JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 16, 535-554 (2000)

On Optimistic Methods for Mobile Transactions

SAN-YIH HWANG
Department of Information Management
National Sun Yat-Sen University
Kaohsiung, Taiwan 804, R.O.C.
E-mail: syhwang@mis.nsysu.edu.tw

We propose a new transaction execution model in a mobile environment where a subset of data items is cached by each mobile host. This execution model allows each mobile host to execute transactions locally and, before their commits, to request certification from the database server on the wired network. The database server certifies a commit request only when the execution of its pertaining transaction was consistent. This protocol behaves like the optimistic concurrency control mechanisms in traditional transaction processing systems. However, traditional optimistic algorithms cannot be applied directly to mobile environments. We propose three strategies for validation of commit requests by the database server. These strategies aim to minimize both the processing overhead and transaction abort ratio. We compare the performance of the proposed strategies via both complexity analysis and simulation and suggest guidelines for choosing the best algorithm for different operating regions.

Keywords: mobile computing, transaction management, databases, optimistic concurrency control, data cache and data replication

1. INTRODUCTION

Data management in mobile computing has attracted a lot of research interest in recent years. The objective of research in this area is to carefully manage scarce resources, such as energy consumption by mobile hosts, wireless communication with limited bandwidth, and restricted connection times due to voluntary or involuntary disconnections, so as to satisfy the data requirements of mobile users. A mobile environment is composed of a set of mobile hosts and fixed hosts on a wired network [13]. Of all the fixed hosts, some hosts, called Mobile Support Stations (MSSs), are identified for communicating with mobile hosts. As wireless communication with MSSs is expensive, slow, and unreliable, each mobile host may put some frequently used data in its local cache to prevent wireless communications as much as possible. However, due to its portable nature, a mobile host has limited capacity in terms of cache storage, making wireless communication unavoidable. To satisfy the data requirements of mobile users while minimizing wireless communication traffic, many researchers have proposed the use of broadcasting mechanisms in a mobile environment [13]. When the required data items are not available in the cache, mobile hosts just passively wait for their arrival by listening to wireless broadcasts. The data broadcasting approach reduces or even eliminates the need for uplink communcations and prevents redundant transmissions of popular data. However, the repeated broadcasts of less popular data could be a waste of communication bandwidth. To negate this disadvantage, a number of studies have taken into account the data demand patterns in deciding on the

Received November 13, 1998; revised July 9, 1999; accepted August 2, 1999.
Communicated by Yi-Bing Lin.
*This research was supported by the National Science Council, R.O.C., under grant NSC 86-2213-E-110-003.

content of both the broadcast channel and the cache [1-3, 8, 10]. Cache replacement strategies were discussed in [12, 7]. To enable power-conservant retrieval of needed data, several index structures imposed on broadcasted data have also been proposed [14, 17].

Another line of research in this area aims to ensure that the cached data is up-to-date. This is not trivial because updates are assumed to occur only at the database server in most works, and their effects may not be reflected in the cache of each mobile host due to difficulties in tracking the mobile hosts that keep the involved data. To reduce the need for uplink communications, some researchers have also advocated the use of broadcasting to disseminate the status of data, called an invalidation report, which can be used by mobile hosts to determine the validity of their cached data. There have been a number of proposals on the contents of invalidation reports and invalidation protocols [5, 16].

All of the above works assumed that updates only occur at the database server on a wired network. Mobile hosts only read data. In [18], the authors proposed an approach to handling update requests of a mobile host that is disconnected from the the wired network. Their approach enables a mobile host to update its local cache while it is disconnected and to propagate the updates to the database server when it is reconnected. However, the focus of their work was to reduce the overhead needed for disseminating update logs to relevant mobile hosts, and transactional consistency was not considered.

