Supporting Semantics-Based Transaction Processing in Mobile Database Applications*

Gary D. Walborn and *Panos K. Chrysanthis*
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260

Abstract

Advances in computer and telecommunication technologies have made mobile computing a reality. However, greater mobility implies a more tenuous network connection and a higher rate of disconnection. In order to tolerate disconnections as well as to reduce the delays and cost of wireless communication, it is necessary to support autonomous mobile operations on data shared by stationary hosts. This would allow the part of a computation executing on a mobile host to continue executing while the mobile host is not connected to the network.

In this paper, we examine whether object semantics can be exploited to facilitate autonomous and disconnected operation in mobile database applications. We define the class of fragmentable *objects which may be split among a number of sites, operated upon independently at each site, and then recombined in a semantically consistent fashion. A number of objects with such characteristics are presented and an implementation of* fragmentable stacks *is shown and discussed.*

1 Introduction

The growth of wide area networking and the emergence of portable and mobile computer systems has changed the way we must look at concurrent access to shared data. Wide area and wireless networking suggests that there will be even more competition for shared data since it provides users with the ability to access information and services through wireless connections that can be retained even while the user is moving. Further, mobile users will have to share their data with others. In this new mobile database environment, the task of achieving the required performance and ensuring the consistency of shared data becomes more difficult than in traditional database systems because of the inherent limitations of the wireless communication channels and restrictions introduced by mobility and portability. In such an environment, increased autonomy of mobile users can mean increased performance, increased functionality, and simplified recovery in the presence of failures.

*This material is based upon work supported by the National Science Foundation under the grant IRI-9210588.

Author's Email: {gwalborn,panos}@cs.pitt.edu

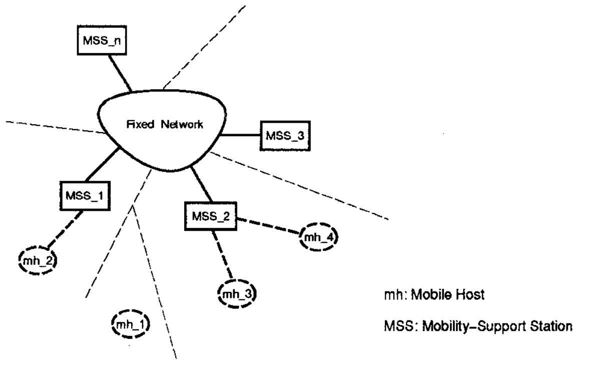

Figure 1: Mobile Network

In a mobile computing environment, the network is made up of *stationary* and *mobile* hosts [16, 15]. Unlike stationary hosts, mobile hosts change location and network connections while computations are being processed. While in motion, mobile hosts retain their network connection through the support of specialized stationary hosts with wireless telecommunication ability, called *mobility-support stations* (Figure 1). Each mobility-support station is responsible for all of the mobile hosts within a given geographical or logical area, known as a *cell.* At any given instant, a mobile host may directly communicate only with the mobility-support station responsible for the area in which the mobile host moves. Further, a mobile host has to compete with the other mobile hosts in a cell for the wireless connections which are both of low bandwidth and expensive in terms of power consumption and monetary cost.

Within this mobile computing environment, shared data are expected to be stored and controlled by a number of database servers executing on stationary hosts. Mobile hosts are assumed to have limited storage capacity for cached data. Mobile hosts cache and store shared data on a hard disk or a flash memory which survive power failures. To ensure the consistency of the shared data in the presence of concurrency and failures, users on both mobile and stationary hosts update and retrieve the data using transactions.

It is possible, even probable, that a mobile host will become *disconnected* from the network due to (accidentally or intentionally) broken communication connections. For example, in Figure 1, MH_1 becomes accidentally disconnected upon entering a region out of the reach of any mobility-support station. On the other hand, planned disconnection occurs when communication on the mobile host is turned off to save energy or reduce communication expense. Such disconnections, however, do not imply failure of the disconnected mobile host. On the contrary, the part of a computation executing on a mobile host may continue executing while the mobile host is moving and not connected to the network.

For temporary disconnection to be tolerated with no disruption of transaction processing, suspending (blocking) transactions executed on either stationary or mobile hosts needs to be minimized. Proper support for *mobile transactions*, i.e., transactions invoked on a mobile host, must provide for local autonomy to allow transactions to be processed and committed on the mobile host despite temporary disconnection. At the time of reconnection, the effects of mobile transactions committed during a disconnection would be incorporated into the database while guaranteeing data and transaction correctness. Therefore, instead of considering and handling disconnections as failures that would require transactions on disconnected mobile hosts to be aborted, in this paper we handle disconnections as a *concurrency* and *cache coherency* problem by considering mobile transactions as long-executing transactions operating on dynamically replicated objects on the mobile host. This is appropriate, since mobile transactions are expected to be long-lived because of long communication delays over wireless channels whether or not disconnection occurs. In fact, temporary disconnections cannot be distinguished from long network delays.

The most permissive concurrency control schemes are those which exploit the semantics of the objects and the operations defined on them, the structure of the database, the structural and behavioral properties of the activities in the database, and the correctness requirements of the applications [8, 26]. We refer to these techniques as *semantics-based transaction processing* techniques. Many of these techniques, which have been proposed to support long lived transactions in distributed database and multidatabase systems, will allow for greater autonomy and simplified recovery after failures, but techniques which require caching large portions of the database or which maintain multiple copies of many data items may have excessive storage and communication costs for a mobile host. In this paper, we introduce two new semantic concepts, namely *fragmentability* and *reorderability* that can be used to facilitate semantics-based transaction processing in mobile database applications.

