PRO-MOTION: Support for Mobile Database Access

Gary D. Walborn and Panos K. Chrysanthis

Department of Computer Science, University of Pittsburgh, Pittsburgh, PA, USA

Abstract: In order 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. It is important, therefore, that we develop transaction processing systems that accommodate the limitations of mobile computing, such as frequent disconnection, limited battery life, low-bandwidth communication and reduced storage capacity, so that we can migrate existing database applications to mobile environments. In this paper, motivated by these needs, we propose a mobile transaction processing system that supports disconnected transaction processing in a mobile client–server environment. The proposed system employs compacts, which encapsulate access methods, state information and consistency constraints, to allow for local management of database transactions on mobile computers.

1. Introduction

Several exciting advances in wireless technology have made data access by mobile computer users possible anywhere, anytime, in any way. To be effective, mobile users should have the ability to both query and update public as well as private corporate databases which typically utilise transactions to provide data consistency and reliability despite concurrent updates and system failures. Thus far, the focus on data management in mobile environments has been primarily on supporting efficient data retrieval while attempting to minimise energy consumption by the mobile computer [1–3]. Transaction processing and efficient update techniques for disconnected mobile operations have just recently attracted some attention.

Our goal has been to devise methods to allow remote database access and update by mobile computers regardless of connection status and despite the various limitations introduced by mobility and portability. Furthermore, we want to be able to support existing (legacy) database applications without modification and, at the same time, support the development of new applications which respond to the mobile environment. To this end, we have developed PRO-MOTION [4], a flexible and adaptive infrastructure for the support of transaction processing in a multi-tier, mobile client–server operating environment. It allows mobile clients to continue executing competing transactions on data items cached locally while they are moving and not connected to the network, incorporating the modified data back into the database when reconnection occurs. In this paper, we will provide a high-level description of PRO-MOTION, its goals, and, in particular, its fundamental building block, the *compact*. Compacts are PRO-MOTION's basic unit of caching and consistency and the basis for PRO-MOTION's flexibility and adaptability to support a number of concurrency control methods which provide varying levels of data consistency.

In our discussion in this paper, we assume a general mobile computing environment (see Fig. 1) in which the network is made up of *stationary* and *mobile* hosts (MHs) [5]. Unlike stationary hosts, MHs change location and network connections while computations are being performed. MHs maintain their connection to the high-speed fixed network by means of specialised stationary hosts, called Mobility Support Stations (MSSs), which are equipped with compatible wireless communications capabilities. We will assume that, at any given instant, each MH is either connected to the network by a specific MSS or completely disconnected from the fixed portion of the network. The logical or physical area served by a single MSS is called a *cell*.

MHs are, in general, less robust than stationary hosts. MHs typically have limited battery life, reduced storage capacity, and are subject to physical hazards, such as falls, immersion, and theft. In addition, the wireless connection to the stationary network tends to be low-bandwidth, expensive, and tenuous. As a result, MHs may become disconnected from the stationary portion of the network if they move beyond the range of any MSS (for example, *mh_1* in Fig. 1). Also, an MH becomes disconnected if the communication sub-

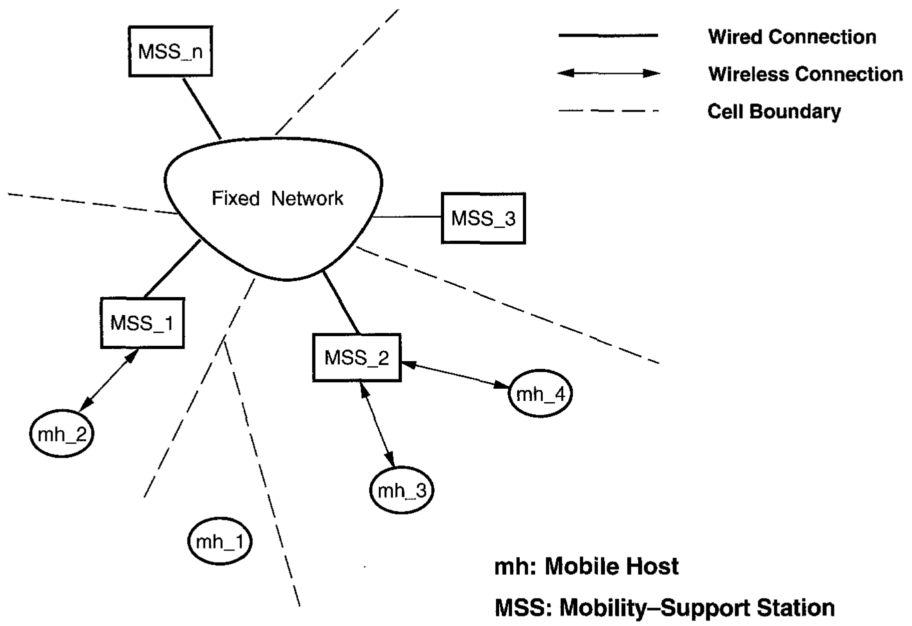

Fig. 1. Mobile network.

system is powered down to conserve energy, save money, and extend battery life. Such a disconnection (unlike a dead battery) does not imply the failure of the MH and processing may continue as long as the MH has the needed resources (e.g. database items or objects) stored locally.