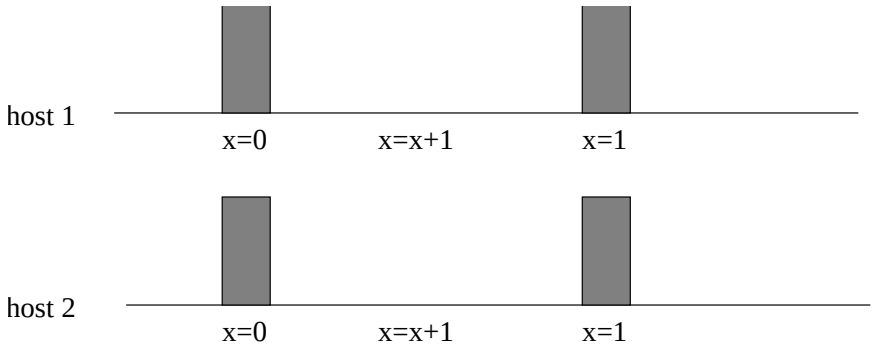
When updates to databases are to be conducted via transactions, most works require that the entire transaction programs sent to the database server and executed there. Some researchers have proposed the client-server transaction execution model that makes use of the computing power of mobile hosts in executing transactions. In this model, mobile hosts are responsible for executing transaction programs with data requests being submitted to the database server interactively. Several works on transaction management reported in the literature were based on the client-server execution model [9, 15, 19]. These works assumed that the same set of data is stored in several database servers on the wired network, some of which could be closer to a mobile host than the rest, and proposed protocols to reduce the communication overhead while at the same time maintaining transactional consistency.

Our work also deals with the transaction management issue in a mobile environment. However, we assume a different execution model: each mobile host executes transaction programs and processes their data requests against its local cache, and only commit requests are sent to the database server on the wired network for certification. Unlike the client-server execution model, this execution model makes use of the transaction processing facilities of mobile hosts and induces less wireless communication, which in turn makes it more suitable to a mobile environment where communication is considered limited and vulnerable.

Allowing each mobile host to process transactions locally without any coordination could cause problems in data consistency. To illustrate, let us assume two transactions T_1 and T_2, resulting from the execution of the following program by different mobile hosts:

begin-transaction $r(x)$ $x = x + 1$ $w(x)$ *commit*.

Initially, let the value of x be 0, and let it be cached by the two mobile hosts *host*1 and *host*2. After the execution of T_1 and T_2, the value of x becomes 1 at both mobile hosts, so the correct value should be 2. This causes the *lost update* problem [6]. The following shows the scenario:



To ensure correct transaction executions, commit operations have to be certified by the database server on the wired network. To this end, optimistic concurrency control algorithms seem to lend themselves very naturally to our execution model. However, they cannot be applied directly. In this paper, we propose three optimistic algorithms for validating transactions to achieve *1-copy serializability*. We also conduct a comprehensive comparison of the proposed algorithms via both theoretical analysis and simulation. To the best of our knowledge, no other work in the literature has been devoted to the investigation of the same topic.

Paper Organization

The rest of the paper is organized as follows. In section 2, we discuss in detail the transaction execution model adopted in our work. In section 3, we describe a scheme for coordinating transaction executions at mobile hosts. Several algorithms for processing commit requests will be proposed and discussed. In section 4, we compare the performance of the proposed algorithms via theoretical analysis and simulation. In section 5, we discuss the extensions to our algorithms in a more general environment. In section 6, we describe the related work in the literature. Finally, in section 6, we give conclusions.

2. MOBILE MODEL

The mobile model we adopt in this paper is similar to that described in [13]. A wired network connects, among others, a set of Mobile Support Stations (MSSs), and each MSS is responsible for communicating with the mobile hosts close to it. When a mobile host moves beyond the effective distance of a MSS, it has to establish a connection with another MSS that is effectively closer. In this paper, we will first limit the scope of our discussion to a single MSS, with which a database server is associated. While this limitation is not really necessary, it will simplify our presentations of the algorithms. Section 5 discusses issues related to extending our algorithms to a more general environment, where more than one MSS communicates with a database server. The reference architecture for now is depicted in Fig. 1.