Fragmentability defines a new class of objects which better suit mobile and disconnected operation by allowing a portion of these (potentially large) objects to be cached and operated upon independently. Specifically, fragmentable objects may be split into disjoint fragments which can be distributed among a number of sites, operated upon independently at each site, and then recombined in a semantically consistent fashion. Stacks, sets, and queues are examples of objects which fall into this category. Reorderable objects are fragmentable objects that exhibit more flexibility in merging their fragments by allowing the fragments of an object to be rearranged.

The rest of this paper is structured as follows: In Section 2 we examine the various semantics-based transaction processing techniques in the context of mobile database applications, emphasizing those which exploit object and operation semantics to support autonomous and disconnected transaction processing. In Section 3, we introduce the concepts of *reorderability* and of *fragmentable* data items (whose elements constitute the unit of caching and reconciliation of updates) and examine several such items. In Section 4, an implementation of *fragmentable stacks* is shown and discussed. Section 5 discusses the future direction of this work and suggests areas for extended investigation and implementation.

2 Exploiting Object Semantics

Almost all types of transaction processing systems employ some semantic knowledge to provide increased availability of data, to achieve a high degree of concurrency, and to simplify recovery in the presence of failures. Broadly speaking, the concurrency semantics of an object depend on the following object characteristics [10]:

1. *semantics of the operations* – which is related to the effects of an operation on the state of an object,
2. *operation input/output values* – which refers to both the direction of information flow to and from an object and the interpretation of the input/output values,
3. *organization of the object* – which refers to the abstract organization of an object (as opposed to its physical implementation), and
4. *object usage* – which refers to how an object is used and what is done with information extracted from the object.

The first three object characteristics have primarily been used to define various forms of *commutativity* which determine semantically whether two operations can be allowed to execute concurrently without compromising *serializability*, the traditional database correctness criterion [9]. The last characteristic, object usage, has been exploited to define application-specific correctness criteria which transcend commutativity and serializability to allow even more operations to execute concurrently and asynchronously.

Although these semantics-based transaction processing techniques have the potential to improve performance, it is not clear that these same techniques can be used to support mobile transaction processing. Therefore, it is necessary to evaluate these existing semantics-based techniques by the measure of how they apply to the mobile transaction processing environment and, in particular, how well they support disconnected operation. Below we examine how these techniques are commonly applied and briefly discuss their advantages and disadvantages in a mobile environment. To simplify the discussion we classify them into two groups: *application independent semantics-based* techniques and *application dependent semantics-based* techniques.

2.1 Application Independent Semantics

Because the semantics of most operations are, by their very definition, well understood, this object characteristic is the first to be used and the one most often exploited. The most commonly used semantic property of operations is *commutativity.*

If two operations commute, then their effects on the state of an object and their return values are the same irrespective of their execution order. Operations which commute can be arbitrarily ordered in a concurrent execution to effect serializability. Commutativity can be used to enhance recovery as well as concurrency. If a protocol allows only commuting operations to execute concurrently, it prevents cascading aborts, localizing recovery to transaction boundaries.

Some operations commute in all object states (e.g., two increments). We can, however, identify operations that commute only in specific object states. This property of operations is called *state-based commutativity.* For example, two operations which push identical values onto a stack may execute in any order and the resultant stack will be the same. That is, push operations with identical input values commute, while push operations with differing input values, do not. Output values can also be used to determine if two operations are state commuting. For example, two push operations that return "stack-overflow" when failing to push a value, commute.

The output values in conjunction with the inputs of the operations have been used for new definitions of conflicting operations (i.e., operations that cannot execute concurrently) which are weaker than commutativity and yet ensure serializability. Although these definitions, such as *serial dependency* [14] and *recoverability* [4], permit a higher degree of concurrency than commutativity and can be used within a traditional replicated database environment, they are associated with more complex recovery and involve more complex transaction management than commutativity.

Clearly, commutativity can be used to enhance the concurrent access to, and simplify recovery of, shared data within a mobile host. For objects stored in the database server, if all operations of an object commute with each other in all states, then this object could be cached and manipulated at a mobile host asynchronously, without any coordination with the database server. The mobile host is only required to periodically propagate to the server the updates of the mobile transactions committed locally. Such objects can be exploited to facilitate the autonomous operation of mobile hosts during disconnections.

However, in reality very few operations of an object commute and cached objects require communication to coordinate the execution of conflicting operations in order to guarantee the consistency of the cached copies. In traditional caching schemes, every time a cached copy is updated, it is propagated to the database server which, in turn, invalidates or updates all other copies. Such communication is prohibitively expensive for mobile hosts which communicate over low bandwidth wireless channels and have limited battery life. Further, because wireless data communication employs broadcast technologies, frequent communication by each mobile host could result in wireless network congestion and decreased throughput. A number of cache coherence schemes have been studied (in the context of mobility and disconnections) that exploit the broadcast characteristics of wireless communication to reduce the communication cost without exploiting any object semantics [7].