The rest of the paper is structures as follows. In the next section, we provide a broad overview of transaction processing concepts and elaborate on the motivations underlying PRO-MOTION. In Section 3, we present a real-world scenario which demonstrates some of the various requirements of mobile transaction processing. In Section 4, we describe our approach to mobile transaction processing, introduce the notion of *compacts*, discuss the advantages they provide and show how they can be used to support our motivating application. Finally, in Section 5, we present our conclusions and suggest areas for further inquiries and testing.

2. Background

In this section, we first briefly review the notion of transactions and how transactions can provide advantages for the building of reliable distributed systems. We then discuss the current proposals for mobile management of data including mobile file systems.

Just as database management systems have proven important to guarantee data consistency in multi-user systems, *transactions* have proven a useful abstraction for database systems. A transaction is a set of interdependent operations on a database that perform a function or carry out a task. Transactions can be used to model a variety of 'real world' activities. Since partial completion of a task can result in data inconsistencies and corrupt the database, a transaction either executes in its entirety, completing the task, or has no effect on the system. That is, each set of operations in a transaction is either *committed*, making the changes permanent in the database, or *aborted*, obliterating all effects and leaving the database unchanged.

Traditionally, each transaction is assumed to have the following ACID properties:

- 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.

By definition, a 'correct' database will remain correct after the execution of any number of transactions in a *serial* fashion (i.e. one at a time). When transactions are executed concurrently, we can guarantee the correctness of the database if the concurrent execution of the transactions results in a database state identical to that achieved by *some* serial execution of the same transactions (i.e. *serialisability*) [6]. The serialisability of transactions is the basis for the correctness and reliability of database systems and enormous strides have been made to improve the concurrency control, recovery, and commit processing protocols to ensure serialisability and the ACID transaction properties [7].

Most current transaction processing systems are built upon the *client–server* paradigm. In a client–server system, the database resides on one or more large (and, presumably, fast) computers called *servers*. The application programs are actually executed on smaller, connected computers known as *clients*. Each operation is communicated to the server, which executes the operation against the database and, in turn, communicates the result to the client. If the clients are mobile, all communications must be completed over the wireless connection. In typical client-server environments, a disconnected client is powerless because all queries and updates must be executed by the server. It becomes clear, therefore, that to process transactions while disconnected, we must keep data on the MH and manage database operations locally.

In order to better describe local transaction processing on an MH, it is often helpful to generalise the traditional ACID properties 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 which are still executing, whereas recovery is the ability to take the database to some state that is considered correct, not necessarily reflecting updates from *all* committed transactions. Traditionally the effects of a transaction are not made visible until the transaction commits and the changes are made permanent in the database. Since no transactions on a disconnected MH can be incorporated in the server database, subsequent transaction using the same data items could not proceed until connection occurs and the mobile transaction commits. By making the results of a transaction visible as soon as it *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 transaction. This leads to the notion of *local* visibility and *local* (vs. *global*) commitment.

If the system is augmented to operate upon local copies of the data (e.g. *data shipping*), the dependence on the server can be reduced but more resources will be consumed on the MH and the availability of data to the mobile host can still be crippled by disconnection. We might be tempted to tolerate such a handicap as a penalty for mobility. We discover, however, that such a disconnection threatens system-wide throughput. Many systems use *lock-based* concurrency control to ensure controlled access to data to prevent data inconsistencies and preserve database integrity constraints. Suppose, for example, that an MH sends a lock operation to the server in preparation for an update and then disconnects. In most database systems, the locked data would be unavailable to *any* client, stationary or mobile, until the MH reconnects and releases the lock. Worse, if the MH is damaged and fails to reconnect at all, the resources could be held indefinitely. If, on the other hand, an *optimistic* concurrency control scheme is employed, competing transactions are allowed to access an item held by the disconnected MH and any operations performed during disconnection are likely to be invalidated, making it difficult, if not impossible, to do useful work on a disconnected MH [8]. All client–server systems are ultimately limited by the reliability and performance of the wireless connection.

