEGIN SYNCHRONIZATION event to the compact manager, freezing the state of all compacts associated with the MH.

In the first stage of resynchronization, the compact registry finds the IDs of all compacts which have been modified since the MH was disconnected. If all of the compacts are valid (i.e., have not expired), the resynchronization can be complete and moves into the final stage. If, however, modified compacts have expired, additional work must be completed before resynchronization ends.

In the second stage, the compact registry transmits the IDs of expired modified compacts and attempts to renew each compact and extend the deadline. If no other entity has locked or modified the compact data, the compact manager can reinstate a compact and validate the MH-side compact as if the compact had never expired. If all expired compacts are thus reinstated, the update can be made in their entirety and the resynchronization proceeds to the final stage.

If, however, modified compacts cannot be reinstated, an additional and more expensive, procedure must be completed. The original values of the valid or reinstated compacts can be obtained from the compact manager. Compacts that could not be reinstated are invalidated. At this point, the event log can be replayed for all committed transactions. If a transaction reads or modifies an invalidated compact, the transaction is considered aborted and all compacts modified by the aborted transaction are left unchanged, but marked unavailable. Similarly, transactions reading from unavailable transactions are aborted and compacts modified by these transactions are marked as unavailable. The compacts used by group transactions are considered together when invalidating compacts. When the event log has been completely replayed, the remaining set of valid compacts (including those marked unavailable) represent a subset of the locally committed transactions that can be incorporated in the server database. The contingency procedures for transactions originally marked as committed that are now marked as aborted must be scheduled to execute. Resynchronization then proceeds to the final stage.

In the last stage of resynchronization, the compact registry inspects the set of compacts for compacts that are valid and modified. Each compact is queried and generates an OP event that is returned to the server-side compact to bring the server data into agreement with the data on the MH. When all the operations have been communicated, the MH sends a COMMIT message to the compact manager. On the server, the set of operations from updates of the resynchronization process are split out into a single, separate transaction that is committed on the database server. In addition, the compact manager attempts to keep locks on all of the items thus committed. If the locks are retained by the compact manager, the associated server-side compacts and the MH-side compacts remain valid and usable. If the locks cannot be kept, the server-side compact and the associated MH-side compacts are invalidated. Once resynchronization is complete, the compacts are released and the MH returns to the hoarding state.

5.4 Correctness in PRO-MOTION

To facilitate the implementation of various mobile transaction processing schemes involving semantic correctness criteria as well as various read/write models, PRO-MOTION allows the compact designer to determine correctness criteria and concurrency control methods for each individual compact type. Because a single transaction may utilize several compacts with varying definitions of correctness, some mechanism must be introduced to classify and enforce correctness guarantees for the database, regardless of which compact methods were used to access the data.

PRO-MOTION utilizes a ten level scale to characterize the correctness of a given transaction execution. While the scale is arbitrary (i.e., the database designer can use any definition for each level in the scale), the concept is most useful when a the scale can be used to categorize the properties of a particular execution. Our current standard for PRO-MOTION is loosely based upon the degrees of isolation defined in the ANSI SQL standard as extended in [2]. We define "Level 9" to represent an actual serial execution of transactions and "Level 8" to represent a serializable execution. Each succeeding level represents a lesser degree of isolation. A "Level 0" operation makes no guarantee whatsoever about the correctness of an operation.

Each method in a compact (corresponding to an operation on the compact data) is assigned an isolation level and a mode indicating whether the operation is a READ operation or a WRITE operation. Any operation which may modify the value of a data item will be categorized as a WRITE operation (e.g., increment() and decrement() for an escrow[16] item). It is interesting to note that a given compact may have multiple methods for a given data item that employ differing levels of correctness. For example, an escrow compact may have "Level 8" (serializable) isolation for increment() and decrement() methods, and a "Level 3" or "Level 4" certification for a relaxed-read() method which returns the expected value of a data item if all outstanding transactions commit. Furthermore, a compact may offer more than one level of isolation for the same operation by providing alternate methods which enforce different levels of isolation. For example, a single compact may provide a read-strict() operation which only completes

if the item can be locked, and a read-relaxed() method which returns a value regardless of the lock status of the data item. The read-strict() method may carry a "Level 8" guarantee while the read-relaxed() operation may carry only a "Level 4", for example.