Commutativity of operations can be exploited in caching methods that employ a concurrency control protocol to ensure cache coherence. For example, in an optimistic concurrency control based scheme, cached objects on mobile hosts can be updated without any coordination (an attractive feature for disconnected operation) but the updates need to be propagated and validated at the database server in order for the invoking transaction to be committed [17]. Unless conflicts between concurrent updates are rare, this scheme will lead to more abortions of mobile transactions (which are expected to be long-lived due to disconnections and long network delays) when compared to transactions executing on stationary hosts. In this case, both commutativity and serial dependency can be exploited to validate more mobile transactions. In a pessimistic scheme in which cached objects can be locked *exclusively,* mobile transactions can be committed locally. However, pessimistic schemes might lead to unnecessary transaction blocking since a mobile host cannot release any cached objects while it is disconnected.

Existing caching methods attempt to cache entire objects or, in some cases, complete files. Transmission of these potentially large objects over low bandwidth communication channels can result in severe wireless network congestion and high communication cost. Furthermore, the limited cache size of a mobile host means that only a small number of (large) objects can be cached on a mobile host at any given time. Thus, it is important that the granularity of caching is as fine as possible so that a working subset of the database, sufficient to sustain a practical level of transaction processing during disconnections, can be cached. Here the semantics of the object organization can be helpful in fragmenting large objects into smaller components that can be cached independently while maintaining consistency. The escrow transactional method [22, 21, 29, 19] is a perfect application of this principle.

The escrow transactional method is designed specifically to improve concurrent access to *aggregate items* by exploiting object structure, state-based commutativity, and integrity constraints. Specifically, the escrow method exploits the fact that aggregate items are numeric values which represent a quantity of interchangeable items, such as units of blood or dollars in an account. Since all of the items are interchangeable, the quantity can be divided among a number of mobile hosts based on their data requirements [20, 31]. Data consistency is ensured by limiting the operations on the cached quantity to increments and decrements that commute provided that certain boundary conditions derived from integrity constraints (e.g., a negative physical inventory) are not violated. Thus, by assuming that the worst case of pending operations will not violate boundary conditions, the database server can incorporate the effects of committed mobile transactions in an arbitrary order that preserves serializability.

Whereas escrow methods ensure serializability of global executions, the demarcation protocol [5], which was also designed to support autonomous updates on aggregate items, ensures only that local executions are serializable. The demarcation protocol does not rely on the commutativity of operations but it does exploit the object organization and integrity constraints. Specifically, it preserves data consistency by enforcing explicit consistency constraints on the cached quantities which establish safety limits during updates. As we will see in the next section, aggregate items are good examples of fragmentable objects.

2.2 Application Dependent Semantics

Methods which exploit object usage enhance performance and functionality of transactions by relaxing the notion of a serializability in favor of weaker, but equally acceptable, application-specific criteria. As opposed to serializability, these methods typically view *data consistency* and *transaction correctness* independently [26].

Data consistency captures correctness from the perspective of objects in the database. Data consistency requirements range from *strict consistency* (as defined by serializability) to *eventual consistency* [1, 28]. Eventual consistency denotes a temporal or spatial divergence from strict consistency the extend of which can be expressed in terms of *degrees of inconsistency.* For example, a degree may require consistency "at a specific real-time," "within some time" or "after a certain amount of change to some data," or enforcing consistency "after a certain value of the data is reached," etc. Divergence control protocols such as those for quasi-copies [1] and for epsilon-serializability [25] allow for the applications to specify their currency requirements and the inconsistency with which they can cope. These protocols can also facilitate the tuning of cache management on mobile hosts according to the available bandwidth and monetary cost of wireless communication [6].

Transaction correctness requirements capture correctness from the perspective of the structure and behavior of transactions [12, 13]. That is, these methods allow applications to specify (1) the *degree of isolation* of different transactions in terms of acceptable transaction interleavings, delegation of operations and early or partial commitment of the operations of transactions, and (2) the *degree of transaction autonomy* in terms of transaction interdependencies such as commit and abort dependencies.

Clearly, these relaxed correctness criteria require more from the transaction developers than serializability which works under the simple assumption that correctly written individual transactions preserve database consistency. In particular, a transaction may have to be aware of the functionality of other transactions and the potential interactions among transactions. This makes transaction development, as well as management, more difficult. Furthermore, transaction and database recovery also becomes more complex. Traditional recovery techniques are typically not sufficient to restore the database to a consistent state after a failure and additional compensating steps are required to semantically undo the effects of previously executed and committed operations. However, these relaxed correctness criteria are useful for mobile transaction processing because they can significantly increase the autonomy of mobile hosts.

Application-specific criteria have been mainly proposed in database environments with characteristics similar to those of mobile database applications. As discussed above, ensuring serializability and maintaining strict consistency among data stored on both mobile and stationary hosts is very constraining as well as expensive [2, 15, 30]. For this reason, current approaches that aim to support transactions which perform updates on cached objects at the mobile hosts advocate application-specific criteria. For example, an *open-nested transaction* model has been proposed in [11] for modeling mobile transactions as a set of subtransactions. The model allows for disconnected operation by supporting unilateral commitment of subtransactions and compensating transactions. Further, it can be customized based on the application by varying the degree of isolation and autonomy of the subtransactions of a mobile transaction.