The first systems that were intended to support processing by a disconnected mobile shared data at the file level and were based on data shipping or caching [9–12]. While many Unix operations can be supported at the file level, transaction processing requires finer granularity for caching and control. Recent research has attempted to develop appropriate transaction models for mobile computing. Of these, some approaches examine how traditional transactions could support remote access from a mobile computer with no local database processing capabilities, such as a *dumb terminal* [13, 14]. In contrast, approaches that aim to support transactions which perform updates at the mobile computer (e.g. [15–18]), propose new mobile transaction models and correctness criteria for data consistency that are weaker than the standard serialisability so that they can cope more effectively with the restrictions of mobility and wireless communication. Even though many applications do not require strict serialisability, there are important applications, including existing business applications such as inventory databases [19], that require the data consistency guarantees offered by serialisability.

Clearly, there is no single proposed transaction model which satisfies all of the requirements for transaction processing within the confines of mobile limitations. It is possible, therefore, that the best system for the management of mobile transactions would involve a number of transaction models and their associated concurrency control protocols and consistency guarantees. In the next section, we present a practical example which illustrates the need for a flexible mobile transaction processing system. This particular example is based upon the actual experience of a common and contract carrier and serves to illustrate the need to support multiple transaction types and varied consistency requirements.

3. Motivating Application: Commodity Dispatch

Mobile computers are becoming more and more common in the trucking industry. Each truck is fitted with a small computer which communicates via satellite or radio links to a database site managed by the trucking company. The mobile computer runs specialised software which gathers data from vehicle instruments, digitiser pens and keyboard. The collected data is used to update the corporate database and is used for billing, compliance, equipment management and driver payroll. Often electronic means will be used to transmit funds directly to the driver for fuel, tolls, permits, living expenses and repairs.

In order to better understand the requirements of mobile transaction processing, let us look at a typical scenario faced by a contract carrier which has accepted a contract to move a large quantity of fertiliser from the manufacturing facility to a number of rural farms and co-ops. If the trucks are privately owned and sub-contracted to the trucking company, each driver is free to accept or reject any load that is offered. Even though the company has little control over each individual truck, the company will be penalised if it fails to complete delivery of the fertiliser within a specified period.

Many interesting problems are raised here. For example, each truck should be offered a portion of the available loads, but care should be taken so that the quantity of fertiliser offered to all of the trucks does not greatly exceed the quantity available. If, for example, every truck is offered the entire quantity, trucks could be stranded with insufficient loads to pay for expenses. In this case, the total amount of fertiliser allocated to each truck may be allowed to vary from actual availability (controlled divergence), as long as the total variance is within limits and eventual consistency is obtained. Since a copy of available load information cached at the mobile host may carry sufficient information to make an attempted pickup, connection to the database server is not required before proceeding to the factory for a load.

As trucks arrive to load with fertiliser, each truck must obtain a shipping manifest from the trucking company. Each manifest uniquely identifies a specific load of fertiliser and indicates the location and time of loading, the pertinent information about the truck and driver, the exact measure of material loaded and a description of any exceptions that should be noted for insurance purposes. This information should be made permanent in the database before the truck is permitted to travel or receive advances for fuel. Therefore, it may be necessary to be connected to successfully complete this transaction.

Once the load is delivered, the driver records the delivery date and time, obtains a signature from the receiving party, notes any discrepancies between the delivered goods and the original bill of lading and checks the list of available loads in order to proceed to the next shipper for loading. The delivery information should be incorporated in the database as soon as possible, but a delay will only postpone the billing process. This transaction could be finalised locally, in spite of disconnection, but made permanent in the database whenever reconnection occurs.

In this scenario, one can easily identify three distinct types of transactions with respect to data consistency requirements:

- DISPATCH transactions which tolerate controlled divergence, with the possible existence of a global constraint (i.e. total offers do not exceed available fertiliser ± tolerance);
- MANIFEST (ACID) transactions, whose updates must be made permanent in the server database before any other transactions pertaining to the load may be processed (i.e. recording load information and obtaining unique manifest number); and
- BILLING transactions, which update manifest information and make their results visible locally so as to not delay the acquisition and loading of more fertiliser, but update the database only when connected.

The transaction processing system presented in the balance of this paper will provide the mechanisms