During transaction processing, each transaction may be assigned a minimal READ level and a minimal WRITE level. These levels can be specified as optional arguments to the BEGIN event for the transaction. Each system will have default READ and WRITE transaction levels assigned by the database administrator. (It is probable that the default READ and WRITE levels would be set at "Level 8", forcing serializable behavior.) No operation will succeed if its isolation level is lower than the corresponding READ or WRITE level of the transaction performing the operation.

Now, having set these ground rules, we are ready to explain how the interactions across compacts can be handled and to address the isolation guarantees provided by a transaction which operates on a variety of compacts. Because dirty data provided by a compact with relaxed correctness criteria could corrupt data written by a compact which guarantees strict correctness, it is necessary that any modification to the database be based on reads that are "at least as strong" as the write to be performed. Based on this premise, PRO-MOTION applies the following method:

As each transaction commits, the set of operations is scanned for the *lowest* guarantee among any of the READ operations and the *highest* requirement among any of the WRITE operations. A given transaction will only be allowed to commit (first locally and then, during reconciliation, globally) if:

- None of the WRITE operations is lower than the WRITE level of the transaction,
- None of the READ operations is lower than the READ level of the transaction,
- All of the involved compacts agree that the transaction is correct (through the 2PC protocol),
- The lowest level of any READ operation is greater than or equal to the highest level required by any WRITE operation.

This system ensures that the correctness of a particular transaction execution is *at least as good* as specified by the compact designer, the individuals certifying the compacts, and the application programmer, while allowing a range of correctness criteria and utilization of object semantics to achieve maximum concurrency.

6 Conclusions

PRO-MOTION provides an infrastructure and transaction model that will support the disconnected processing of transactions by a mobile host. As a mobile transaction processing system, PRO-MOTION offers many unique advantage over other proposed systems. Unlike systems which rely upon the application to enforce consistency, PRO-MOTION is *data-centric*. The compact is the basic unit of caching and consistency. Therefore, it is the compact which determines correctness criteria, granularity, and conflict determination for each data type. Furthermore, multiple compacts may be defined for a single data type, employing varying correctness criteria and supporting different degrees of isolation, allocation strategies, access privileges, or renegotiation strategies. This flexibility allows PRO-MOTION to implement a number of mobile data management schemes, including leases, check-in check-out items, cooperative activities, and escrow items, by properly implementing the compact for the data items involved. Indeed, it is possible to support different correctness criteria *at the same time* by utilizing compatible, but different compacts. At the same time, PRO-MOTION frees the application programmer from considering correctness and concurrency concerns while designing applications and prevents an errant application from violating data consistency.

PRO-MOTION is efficient because only compacts needed to manage required resources are maintained on the MH. Furthermore, because compacts are written in Java, much of the code is maintained in the Java Virtual Machine and need not be replicated in each compact. Also, the compacts are designed to share code (methods) wherever possible. Because compacts allow for varying degrees of granularity (based upon data type and modes of access) and can query available storage and modify their behavior accordingly, compacts can make more efficient use of limited mobile data storage capacity. Because caching, transaction processing, and recovery are controlled by compacts and the code contained in the compacts, these characteristics can dynamically change to reflect changing conditions on the MH. Compacts can submit queries about the mobile status and respond accordingly.

We hope that PRO-MOTION will simplify the implementation and testing of various mobile transaction processing schemes. We currently have simple compacts written in Java and are designing a database server that directly supports compacts. We are currently using laptop computers to test the mobile components of PRO-MOTION, but hope to test handheld devices as soon as a Java Virtual Machine is ported to Windows CE.

References

[1] Barbara, D. Certification Reports: Supporting Transactions in Wireless Systems. In: *Proc. of the 17th Int'l Conference on Distributed Computing Systems*, 1997.

[2] Berenson H., P. Bernstein, J. Gray, J. Melton, E. O'Neil, P. O'Neil. A Critique of ANSI SQL Isolation Levels. *Proc. of ACM SIGMOD Conference*, 1995, pp. 1-10.