In order to support mobile transactions that can accommodate spatial inconsistencies, dynamic object clustering has been proposed based on objects' degree of inconsistency and two types of read and write operations: *weak-read, weak-write, strict-read* and *strict-write* [23, 24]. Strict-read and strict-write operations have the same semantics as the read and write operations invoked by traditional ACID transactions [9]. A weak-read returns the value of a locally cached object written by a strict-write or a weak-write. A weak-write operation updates a locally cached object which might become permanent on cluster merging if the weak-write does not conflict with any strict-read or strict-write operation.

A common characteristic of the different divergence control methods, escrow methods, and the demarcation protocol, is that all deal with read/write objects or objects with numeric values. In the case of read/write objects and traditional caching schemes, the entire object is cached, which could, potentially, entail intolerable overhead in a mobile environment. Also, to make effective use of these methods, we need to apply them to as many data types as possible. In our approach to mobile transaction processing, we propose the utilization of object organization and application semantics to break objects into smaller pieces to be cached independently and manipulated asynchronously. The balance of this paper will outline our ideas and progress toward this end.

3 Fragmentable Objects

Consistency of data is a primary issue in all systems in which data are dispersed over multiple sites and in which both updates and retrievals are supported at all sites. As is evident from the previous section, the characteristics of the mobile environment, in particular the inherent restrictions of the wireless medium and of the mobile hosts (Figure 2), make semantics-based transaction processing the only viable way to ensure data consistency in mobile database applications. To better support disconnected operations and, whenever possible, strict data consistency, the approach presented here utilizes all types of 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.

The basic idea is to split large and complex objects into smaller fragments of the same type as the large object by exploiting the object organization. With the appropriate split, a mobile host can cache an *object partition* (consisting of one or more object fragments) of just the right size, minimizing 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. The objective is to support fragmentation of all objects in the database by exploiting commutativity of operation based on consistency constraints and object usage. 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. In the rest of this paper we will primarily focus on the fragmentation of objects which ensure strict consistency.

A "master copy" of each object resides on a stationary database server (or, in the case of a distributed system, at a number of database servers). Mobile hosts specify the granularity of an object to be cached and usage constraints by using the split operation. A merge operation is provided to allow transactions to release and incorporate cached fragments back into the master copy. Object fragments can be logical or physical divisions of the data object. Physical fragments need to be physically re-assembled into a single object while logical fragments are combined with some logical or arithmetic operation. Split and merge operations are type specific and as such the algorithms for splitting and merging object fragments could be encapsulated in the objects themselves. We refer to objects extended with these two operations (i.e., split and merge) as *fragmentable* objects.

When a mobile host requires access to an object, the mobile host sends a cache request to the database server by invoking the split operation with two parameters: *selection criteria*, and *consistency conditions*. The selection criteria specify the object to be cached and the required size of the object partition. When the object partition is cached on a mobile host, it is

Wireless Medium	**Mobile Hosts**
low bandwidth	small size
high bandwidth variability	small screen
energy demanding	limited batery life
monetarily expensive	limited storage -- both volatile and non-volatile
physical broadcasting in a cell	frequent disconnections -- unpredictable and predictable
	susceptible to failures

Figure 2: Wireless Network Characteristics

logically removed from the master copy of the object and is only accessible by the transactions on the mobile host. However, the remaining part of the master copy is not affected and it is accessible at the server.

The consistency conditions specify constraints on the fragment which need to be satisfied to maintain the consistency of the entire object. These conditions might include (a) allowable operations and constraints on their input values and (b) 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. For examples, operations on fragmented aggregate items are restricted to increment and decrement. Sometimes the constraints associated with an object fragment will be directly related to a constraint that was associated with the entire object, e.g., negative physical inventory. In other cases, the constraint may be necessary to assure that the fragments in the partition can be merged while preserving the changes brought about by transactions committed against the fragments. For example, requiring that the values to be added in a set must fall within a specific range. Finally, the constraints may be related to the degree of consistency that is required by an application. An example of a condition on the state of an object is the ability to test for an "empty stack" (which would necessitate including the fragment at the "bottom" of the stack).

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. Often the order of recombination will be dictated by the structure of the original object or the operations performed on each fragment. There are objects, however, whose fragments may be merged in a number of ways without destroying the data consistency of the object. In particular, there are objects in which the ordering of the fragments is an artifact of the sequence of operations performed against the object. When the fragments of an object can be rearranged to reflect an alternative sequence of operations on the object, we say that the object is *reorderable.*

3.1 Examples of Fragmentable Objects

Aggregate Items: As mentioned previously, aggregate items are an excellent example of fragmentable items. Aggregate items are logically fragmented. For this reason, the storage required for each fragment is identical. The selection associated with the split command is the entire object and the constraint conditions associated with each fragment state the minimum and maximum value that the aggregate fragment may assume. Aggregate items are merged logically by adding or subtracting fragment values.

Operations on such aggregate items usually include (but are not limited to) increasing, decreasing, or querying the current value. Because querying does not commute with either increments or decrements and the exact value of the entire object cannot be determined at the mobile host, only the first two operations (increase and decrease) are allowed to process against a fragment. However, these operations commute provided that the constraints on the minimum and maximum value of the object are not violated. By assuring that the worst case of a number of pending operations will not violate boundary conditions, serializability is ensured while enhancing performance and allowing for disconnected operation.

Sets: Sets are collections of independent items (members) in which the relative ordering of members is unimportant. Sets can be split into a number of subsets and these subsets may be combined in an arbitrary order to reconstitute the original set. Subsets represent physical fragments of sets and the storage required is directly related to the number of elements in the fragment.