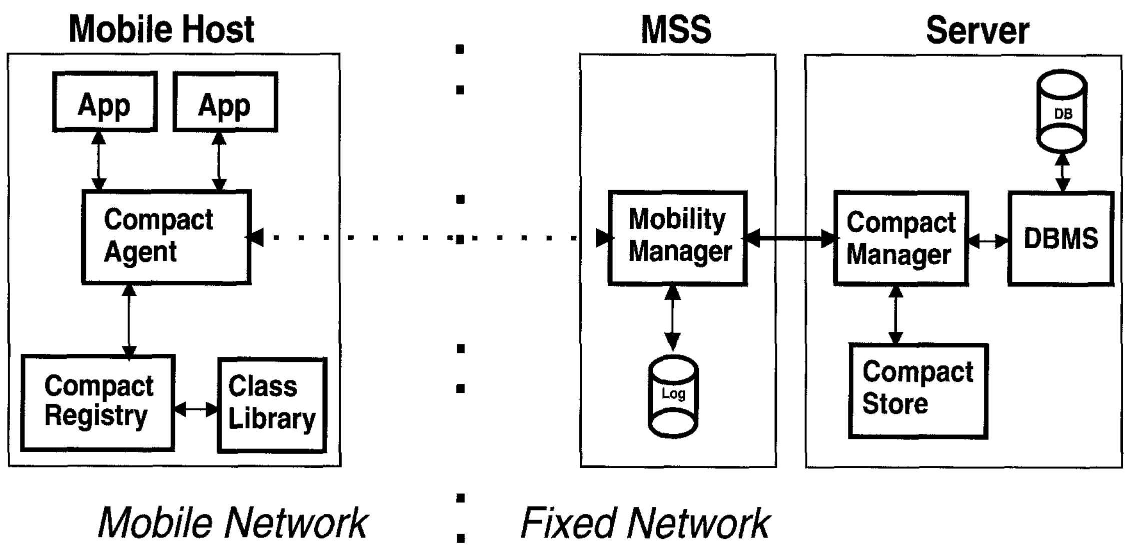

Fig. 2. System architecture.

to support these different types of transactions which maintain database consistency while allowing flexibility in coordinating and reconciling competing transactions.

4. PRO-MOTION: A Mobile Transaction Processing System

The limitations of the mobile environment present a number of challenges to traditional transaction processing systems (as discussed in Section 2). To that end, in this section, we propose a new transaction processing system, called PRO-MOTION, to deal with the problems introduced by disconnection and limited resources. The salient features of PRO-MOTION are:

- the use of *compacts*, which function as the basic unit of data replication for caching, prefetching, and hoarding,
- transaction management on the mobile host and, finally,
- exploitation of object semantics wherever possible to improve site autonomy and increase concurrency.

These features, which are described below, when used together, can minimise the handicap imposed by mobile limitations.

The high-level architecture of PRO-MOTION, shown in Fig. 2, consists of a *compact manager* at the database server, a *compact agent* at the mobile host to negotiate and manage compacts and provide local transaction management for the MH, and a *mobility manager* executing at the MSS to help manage the flow of updates and data between the other components in the system. The use of the mobility managers allows PRO-MOTION to be generalised into a multi-tier client-server architecture. In this paper, however, we will deal only with a three-tier structure.

4.1. Compacts

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 the data to the MH. The MH, in return, agrees to assume responsibility for the data and to honor specific conditions set forth by the database server. 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 (Fig. 3) which encapsulate

- the cached data;
- methods (i.e. code) for the access of the cached data;

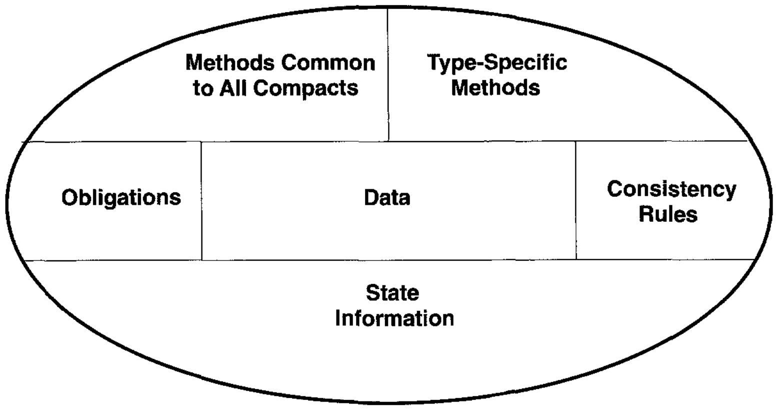

Fig. 3. Compacts as objects.

- information about the current state of the compact;
- consistency rules, if any, which must be followed to guarantee global consistency of the data item;
- 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 database server and the mobile hosts. Compacts are obtained from the database via *requests* by the MH when a real or anticipated data demand is created. If 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 active 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;
- abort(), to abandon the changes made to the compact data by a given transaction; and,
- checkpoint(), to store the current state of the compact for purposes of recovery.

The implementation of a common interface simplifies the design of the compact agent and guarantees the minimum acceptable functionality of a specific compact instance.

The compacts are periodically *updated* as the result of processing transactions on the MH. When the needs of the mobile host or the database server change, compacts may be *renegotiated* to redistribute resources and, when the MH no longer needs the resources, compacts are *returned* to the database server and deleted from the local compact registry and the compact store (if necessary).

4.2. Using compacts to implement dynamic replication

The flexibility offered by compacts allows PROMOTION to support a number of dynamic replication schemes (as well as caching) with a variety of consistency constraints. Compacts can be used to represent both simple schemes such as check-in/

check-out items and leases [20], as well as other more complex correctness criteria [21]. Let us illustrate this by showing how leases and check-in/check-out items can be realised using compacts and how these compacts can be used to support transactions with different visibility properties, such as the last two transactions in the example in Section 3.