To speed up data processing and to reduce wireless bandwidth consumption, each mobile host caches a subset of data in its local storage. Each mobile host evaluates the validity of the cached data items by listening to the invalidation reports periodically broad-

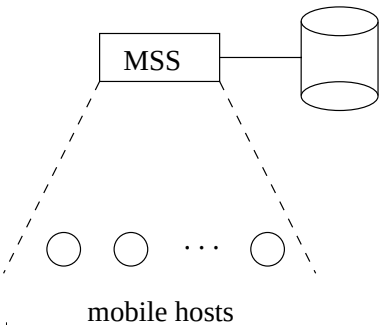


Fig. 1. Reference architecture of a mobile system.

casted by the MSS. An invalidation report provides information about the status of the data items stored in the database of the MSS. Each mobile host repeats the cycle of listening to the invalidation report and processing transactions. The duration of the cycle is called the *broadcasting interval*. Note that we make no assumptions about the content of the invalidation reports and invalidation protocols. Any invalidation protocol [5, 16] can be used with our proposed algorithm to achieve transactional consistency in a mobile environment.

We also assume the use of strict 2PL by each mobile host in handling transactions. This assumption is reasonable because strict 2PL is the dominant concurrency control mechanism used in commercial DBMS products. Read and write operations of a transaction are processed solely by a local mobile host. If the data needed by an operation is not available in the local cache, an uplink request is submitted, and the required data is placed in the local cache before processing. To achieve global consistency, commit requests are first validated by the MSS before they are processed by mobile hosts.

We further assume that transactions executed on mobile hosts are short so that they can be completed before the the next broadcasting interval begins. This assumption is made to ensure that the data values read by a transaction at different times are consistent.

3. THE ALGORITHMS

This section first outlines a protocol used by the MSS and mobile hosts for processing transactions. This protocol is optimistic in the sense that all read/write operations are processed solely by mobile hosts, and that only commit operations are validated by the MSS. We then propose several alternative ways for the MSS to validate commit requests.

3.1 The MSS and Mobile Host Protocol

When a mobile host receives a read/write operation, it checks if the data item required for the operation is available in the local cache. If it is, this operation is processed immediately by the mobile host; otherwise, a data request is sent to the MSS. Upon receiving the value of the data item as well as the timestamp that records the time it was last updated, the mobile host determines if the data item has been updated since the invalidation report was last received. If it was, which implies that the value of the data item is too

new and may not be consistent with the other data values stored in the local cache, the pertaining transaction has to be rejected. Otherwise, the read/write operation is scheduled as usual.

When a mobile host receives a commit operation, it passes it on to the MSS for certification. The MSS determines whether committing the requesting transaction may violate serializability. If it will, the commit request is rejected; otherwise the MSS certifies the associated transaction, allowing it to be applied to the database. Fig. 2 shows how the MSS handles requests from mobile hosts.

When a request from a mobile host arrives at an MSS:
Case 1: a data request *d*
Return the value of *d* and the timestamp of its last update.
Case 2: a commit request from a transaction *T*
If committing *T* violates serializability
then return *reject*
else apply *T* to the database and return *accept*

Fig. 2. Outline for a MSS to handle requests from mobile hosts.

The following subsections discuss how the MSS conducts certification. Specifically, we will discuss how the three types of certifiers, SGT, 2PL, and TO [6], can be modified to work in a mobile environment, resulting in three new certification algorithms.

3.2 Algorithm MTC-SG – Based on the SGT Certifier

The traditional SGT certifier dynamically maintains the serialization graph by adding appropriate edges for each received read/write operation. When a commit operation is received, the SGT certifier checks if the serialization graph, which involves active transactions and transactions recently committed, is acyclic. If it is, the associated transaction is certified and can be executed accordingly. Otherwise, it is aborted, and the related edges are removed from the serialization graph. This approach cannot be applied in a mobile environment because the operations of a transaction are not known to the MSS before its commit time; thus, the MSS is not aware of the existence of active transactions currently executing at mobile hosts. We modify the SGT certifier and call the modified algorithm MTC-SG, denoting **M**obile **T**ransaction **C**ommit using the **S**erialization **G**raph. Instead of keeping active transactions and transactions committed recently in the serialization graph, MTC-SG maintains in the serialization graph only transactions committed by the MSS since last broadcasted. When a transaction T_i requests execution, MTC-SG first adds edges that involve T_i and each conflicting committed transactions, and then checks whether the induced edges cause any cycle in the serialization graph. If so, the commit request is rejected; otherwise, it is accepted.