[3] Bernstein P. A., V. Hadzilacos and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[4] Chrysanthis P. K. Transaction Processing in a Mobile Computing Environment. In: *Proc. of IEEE Workshop on Advances in Parallel and Distributed Systems*, 1993, pp. 77-82.

[5] Demers A., et al. The Bayou Architecture: Support for Data Sharing among Mobile Users. In: *Proc. of the*

Workshop on Mobile Computing Systems and Applications, 1994.

[6] Dunham, M., A. Helal, and S. Balakrishnan. A Mobile Transaction Model That Captures Both the Data and Movement Behavior, *ACM/Baltzer Journal on Special Topics in Mobile Networks.*

[7] Gray C. G. and D. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In: *Proc. of 12th ACM Symposium on Operating Systems Principles*, 1989, pp. 202–210.

[8] Imieliński T. and B. R. Badrinath. Mobile Wireless Computing: Challenges in Data Management. In: *Communication of ACM*, 1994, 37(10):18–28.

[9] Ioannidis J., D. Duchamp and G. Q. Maguire. IP-Based protocols for mobile internetworking. In: *Proc. of ACM SIGCOMM Symposium on Communication, Architectures and Protocols*, 1991, pp. 235–245.

[10] Jain R. and K. Narayanan. Network Support for Personal Information Services to PCS Users. In: *Proc. of IEEE Conference Networks for Personal Communications*, 1994.

[11] Jing, J., O. Bukhres, and A. Elmagarmid. Distributed Lock Management for Mobile Transactions. In: *Proc. of the 15th Int'l Conference on Distributed Computing Systems*, 1995, pp. 118-125.

[12] Kistler J. and M. Satyanarayanan. Disconnected operation in the Coda file system. In: *ACM Transactions on Computer Systems*, 1992, 10(1):3–25.

[13] Krishnakumar N. and R. Jain. Protocols for maintaining inventory databases and user service profiles in mobile sales applications. In: *Proc. of the Mobidata Workshop*, 1994.

[14] Kuenning G. and G. J. Popek. Automated Hording for Mobile Computers. In: *Proc. of the 16th ACM Symposium on Operation System Principals*, 1997.

[15] Lu, Q., Satyanarayanan, M. Isolation-Only Transaction for Mobile Computing. *Operating Systems Review* 28(2):81-87, 1994.

[16] O'Neil P. The Escrow Transactional Method. In: *ACM Transactions on Database Systems*, 1986, 11(4):405–430.

[17] Pitoura E. and B. Bhargava. Maintaining Consistency of Data in Mobile Distributed Environments. In: *Proc. of 15th Int'l Conference on Distributed Computing Systems*, 1995, pp. 404–414.

[18] Ramamritham K. and P. K. Chrysanthis. *Advances in Concurrency Control and Transaction Processing*, IEEE Computer Society Press, 1996.

[19] Salem, K., H. Garcia-Molina, and J. Shands, "Altruistic Locking," *ACM Transactions on Database Systems*, Vol. 19, No. 1, 1994, pp. 117–165.

[20] Tait D. C., H. Lei, and H. Chang. Intelligent File Hoarding for Mobile Computing. In: *Proc. of the Workshop on Mobile Computing*, 1995, pp. 119–125.

[21] Yeo L. H. and A. Zaslavsky. Submission of Transactions from Mobile Workstations in a Cooperative Multidatabase Processing Environment. In: *Proc. of the 14th Int'l Conference on Distributed Computing Systems*, 1994.

[22] Walborn G. and P. K. Chrysanthis. Supporting Semantics-Based Transaction Processing in Mobile Database Applications. In: *Proc. of the 11th Symposium of Reliable Distributed Systems*, 1995, pp. 31–40.

[23] Walborn, G. and P. K. Chrysanthis. PRO-MOTION: Management of Mobile Transactions. In: *Proc. of the Symposium on Applied Computing*, 1997, pp. 101–108.

[24] G. Walborn and P. K. Chrysanthis. "PRO-MOTION: Support for Mobile Database Access," *Personal Technologies*, Vol. 1, No. 3, 1997, pp. 171-181.