Lease semantics provide that a shared data item is given to all requesters (leaseholders) with an expiration time. Leaseholders with read access are free to read the item (as long as the lease has not expired), but must obtain permission from all other leaseholders before modifying a leased item. Therefore, a compact designed to support leases (Fig. 4) includes, in addition to the methods for the common interface, two type-specific methods, read() and modify(). The read() method simply checks the expiration (specified in the compact as an obligation) and, if unexpired, satisfies the read request, returning the value of the compact data (Manifest 10292, B/L#A3439392...). When the modify() method is invoked and the lease has not already been converted to write access, the compact must communicate with the database server. The database server obtains permission from the other leaseholder compacts and communicates permission (or refusal) to the lease compact on the MH performing the modification. This might be an opportune time to renegotiate the compact deadline to prevent monopolisation of the resource by the MH requesting write access. A lease compact requires no consistency rules and a contains a single obligation, the expiration time (Deadline=48999.00). The compact state (Last Update) is maintained by the type-specific and interface methods.

Our trucker can use such a compact to obtain exclusive access to the manifest needed to officially record his/her acceptance of a load (i.e. the MANIFEST transaction in our example). Perhaps a small set of manifests would be reserved to each truck and kept in a single compact. The deadline would then be arranged so that, if the unit disappeared, the manifests would eventually be assigned to another truck. No use of a manifest from an expired compact would be allowed to satisfy the transaction, as that manifest may have already been re-assigned to another load. An indefinite deadline (i.e. infinite) would reserve the manifest to the requesting truck forever, requiring some type of administrative intervention to return the manifest to the database server.

Compacts may also be used to manage check-in/check-out items. A check-in/check-out item is little more than a lease item with exclusive read and write access to the data item. When a data item is first requested for check-out, no associated active compact object (i.e. a compact with an unexpired deadline) will exist in the compact store at the database server.

A check-in/check-out compact could be used for the final BILLING transaction in our example. Since the MH holds complete read and write access to the data item, the MH becomes ultimately responsible for the correctness of the data in the compact. When local transactions update the manifest and signal an intent to commit, the results can be made immediately visible to subsequent transactions, *even though the results may not have been recorded at the database server.* This technique allows multiple transactions to access and update the manifest data even though the MH is disconnected. The central database can be updated when the MH reconnects. If the compact has expired, there are two possibilities:

- If no other transaction has modified the manifest, the update may be stored despite the expired deadline. This is an example of optimistic concurrency control.
- If the copy of the manifest in the centralised database *has* been modified, the update will be aborted and the truck driver will be requested to repeat the entry of the proof of delivery, effectively 'redoing' the transaction.

The notify() method of this particular compact, triggered by the reconnection of the MH to the fixed network, could be written to handle this reconciliation of the manifest compact with the database server.

Because compacts include the code for access to shared objects, compacts provide a high degree of adaptability which may be exploited to respond dynamically to changes in the mobile environment. For example, compacts may react to an increased availability of hard disk by automatically increasing the amount of data cached through prefetching and hoarding, or may adapt to an overloaded communication subsystem by reducing the frequency of transmitted updates. The use of compacts, therefore, creates a flexible and adaptable framework to support mobile transaction processing.

In each of these cases, we have processed as much of the transaction on the mobile host as possible, without resorting to communication with the database server. The database server is contacted only when convenient or when absolutely

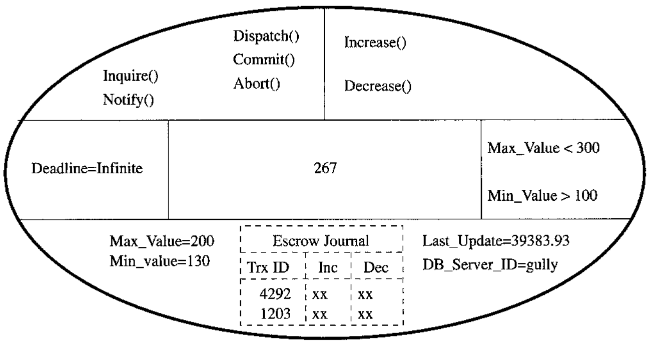

Fig. 5. Escrows as compacts.

required by the semantics of the transaction. The compacts associated with each data type (and, by association, each transaction type) are designed to meet the requirements of a particular data object, simplifying the job of local transaction management.

In the next section, we elaborate further on the usage of compacts and illustrate one additional advantage offered by compacts, namely, their ability to support concurrency control methods which exploit object structure and/or semantics [22–24].