When strict consistency is required, each mobile host needs to specify a range of elements as part of the selection criteria. All set fragments must be disjoint. When strict consistency is required, the constraint conditions will specify that testing for membership and insertion of items may only occur for elements within the range specified in the constraint conditions. A group of set fragments may be recombined into a single set by performing a *union* of the fragments. It should be noted that a tables is a type of set and fragmentation of tables with strict consistency is equivalent to horizontal fragmentation in a relational database system.

If weak consistency is sufficient, the set fragments may overlap and insertion of members may result in duplication across different fragments. These differences can be resolved by dropping duplicates from the set when the fragments are merged. If testing of membership is prohibited, the insertion of a duplicate member does not violate consistency constraints. If testing for membership must be allowed, consistency must be relaxed to permit such insertions.

Stacks: Stacks are lists of data that are accessed in a LIFO (last-in, first-out) fashion by means of push and pop operations. Push places a data item on the top of the stack. A pop removes the next available item from the top of the stack and returns the data item to the caller. The nature of these two operations implies that changes are made only to one end of the stack. All data items in the stack except the top cannot be accessed and are, essentially, "hidden" from all transactions.

When stacks are accessed by database transactions, each transaction may independently push and pop data items. In any serial execution all of the items pushed by a particular transaction will be adjacent in a stack. Strict consistency is maintained as long as this constraint is satisfied. Weak consistency constraints would allow for some interleaving to occur among objects from cooperating transactions.

A stack fragment is the basic unit of consistency and is defined to be any portion of the stack which contains interdependent data. All data items items pushed by a single transaction are interdependent. If a data item is popped by a transaction, any interdependent items on the top of the stack become interdependent with the data items of the transaction which performed the pop. Interdependent data items must be contained in a single fragment.

Stacks are physically fragmented objects. Each requesting mobile host caches one or more stack fragments. When the stack fragments are returned to the master copy, the stack fragments are physically reinserted into the master stack. Stacks are also reorderable, fragments may be rearranged to provide larger contiguous segments to satisfy a cache request. As an example, consider a stack S with initial contents S_I, and four transactions, T_1, T_2, T_3, and T_4. Assume that T_1 and T_2 each push three items on S and commit. The stack now looks like this:

$$\boxed{C_{T_2}\ B_{T_2}\ A_{T_2}}\boxed{C_{T_1}\ B_{T_1}\ A_{T_1}}\boxed{S_I}\|$$

Which is equivalent to the serial execution $T_1 \rightarrow T_2$. Further, assume that the stack is fragmented into three parts,

- $\boxed{C_{T_2}\ B_{T_2}\ A_{T_2}}$
- $\boxed{C_{T_1}\ B_{T_1}\ A_{T_1}}$
- $\boxed{S_I}\|$

and the first two components are given to mobile hosts, MH_1 and MH_2, respectively. If T_3 on MH_1 performs a two pops and commits, and T_4 on MH_2 performs two pops, a push, and commits, we are left with the following fragments:

- $\boxed{A_{T_2}}$ (on MH_1)
- $\boxed{A_{T_4}\ A_{T_1}}$ (on MH_2)
- $\boxed{S_I}\|$

Now, we can merge these fragments into a single stack that corresponds to a serializable history. A simple concatenation yields:

$$\boxed{A_{T_2}\ A_{T_4}\ A_{T_1}\ S_I}\|$$

which is the same as a serial execution $T_1 \rightarrow T_4 \rightarrow T_2 \rightarrow T_3$ against the initial stack S_I. To each of the transactions, T_3 and T_4, this is a perfectly acceptable sequence of events (and, of course, T_1 and T_2 committed before the fragmentation had even occurred). An equally correct history can be formed by reordering the two fragments in this fashion:

$$\boxed{A_{T_4}\ A_{T_1}\ A_{T_2}\ S_I\ \|}$$

which corresponds to the serial execution $T_2 \rightarrow T_3 \rightarrow T_1 \rightarrow T_4$ against the initial stack S_I. In each of these equally acceptable histories, T_1 *must* precede T_4 and T_2 *must* precede T_3 because T_4 reads from T_1 and T_3 reads from T_2.

A more detailed description of fragmentable stack implementation is included in Section 4.

Queues: Queues are lists of data that are accessed in a FIFO (first-in, first-out) fashion. Enqueue and dequeue are the two basic queue operations. Enqueue places a data item in the rear of a queue. Dequeue removes the next available item from the front of the queue and returns the data item to the caller. The nature of these two operations implies that changes are made only to the two ends of the queue. All data items in the queue except the first and last are not accessible.

Queues are also physically fragmented objects. As with stacks, a queue fragment is any portion of the queue which contains interdependent data. All data items items enqueued by a single transaction are interdependent. Interdependent data items must be contained in a single fragment. Queue fragments in which the original items have been completely consumed can be merged back in a fashion similar to stacks while maintaining strict consistency. However, the behavior of a fragmented queue more closely approximates that of "weak queues" as proposed by [27] if items of the original fragment are merged back into the master queue.