Recall that we assume that each mobile host employs strict 2PL as its internal concurrency control mechanism. Thus, when a committing transaction *T* conflicts with some committed transaction *T'*, the serialization order between *T* and *T'* can be easily analyzed as follows:

case 1: a read operation $r_T(x)$ of T conflicts with a write operation $w_{T'}(x)$ of T'. In this case, if T and T' execute at the same mobile host, the value of x read by $r_T(x)$ must have been written by $w_{T'}(x)$. Thus, an edge $T' \rightarrow T$ is added to the serialization graph. Otherwise, since T' was processed after the last broadcast, the new value of x written by $w_{T'}(x)$ must not be read by $r_T(x)$. Thus, an edge $T \rightarrow T'$ can be added.

case 2: a write operation $w_T(x)$ of T conflicts with a read operation $r_{T'}(x)$ *of* T'. In this case, $r_{T'}(x)$ must not read the data value of x written by $w_T(x)$. Thus, an edge $T' \rightarrow T$ is added.

case 3: a write operation $w_T(x)$ of T conflicts with a write operation $w_{T'}(x)$ *of* T'. Since the data updated by a transaction is written to the database of the MSS only at the commit time, the serialization order follows the committing order. Again, an edge $T' \rightarrow T$ is added.

Fig. 3 depicts the algorithm MTC-SG for processing a commit request from a transaction T.

```
{CT is the set of transactions committed after the last broadcast.}
{SG is the serialization graph of CT.}
        for each transaction T' in CT do
        begin
                if the read set of T overlaps the write set of T'
                then if (T' and T are from the same mobile host)
                        then add T' → T to SG;
                        else add T → T' to SG;
                if the write set of T overlaps the read set or write set of T'
                then add T' → T to SG;
        end
        if SG contains any cycle then
        begin
                eliminate all edges induced by T from SG;
                reject the commit request;
        end else
        begin
                CT ← CT ∪ {T};
                accept the commit request;
        end
end
```

Fig. 3. Algorithm MTC-SG for handling the commit request of a Transaction T.

3.3 MTC-SQ

The processing overhead of algorithm **MTC-SG** is not trivial because determining the existence of cycles in the serialization graph takes time, especially when the number of committed transactions becomes large. In this section, we look into the possibility of em-

ploying the 2PL certifier or TO certifier in a mobile environment. The traditional 2PL certifier schedules a read/write operation immediately and, when a commit request is received, checks each operation to see if it conflicts with an operation of any other active transaction. If it does, the commit request is rejected; otherwise, it is accepted. As in the case of the SGT certifier, the 2PL certifier also needs information about active transactions, which is not available in a mobile environment. We can of course modify the 2PL certifier in such a way that operations of a committing transaction are compared with those of transactions committed after the last broadcast. This approach, though simple, suffers from low concurrency as no conflicting operations can exist during each broadcasting interval. In fact, conflict operations should be allowed if a particular conflict order among transactions is followed. This idea leads us to investigation of the TO certifier approach.

Like the 2PL certifier, the TO certifier schedules the read/write operations immediately and postpones conflict checking until a commit request is received. The commiting transaction is certified only if all conflicts involving its operations are in timestamp order. Since the operations of these ongoing transactions are not available to the MSS in a mobile environment, conflict checking can be performed by looking at the operations of the committing transaction and those of transactions committed after the last broadcast. Furthermore, a transaction is assigned a timestamp by the MSS when its commit request is received. That is, all conflicts must follow the commit order.