4.3. Using compacts to support semantics-based concurrency control

In order to better support disconnected operations and strict data consistency, such as that provided by serialisability, the mobile transaction processing system can exploit object semantic information to provide finer granularity of caching and concurrency control and to allow for asynchronous manipulation of the cached objects and unilateral commitment of transactions on the mobile host. As an example, escrow items and fragmentable items have characteristics that make them especially suitable for mobile transactions.

The basic idea behind both *escrow* [25] and *fragmentable* items [18] is to split large or complex objects into smaller fragments of the same type by exploiting the object organisation. With the appropriate split, a mobile host can cache a *data partition* (consisting of one or more fragments) of just the right size, minimising the storage requirements on a mobile host. The second idea is to make these fragments the unit of reconciliation of updates, that is, the unit of consistency. To allow more flexibility (as well as to deal with situations in which fragmentation under strict consistency requirements is not possible) applications can explicitly define the consistency constraints to be enforced.

A 'master copy' of the escrowable or fragmentable data resides on a database server. Mobile hosts specify the granularity of the data to be cached when placing a compact request by specifying the required size of the data partition. The data partition is logically removed from the 'master copy', packaged into a compact and transmitted to the MH. The data contained in the compact is only accessible by the transactions on the mobile host. However, the remaining part of the 'master copy' is not affected and it is available to other MHs as subsequent compacts. Object fragments can be *logical* (i.e. escrow) or *physical* (i.e. fragments) divisions of the data object. During the update, physical fragments need to be physically re-assembled into a single object while logical fragments are combined with some logical or arithmetic operation.

In order to support unilateral commitment of transaction executing on a mobile host, we must retain the effects of transaction operations on each fragment when the fragments are merged. The consistency conditions embodied in the compact methods specify constraints on the fragment which need to be satisfied to maintain the consistency of the entire object. These conditions might include allowable operations and constraints on their input values and conditions on the state of the object. Some operations on fragmentable items may be disallowed or restricted to guarantee that the fragments may be properly merged.

Figure 5 illustrates a compact designed to support escrowable data (i.e. a portion of an aggregate[1] item which has been allocated to satisfy requests

[1]An aggregate is a data object which represents a quantity of identical and interchangeable items, such as bushels of wheat or dollars in an advance account.

from transactions executing on the mobile host). In this particular case, taken from the DISPATCH transaction in our trucking example, the quantity indicates the amount of fertiliser available to the truck. In addition to the methods which provide a common interface, this compact contains two type-specific methods: increase and decrease. The data (i.e. 267) may only be accessed by these two methods. Each call to increase is validated against the maximum value (i.e. 300) specified in the consistency rules, and the compact state (i.e. Escrow Journal), which reflects validated requests by concurrent transactions. The call succeeds only if the consistency rules will not be violated. Similarly, each call to decrease (caused, for example, by the truck's acceptance of a load) is validated against the minimum value (i.e. 100) and the compact state. Adherence to the consistency rules will insure that the global consistency of the aggregate values will be guaranteed even if transactions are allowed to commit unilaterally on the MH. Because unilateral commitment is possible, transactions on other MHs may proceed even though the only obligation, a deadline for return of the escrow quantity, has been set to unlimited (i.e. Infinite). In most cases a definitive deadline would be set to insure that any quantity of fertiliser not moved by this truck would be made available to competing units. At some point the expiration of the compact could be set to coincide with the expiration of the contract with the fertiliser shipper, if no loads will be available beyond that time. Renegotiation of these escrow contracts would be used to allow aggressive drivers to control the bulk of the material being hauled.

4.4. System architecture

As mentioned in the introduction, one of our motivations has been to support existing database applications by facilitating data access by transactions on mobile hosts. In such an environment, it might not be easy, or even possible, to incorporate logic to manage compacts into the legacy database server[2]. If a database server lacks compact management capabilities, a *compact manager* can provide that functionality. The compact manager acts as a front-end to a database server, shielding the database server from the idiosyncrasies of the mobile environment. The compact manager may execute on an independent host, or it may execute on the same host as the database server (as shown in Fig. 2).

If a compact manager is added to a legacy database, our system utilises an open nested transaction model as the basis for concurrency control and recovery for mobile transactions processed against the database server. To the database server, the compact manager appears to be an ordinary database client, executing large, long-lived transactions. These transactions become the root transactions of our nested transaction model. Resources needed to create compacts are obtained by these transactions through normal database operations (reads and writes). Mobile transactions (transactions processed by each mobile host) appear as children in the open nested transactions. The transactions processed on the mobile host appear as siblings. Each sibling transaction may commit or abort independently as long as the consistency constraints expressed in the compacts that have shared access are not violated. It should be noted that a mobile transaction may invoke multiple compacts and a compact may support the execution of multiple transactions. The responsibility for the correct execution of mobile transactions is assumed by the MH and accomplished by utilising the methods encapsulated in the compacts. The root transactions are managed by the database server and committed by the compact manager.