3.2 Formal Definitions

Below we will formally define *fragmentable objects* and *reorderable objects*. An object *ob* is described by a pair (O, C) where O is the *state* of the object and C is a (possibly empty) set of *consistency conditions* on the object. Any *legal* history of *ob* satisfies the constraints specified in C. A history $H_{(ob)}$ of an object represents the concurrent execution of a set of transactions indicating the (partial) order of operation invocations on the object. $H_{(ob)} = p_1 \circ p_2 \circ \ldots \circ p_n$, indicates both the order of execution of the operations, (p_i precedes p_{i+1}), as well as the functional composition of operations. Thus, a state O of an object produced by a sequence of operations equals the state produced by applying the history $H_{(ob)}$ corresponding to the sequence of operations on the object's initial state. Further, $H_{(ob)}$ is consistent with the operation invocation of individual transactions.

Definition 3.1: An object is $ob = (O, C)$ is *fragmentable* iff it can be split into fragments $(O_1, C_1), (O_2, C_2) \ldots (O_n, C_n)$ each of which supports the same operations as *ob* (i.e., fragments have the same type as *ob*) such that:

(1) Transactions can be processed asynchronously against individual fragments (O_i, C_i), each transaction against a single fragment of an object, producing a history h_i that satisfies the constraints C_i and results in new fragment state O'_i, and

(2) The resultant fragments (O'_1, C_1), (O'_2, C_2) $\ldots(O'_n, C_n)$ can be merged back into the original object $ob = (O', C)$ where O' corresponds to a legal history that is the concatenation of $O \circ h_1 \circ h_2 \circ \ldots \circ h_n$.

That is, a fragment corresponds to a subhistory of the original object that can be expanded in a consistent manner as long as its associated consistency conditions are not violated. A fragment can be obtained by projecting on the history of the object with respect to the selection conditions. The definition of a legal history depends on the semantics of the object and the consistency requirements of the application. In the case of strict consistency, a legal history is a serializable history.

Definition 3.2: An object $ob = (O, C)$ is *reorderable* iff *ob* is fragmentable and the resultant fragments $(O'_1, C_1), (O'_2, C_2) \ldots (O'_n, C_n)$ can be merged back into the original object $ob = (O', C)$ where O' corresponds to a legal history that is not necessarily a concatenation of $O \circ h_1 \circ h_2 \circ \ldots \circ h_n$. That is, the fragments can be merged back in more than one way where each corresponds to a different legal history.

Informally, reorderable objects allow the re-arranging of the fragments during merging. Reordering an object is equivalent to reordering the operations on the object (and the resultant history), thus increasing the number of allowable interleavings of operations. In this light, reorderability can be seen as an instance of serial dependency [14].

4 Implementation of Fragmentable Stacks

Although splitting an object into suitable fragments and merging them back in a consistent manner is type dependent, in this section we will show a fragmentable stack implementation in order to indicate the overhead and the complexity required to manage and cache fragmentable objects. This stack implementation maintains strict consistency as defined by serializability on an unbounded stack.

Maintaining Fragment Boundaries

To insure serializability, some constraints must be imposed on suitable fragmentation of the stack. In particular, the stack may only be fragmented between elements pushed by unrelated transactions. The basic strategy for fragmenting a single stack (or each of a number of independent stacks) is:

For each fragment, F_k, of stack, S, if $\exists$ data item D_i in F_k written by committed transaction T_n, then $\forall$ data items D_x, such that D_x was written by transaction T_n, D_x must belong to fragment F_k. Furthermore, if transaction T_n reads data from transaction T_m, all remaining data items D_y, such that D_y was written by transaction T_m

must belong to the fragment. Simply put, interdependent items must be contained in a single stack fragment.

If data dependencies exist between a number of queues and/or stacks, this rule may be expanded to include each interdependent object. Interdependent objects may be collapsed into a single large object using criteria similar to those for forming atomic sets in predicate-wise serializability testing. In general, a fragment must contain all data which is interdependent.

Special markers are used to signal the division between fragments. Each marker represents the beginning of stack fragment. To guarantee that the fragmenting strategy stated for interdependent data in stacks is observed, markers are inserted at appropriate points and removed using two simple rules:

a. If a transaction, in the course of reading an entry from the stack, encounters a marker, it reads and discards the marker.

b. If a transaction has read a marker from the stack, or if a transaction has pushed an item onto the stack, the transaction pushes a marker on the stack following its committed data.

These rules are observed at the database server and at each mobile host which has a portion of the fragmentable stack. Since a single marker is required for each stack fragment and the number of fragments cached by a mobile host should be small, the storage overhead for the mobile host is minimal. The maximum required space is consumed if each fragment contains a single data item and the overhead for this case is a constant factor of the number of data items in the stack.

Caching of Fragments

For simplicity, we will assume that a master copy of the stack will be maintained by a single database server. An allocation request from the mobile host specifies a stack, the number of data items it desires to cache, and a flag. The flag is used to indicate whether the mobile needs to test for an "empty stack" condition. Only the host holding the "bottom" stack fragment can detect an "empty stack". If this fragment is requested and has already been allocated to another host, the allocation fails.

To allow operation while disconnected, the mobile will request more stack items than needed by any one transaction. The database server will attempt to split the stack to accommodate the mobile host. A *stack partition* is a group of one or more stack fragments from the master stack. The database server should return a stack partition containing enough data items to fulfill the mobile host's request.

In order to keep track the current disposition of each stack fragment, the master stack is augmented with data tags. Fragments cached at a mobile host are tagged as belonging to the mobile host and logically removed from the master stack. In contrast to fragmentation markers, these tags are not part of a fragment per se and hence are not cached at a mobile host.