In MTC-SG, commit requests are sequentially processed by the MSS, and a committing transaction T may be earlier in serialization order than any committed transaction T' only when the following two conditions both hold. (These are the conditions in MTC-SG that yield the serialization order $T \rightarrow T'$.)

1. the read set of T overlaps the write set of T', and
2. T' and T are from different mobile hosts.

Thus, a simple way to guarantee a serializable execution is to reject the commit request of a transaction if it satisfies the above conditions. This ensures that the serialization order will follow the commit processing order.

Though simple, the above approach tends to trigger many unnecessary transaction aborts. To illustrate, suppose a transaction that writes some data item x has committed. The above approach will abort all the following transactions that read x. Thus, if an update transaction commits very early within a broadcasting interval, then many transactions may be aborted, including read only transactions. In fact, most read only transactions can be serialized before the conflicting update transactions are since they read the data items that have not yet been updated by the update transactions. Thus, we adopt the following alternative. A sequential order of the committed transactions is maintained by the MSS. When the commit request of a transaction T arrives, the MSS checks if T can be inserted at some point in the sequential order such that the resulting sequential order will comply with the serialization order. If it can, T is committed; otherwise, T is aborted. The serialization order between T and a committed transaction T' is defined as follows.

Definition 1: For a committing transaction T and a committed transaction T', T is said to be serialized before T' if

1. T and T' are executed by different mobile hosts, and
2. readset of T overlaps the writeset of T'.

Definition 2: For a committing transaction T and a committed transaction T', T is said to be serialized after T' if

1. the readset of T overlaps the writeset of T', and T and T' are executed by the same mobile host, or
2. the writeset of T overlaps either the readset or writeset of T'.

Without loss of generality, let the sequential order of the committed transactions be $T_1, T_2, ..., T_n$. When the commit request of a transaction T arrives, we need to find two integers *low* and *up*, where $low = \text{Max}\{i \mid T \text{ is serialized after } T_i, 1 \le i \le n\}$ and $up = \text{Min}\{i \mid T \text{ is serialized before } T_i, 1 \le i \le n\}$, such that $low < up$[1].

Example 1: Let T_1, T_2, T_3 be committed transactions. T_1 and T_3 are executed by the same mobile host while T_2 and the committing transaction T are executed by separate mobile hosts. The readsets and writesets of the three committed transactions and T are shown below:

Transaction	T_1	T_2	T_3	T
readset	$\{x_1, x_2\}$	$\{x_2, x_3\}$	$\{x_1, x_4\}$	$\{x_3, x_4\}$
writeset	$\{x_1\}$	$\{x_2\}$	$\{x_4\}$	$\{x_3\}$

Let the current sequential order be $T_1 \rightarrow T_2 \rightarrow T_3$. It can be observed that T is serialized after T_2 and before T_3. That is, $low = 2$ and $up = 3$. Thus, the new sequential order becomes

$T_1 \rightarrow T_2 \rightarrow T \rightarrow T_3$.

The algorithm is called MTC-SQ and is shown in Fig 4.

```
{n is the number of transactions committed after the last broadcast.}
{CT is an array of transactions committed after the last broadcast.}
low:=0; up:=n+1;
for i:=1 to n do
begin
        if T is serialized before CT[i]
                up = i;
                break;
end
for i:=n downto 1 do
```

[1] If *low* (*up*) is undefined, i.e., T is not serialized after (before) any committed transaction, then $low = 0$ ($up = n + 1$).

```
begin
        if T is serialized after CT[i]
                low = i;
                break;
end
if (low < up)
begin
        for i:=n downto up do
                CT[i+1] = CT[i];
        CT[up] = T;
        n:=n+1;
        accept the commit request of T;
end
else reject the commit request of T;
```

Fig. 4. Algorithm MTC-SQ for handling the commit request of a transaction T.