On each MH, a *compact agent* is responsible for processing requests on behalf of transactions executing on the MH. These compact agents are more than just the interface between the compact manager and the transactions on the mobile hosts. The compact agent is much like the daemon responsible for cache management in the CODA file system [17]. 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. The compact agent is responsible for concurrency control, logging and recovery. Consequently, transaction requests, commits, aborts and journals are managed by the compact agent. Requests from local transactions are processed against the compacts and are granted or denied. When a transaction commits or aborts, the compacts and transaction journals are updated accordingly.

[2]A similar situation exists in multidatabase systems that attempt to integrate pre-existing database systems while retaining the *design autonomy* of the component database systems [6].

Recall that each compact includes a set of methods used for management which are common to all compacts. In addition, each compact may contain specialised methods which support the particular type of data or concurrency control specific to that particular compact. As a result, many of the functions associated with the compact agent are actually executed by the compacts themselves upon receipt of messages from the compact agent triggered by executing transactions or changes in the MH state.

Some system events will cause the compact's state at the compact manager to be updated to reflect the effects of all outstanding committed transactions. The specific triggers for updates may vary, but should include:

1. When renegotiation must be performed for additional resources and the update can be *piggybacked* onto the request.
2. As part of a handoff procedure when a transfer to a new cell is initiated.
3. When the number of commit requests reaches a predetermined threshold, usually determined by the memory capacity of the mobile host.
4. In a 'panic' situation arising from impending disconnection from:
   - weak or low battery power;
   - deliberate disconnection by the user;
   - or partial loss or weakness of signal detected by the communication subsystem.
5. A 'safe' interval before an approaching deadline (e.g. a trigger could be set for midway between the last attempt and the deadline).
6. When specifically requested by a critical application being processed on the mobile host.

Once the update is acknowledged by the compact manager, the compact agent updates journals and logs appropriately. In this manner, entire groups of committed transactions may be processed with a single update to the compact at each end, with significant savings in communication overhead.

Recall that each of the interactions between the compact agent and the compact manager are processed via the MSS. To improve system performance, the MSS can take an active role in the processing of compacts between the compact agent and the compact manager. The module which performs these functions on the MSS is called the *mobility manager* (MM). Once an update is sent to the MM, the MM functions on behalf of the compact agent to complete delivery of the update message. This *update by proxy* helps insure that the updates are received by the compact manager in a timely fashion. If the MH sends an update and immediately disconnects, the update can be recorded by the compact manager and the acknowledgment can be stored by the MM and properly processed later, once the MH reconnects. The MM maintains *mobility tables* in which each *mobile control block* (MCB) contains location and database access information which pertains to a single MH. When a MH moves between cells, the MM uses the information stored in the MCB to facilitate the handoff to a new MSS (and corresponding MM).

The addition of the compact manager, compact agent and mobility manager provides a functional infrastructure that will move some of the responsibility for processing and committing transactions down to the MSS and the MH, reducing the dependence on communications with the database server.

5. Conclusions

Recognising the need for a suitable mobile transaction processing system, in this paper we have presented PRO-MOTION, which provides support for transaction processing by disconnected mobile hosts without grossly impairing access by transactions executed by the stationary host and other MHs. PRO-MOTION is designed to facilitate the migration of existing applications and the development of new database applications in a mobile environment.

We have introduced compacts, PRO-MOTION's basic unit of caching, prefetching, and hoarding, which encapsulate access methods with database data to allow uniform management of transactions despite varying consistency constraints and correctness criteria. Compacts provide a mechanism for flexible, adaptive, and extensible support of traditional transactions, extended transaction models, and new schemes exploiting data structure or operation semantics to achieve efficiency and correctness. The management of compact deadlines allows for automatic recovery of resources held by MHs which exceed negotiated limits on disconnection. By associating data with the methods (i.e. code) that manipulate the data, PRO-MOTION provides the mechanism by which the MH will automatically receive code necessary to manipulate data in the compact. This eliminates the need to write a comprehensive local transaction manager which contains code for types which may never be used and allows for the automatic updating of access methods as the system evolves.

We are building a prototype PRO-MOTION system based upon the ideas presented in this paper to test the concept and experiment with various implementations. Our implementation is well underway using Java. A number of data types, including fragmentable stacks and leases, have been coded in the form of a portable compact library that will enhance sharing and reusability.

Acknowledgements

We would like to express our thanks to the anonymous reviewers for their thoughtful comments. This material is based upon work supported by the National Science Foundation under grant IRI-95020091 and partially funded by B-Right Trucking Company.