Push and Pop on Fragments

On a mobile host, transactions may pop items from and push items on the stack partition as if the stack partition were the entire stack. Recall, however, that an empty stack partition is *not* equivalent to an empty stack. If a transaction attempts to pop from an empty stack partition, the condition is signaled. If the stack partition contains the "bottom" fragment, then the pop operation detects an empty stack condition. Otherwise, the transaction which attempted the pop may block while the stack partition is expanded (by caching more stack fragments) or it may be aborted, whichever is more appropriate.

Conventional concurrency control and recovery techniques may be used to provide controlled access to the stack partition and to ensure serializability [9]. Assuming two-phase locking, uncommitted data is not made available to mobile transactions other than the transaction that pushed them and items pushed by a single transaction are contiguous in the stack partition. When a transaction on the mobile host commits, its pushed items are followed by a marker.

As transactions are processed, their effects are logged on the mobile host where the transaction executes to facilitate recovery from failed or aborted transactions. Mobile host logs are checkpointed periodically on an appropriate mobility-support station [18, 3].

Merging of Fragments

When a mobile host reconnects or the stack partition is no longer needed at the mobile host, any stack fragments remaining in the stack partition must be reconciled with master stack on the database server. The stack partition along with the log for committed transactions are checkpointed on the appropriate mobility-support station, transferred to the database server and deleted from the mobile host. Once the transfer to the database server completes, the checkpoint may be deleted.

Items consumed by committed transactions on the mobile host are deleted from the master stack by removing the stack fragments tagged with the mobile host id. Since stacks are reorderable, if the remaining stack partition does not contain the "bottom element," it is placed in the location of one of the deleted stack fragments. Otherwise, it replaces the fragment with the "bottom element" in the master stack. Finally, the fragment tags are adjusted to reflect each fragment state.

Example

To recap and illustrate the need for markers, let us consider an example of a stack that is used by a set of mobile transactions to evaluate arithmetic expressions in a cooperative fashion.

Assume that each expression comes from a single transaction. Each expression may be partially evaluated by a transaction which pops stack entries until it finds an operator, performs the operation, pushes the result, and repeats. (The "=" operator prints the result and pushes nothing.) We will use "M" to denote a marker. Initially, the stack contains two expressions pushed and committed by transactions T_2 and T_1 in that order.

T1	*T2*
M 5 2 + 8 * 6 - =	M 14 7 / 50 * 4 / 5 - =

Furthermore, assume that a mobile hosts MH_1 requests a stack fragment of size 5 and a mobile host MH_2 a fragment of size 3. According to the stack *fragment boundary* rule, the only possible split of the stack is:

	T1
MH_1	M 5 2 + 8 * 6 - =

	T2
MH_2	M 14 7 / 50 * 4 / 5 - =

Now consider that a transaction, *T3*, reads the first three items from the stack (fragment) at MH_2 (i.e., "14" "7" "/"), evaluates the sub-expression, places the result on the stack and commits.

	T1
MH_1	M 5 2 + 8 * 6 - =

	T3	*T2*
MH_2	M 2	50 * 4 / 5 - =

Let us assume that when the two stack fragments are merged, they retain their original order.

T1	*T3*	*T2*
M 5 2 + 8 * 6 - =	M 2	50 * 4 / 5 - =

Note that the markers correctly indicate that proper fragmentation of this resultant stack ***must not*** split the results of *T3* from *T2*. If we ignore the source transactions of the data items and look only at the fragments delimited by the markers, we see that the stack consists of only two fragments, *F1* and *F2* :

F1	*F2*
M 5 2 + 8 * 6 - =	M 2 50 * 4 / 5 - =

Because no marker exists between the items from transactions *T3* and *T2*, any fragment containing items from one must contain items from both and the fragmentation criteria that guarantee serializability are satisfied. The dependency results because *T3* reads from *T2* and, therefore, *T3 cannot* possibly precede *T2*. The resultant stack is reorderable and fragmentable without further restriction. Use of the markers maintains correct fragment boundaries and prevents this incorrect partitioning or reordering in a time and space efficient manner.

5 Conclusions

In this paper we have examined different types of semantic information with respect to their applicability in the context of mobile transaction processing. This examination led us to the realization that semantic information that requires access to the entire object is not suitable for mobile transactions irrespective of their potential power to allow for higher degrees of concurrency. For this reason, we have focused on those semantic notions that allow for concurrent executions on independent fragments of an object that can be cached locally in a mobile host and toward this end we have introduced the notions of ***fragmentable*** and ***reorderable*** objects.

The importance of escrow methods for distributed (and, particularly, mobile) computing has long been recognized. In a very real way, we have made a first attempt to apply these escrow methods to non-aggregate items. By encompassing a large class of escrowable objects, we can support a greater variety of applications on mobile platforms that require strict data consistency. At the same time, we can provide equal support for applications that can tolerate different degrees of inconsistencies by allowing fragments to diverge in a controlled manner.

In the future, we intend to continue our investigation on fragmentable and reorderable objects and on methods to maintain data consistency in mobile database environments in general. Finally, we want to to apply our ideas in practice by developing protocols to fragment and merge objects and to test implementations of these fragmentable objects in a simulated mobile environment in search of more efficient mechanisms.

Acknowledgements: We would like to thank S. Weissman and R. Conticello for their helpful suggestions.