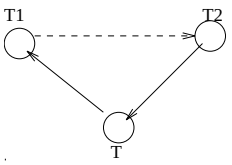
3.4 MTC-Hybrid

Although the processing overhead of MTC-SQ is significantly less than that of MTC-SG, it incurs a potentially higher abort ratio. This happens because there exist scenarios in which a committing transaction fails to find a position in the sequential order of MTC-SQ but it does not introduce any cycle in the serialization graph, as shown by the following example.

Eexample 2: Suppose there are two committed transactions T_1 and T_2, and the sequential order is T_1 followed by T_2. The readsets and writesets of T_1 and T_2 are shown below:

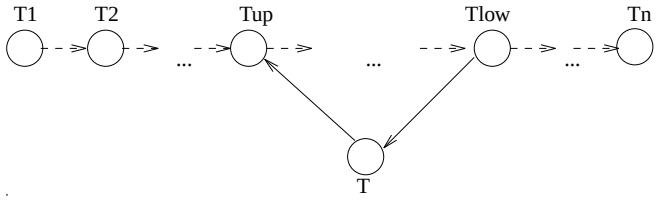
Transaction	T_1	T_2
readset	$\{x\}$	$\{y\}$
writeset	$\{x\}$	$\{\ \}$

The readset of the committing transaction T is $\{x, y\}$, and the writeset is $\{y\}$. Assume that T_1, T_2, and T are executed by different mobile hosts. Thus, T is serialized after T_2 and before T_1. Apparently, T cannot be inserted into the sequential order $T_1 \rightarrow T_2$. However, it can be seen that the resulting serialization graph, where arrows with solid lines represent serialization orders and those with dotted lines indicate sequential order, is acyclic:



The reason why MTC-SQ may cause unnecessary aborts is that it maintains a *total order* of transactions, rather than a *partial order* as maintained by the serialization graph in MTC-SG. It is clear that if a committing transaction passes the test of MTC-SQ, then it must also passes the test of MTC-SG, but the reverse is not always true. This suggests a hybrid approach that combines both MTC-SQ and MTC-SG. Specifically, it first uses MTC-SQ to check whether a committing transaction *T* is able to find a position in the sequential order. If it can, *T* is committed. Otherwise, MTC-SG is employed to see if *T* introduces a cycle in the serialization graph. If it does, the commit of *T* is rejected. Otherwise, *T* is committed, and the sequential order is adjusted.

Let us discuss in more detail the case where a committing transaction *T* does not pass the test of MTC-SQ. In this case, the two integers *low* and *up* computed by MTC-SQ must be *low* > *up*. That is, it is found that *T* must be serialized between T_{low} and T_{up}, and this is not possible because *low* > *up*. This scenario can be visualized as follows:



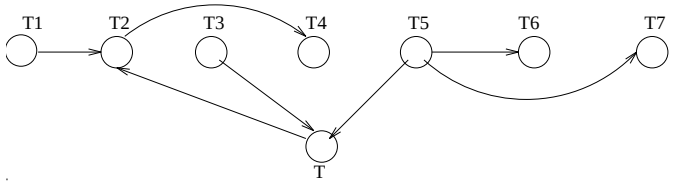
Note that the serialization order between two transactions T_i and T_j can only be T_i followed by T_j if $i < j$. Thus, if there exists a cycle on the serialization graph of $\{T_1, T_2, ..., T_n, T\}$, then this cycle can only involve T_{up}, T_{up+1}, ..., T_{low} and *T*, and no more. Thus, only the subgraph involving $\{T_{up}, T_{up+1}, ..., T_{low}, T\}$ needs to be checked. This observation enables us to dramatically reduce the processing overhead. The approach, called MTC-Hybrid, is shown in Fig. 5.

```
{n is the number of transactions committed after the last broadcast.}
{CT is an array of transactions committed after the last broadcast.}
low:=0; up:=n+1;
for i:=1 to n do
begin
        if T is serialized before CT[i]
                up = i;
                break;
end
for i:=n downto 1 do
begin
        if T is serialized after CT[i]
                low = i;
                break;
end
if (low < up)
begin
         for i:=n downto up do
           CT[i+1] = CT[i];
```