References

1. Imieliński T, Viswanathan S, Badrinath B Energy efficient indexing on air. In: Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, 1994
2. Acharya S, Alonso R, Franklin M, Zdonik S. Broadcast disks: data management for asymmetric communication environments. In: Proceedings. of the 1995 ACM SIGMOD International Conference on Management of Data, 1995 pp 199–210
3. Datta A, VanderMeer D, Kim J, Celik A, Kumar V. Adaptive broadcast protocols to support power conservant retrieval by mobile user. In: Proceedings of the 13th IEEE International Conference on Data Engineering, 1997 pp 124–133
4. Walborn G, Chrysanthis P K. PRO-MOTION: management of mobile transactions. In: Proceedings. of the Symposium on Applied Computing, 1997 pp 101–108.
5. Ioannidis J, Duchamp D, Maguire G Q. IP-Based protocols for mobile internetworking. In: Proceedings of ACM SIGCOMM Symposium on Communication, Architectures and Protocols, 1991 pp 235–245
6. Bernstein P A, Hadzilacos V, Goodman N. Concurrency Control and Recovery in Database Systems. Addison–Wesley, Reading, MA, 1987
7. Ramamritham K, Chrysanthis P K. Advances in Concurrency Control and Transaction Processing. IEEE Computer Society Press, Washington, DC, 1996
8. Barbara D. Certi cation reports: supporting transactions in wireless systems. In: Proceedings of the 17th International Conference on Distributed Computing Systems, 1997
9. Kisler J, Satyanarayanan M. Disconnected operation in the Coda file system. ACM Trans Computer Systems 1992 10(1): 3–25
10. Tait D C, Lei H, Chang H. Intelligent file hoarding for mobile computing. In: Proceedings of the Workshop on Mobile Computing, 1995. pp 119–125
11. Kuenning G, Popek G J, Reiher P. An analysis of trace data for predictive file caching in mobile computing. In: Proceedings of USENIX Summer 1994 Conference, 1994
12. Tait D C, Duchamp D. Service interface and replica management algorithm for mobile file system clients. In: Proceedings of the 1st International Conference on Parallel and Distributed Information Systems, 1991 pp 190–197
13. Jain R, Narayanan K. Network support for personal information services to PCS users. In: Proceedings of IEEE Conference Networks for Personal Communications, 1994
14. Yeo L H, Zaslavsky A. Submission of transactions from mobile workstations in a cooperative multidatabase processing environment. In: Proceedings of the 14th International Conference on Distributed Computing Systems, 1994
15. Dunham M, Helal A, Balakrishnan S. A mobile transaction model that captures both the data and movement behavior. ACM/Baltzer Journal on Special Topics in Mobile Networks (to appear).
16. Chrysanthis P K. Transaction processing in a mobile computing environment. In: Proceedings of IEEE Workshop on Advances in Parallel and Distributed Systems, 1993 pp 77–82
17. Pitoura E, and Bhargava B. Maintaining consistency of data in mobile distributed environments. In: Proceedings of 15th International Conference on Distributed Computing Systems, 1995 pp 404–414
18. Walborn G, Chrysanthis P K. Supporting semantics-based transaction processing in mobile database applications. In: Proceedings of the 11th Symposium of Reliable Distributed Systems, 1995 pp 31–40.
19. Krishnakumar N, Jain R. Protocols for maintaining inventory databases and user service profiles in mobile sales applications. In: Proceedings of the Mobidata Workshop, 1994
20. Gray C G, Cheriton D. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In: Proceedings of 12th ACM Symposium on Operating Systems Principles, 1989 pp 202–210.
21. Ramamritham K, Chrysanthis P K. A taxonomy of correctness criteria in database applications. VLDB J 1996; 4(1): 181–293
22. Badrinath B R, Ramamritham K. Semantics-based concurrency control: beyond commutativity. ACM Trans Database Systems. 1992; 17(1): 163–199.
23. Chrysanthis P K, Raghuram S, Ramamritham K. Extracting concurrency from objects: a methodology. In: Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data. 1991 pp 108–117
24. Agrawal D, El Abbadi A, Singh A K. Consistency and orderability: semantics-Based. ACM Trans Database Systems. 1993; 18(3): 460–486
25. O'Neil P. The escrow transactional method. ACM Trans Database Systems 1986; 11(4): 405–430
26. Breitbart Y, Garcia-Molina H, Silberschatz A. Overview of multidatabase transaction management. VLDB J 1992; 1(2): 181–293.
27. Jing J, Bukhres O, Elmagarmid A. Distributed lock management for mobile transactions. In: Proceedings of the 15th International Conference on Distributed Computing Systems. 1995 pp 118–125
28. Imieliński T, Badrinath B R. Mobile wireless computing: challenges in data management. Commun ACM 1994; 37(10): 18–28

Correspondence and offprint requests to: Gary D. Walborn, (gwalton@cs.pitt.edu) Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260, USA.