References

[1] Alonso R., D. Barbara, and H. Garcia-Molina. Data Caching Issues in an Information Retrieval Systems. *ACM Transactions on Database Systems*, 15(3):359–384, Sept. 1990.

[2] Alonso R., and H. Korth. Database Issues in Nomadic Computing. *Proc. of ACM SIGMOD Conf.*, pp. 388-392, May 1993.

[3] Archarya A. and B. Badrinath. Checkpointing distributed applications on mobile computers. *Proc. of Int'l Conf. on Parallel and Distributed Information systems*, pp. 73-80, 1994.

[4] Badrinath B. R. and K. Ramamritham. Semantics based Concurrency Control: Beyond Commutativity. *ACM Transactions on Database Systems*, 17(1):163–199, Mar. 1992.

[5] Barbara D. and H. Garcia-Molina. The Demarcation Protocol: A Technique for Maintaining Constraints in Distributed Database Systems. *Proc. of the Int'l Conf. on Extending Data Base Technology*, Mar. 1992.

[6] Barbara D. and H. Garcia-Molina. Replicated Data Management in Mobile Environments: Anything New Under the Sun? *Proc. of the IFIP Conf. on Applications in Parallel and Distributed Computing*, Apr. 1994.

[7] Barbara D. and T. Imieliński. Sleepers and Workaholics: Caching Strategies in Mobile Environment. *Proc. of the ACM SIGMOD Conf.*, pp. 1-12, May 1994.

[8] Barghouti, N. and G. Kaiser. Concurrency Control in Advanced Database Applications. *ACM Computing Surveys*, 23(3):269–317, 1991.

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

[10] Chrysanthis P. K., S. Raghuram, and K. Ramamritham. Extracting Concurrency from Objects: A Methodology. *Proc. of the ACM SIGMOD Conf.*, pp. 108–117, May 1991.

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

[12] Chrysanthis P. K., and K. Ramamritham. Synthesis of Extended Transaction Models Using ACTA. *ACM Transactions on Database Systems*, 19(3):450-491, Sept. 1994.

[13] Elmagarmid A. K., editor. *Database Transaction Models for Advanced Applications.* Morgan Kaufmann, 1992.

[14] Herlihy M. Apologizing Versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types. *ACM Transactions on Database Systems,* 15(1):96-124, Mar. 1990.

[15] Imielinski T and B. Badrinath, Mobile Wireless Computing: Challenges in Data Management. *Communication of ACM*, 37(10):18-28, Oct. 1994.

[16] Ioannidis J., D. Duchamp and G. Q. Maguire. Ip-Based protocols for mobile internetworking. *Proc. of ACM Symposium on Communication, Architectures and Protocols,* pp. 235-245, 1991.

[17] Kisler J. and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transanctions on Computer Systems*, 10(1), 1992.

[18] Krishnan P., N. Vaidya and D. Pradham. Recovery in Distributed Mobile Environments. In *Proc. of IEEE Workshop on Advances in Parallel and Distributed Systems,* pp. 83-88, 1993.

[19] Krishnakumar N. and A. Bernstein. High Throughput Escrow Algorithms for Replicated Databases. *Proc. of the 18th Conf. on Very Large Databases*, pp. 175-186, Aug. 1992.

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

[21] Kumar A. and M. Stonebraker. Semantics-based Transaction Management Techniques for Replicated Data. *Proc. of the ACM SIGMOD Conf.*, May 1988.

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

[23] Pitoura E. and B. Bhargava. Building Information Systems for Mobile Environments. *Proc. of the 3rd Int'l Conf. on Information and Knowledge Management,* pp. 371-378, 1994.

[24] Pitoura E. and B. Bhargava. Maintaining Consistency of Data in Mobile Distributed Environments. *Proc. of the 15th Int'l Conf. on Distributed Computing Systems*, June 1995.

[25] Pu C. and A. Leff. Replica Control in Distributed Systems: An Asynchronous Approach. *Proc. of the ACM SIGMOD Conf.*, pp. 377–386, May 1991.

[26] Ramamritham K. and P. K. Chrysanthis. A Taxonomy of Correctness Criteria in Database Applications. (To appear) *Journal of Very Large Databases,* 4(1), Jan. 1996.

[27] Schwarz P. M, and A. Z. Spector. Synchronizing Shared Abstract Data Types. *ACM Transactions on Computer Systems*, 2(3):223–250, Aug. 1984.

[28] Sheth A. and M. Rusinkiewicz. Management of Interdependent Data: Specifying Dependency and Consistency Requirements. *Proc. of the Workshop on Management of Replicated Data*, pp. 133–136, Nov. 1990.

[29] Soparkar N. and A. Silberschatz. Data-value Partitioning and Virtual Messages. *Proc. of the 9th ACM Symposium on Principles of Database Systems*, pages 357-367, 1990.

[30] Tait D. C. and D. Duchamp. Service Interface and Replica Management Algorithm for Mobile File System Clients. *Proc. of the 1st Int'l Conf. on Parallel and Distributed Information Systems*, pp. 190-197, 1991.

[31] Walborn G. and P. K. Chrysanthis. "Using the Escrow Transactional Method to Manage Replicated Data in Disconnected Mobile Operations" *CS Technical Report 94-32*, University of Pittsburgh, June 1994.

[32] Weihl W. and B. Liskov. Implementation of Resilient, Atomic Data Types. *ACM Transactions on Programming Languages and Systems*, 7(1):244–269, Apr. 1985.