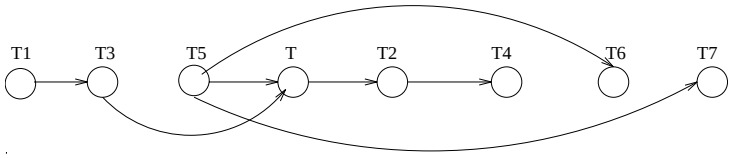
```
                    CT[up] = T;
                n:=n+1;
                accept the commit request of T;
        end
        else begin {cannot find a position in the sequential order}
                {SG' is the serialization graph involving {T_up, T_up+1, ..., T_low}};
                if T introduces any cycle in SG'
                then begin
                        eliminate all edges induced by T;
                        reject the commit request of T;
                end else {adjust the positions of {T_up, T_up+1, ..., T_low} in CT}
                begin
                        {RT is an array of transactions in SG that can be}
                        {reached by T and follow the relative order in CT.}
                        {N RT is an array of transactions in SG that can NOT be}
                        {reached by T and follow the relative order in CT.}
                        for i:=1 to |N RT| do
                                CT[up+i] = N RT[i];
                        CT[up+|N RT|] = T;
                        for i:=1 to |RT| do
                                CT[up+|N RT|+i] = RT[i];
                        n:=n+1;
                        accept the commit request of T;
                end
        end
end
```

Fig. 5. Pseudo code of MTC-hybrid.

Example 3: Let $\{T_1, T_2, ..., T_7\}$ be the set of committed transactions, and let T be the transaction that submits the commit request. The sequential order is $T_1, T_2, ..., T_7$. The following depicts their serialization order. An edge $T_i \rightarrow T_j$ indicates that T_i conflicts with and is serialized before T_j.



It can be seen that $low = 5$ and $up = 2$, and that there do not exist any cycles that involve T in the new serialization graph. The positions of transactions T_2, T_3, T_4, T_5 need to be adjusted such that transactions that can be reached by T (i.e., $\{T_2, T_4\}$) must be placed after those not reached by T (i.e., $\{T_3, T_5\}$) in the new sequential order. The following shows the new sequential order:



4. PERFORMANCE EVALUATION

This section compares the relative performance of the three commit certification algorithms. We consider two primary performance metrics, namely, the abort ratio and processing time complexity. The abort ratio is the number of aborted transactions divided by the total number of commit requests submitted to the MSS. The processing time complexity is the time needed to process a commit request. The desired algorithms should have both a low abort ratio and low time complexity.

In the following, we compare the abort ratios of the MTC-SG, MTC-SQ, and MTC-Hybrid algorithms.

Lemma1: If a commit request is accepted by MTC-SQ, it must also be accepted by MTC-SG.

Proof: We will show that if a commit request is not accepted by MTC-SG, it is not accepted by MTC-SQ either. When the commit request of a transaction T is rejected by MTC-SG, T must introduce a cycle in the serialization graph. Since a cycle cannot be converted to a sequential order, MTC-SQ will also reject the commit of T. □

Lemma 2: If a commit request is accepted by MTC-SG, it must also be accepted by MTC-Hybrid and vice versa.

Proof: **acceptance of MTC-Hybrid ⇒ acceptance of MTC-SG:** MTC-Hybrid first uses MTC-SQ. If a commit request passes the test of MTC-SQ, according to Lemma 1, it must be accepted by MTC-SG. Otherwise, MTC-Hybrid just uses MTC-SG for the second test.

acceptance of MTC-SG ⇒ MTC-Hybrid: If a commit request is accepted by MTC-SG, it will either pass the first test or eventually pass the second test of MTC-Hybrid, which is the same as the test of MTC-SG. □

Theorem 1: Let $A(S)$ denote the abort ratios induced by a commit certification algorithm S. Then,

$$A(\textit{MTC-Hybrid}) = A(\textit{MTC-SG}) \leq A(\textit{MTC-SQ}).$$

Proof: This is straightforward from Lemmas 1 and 2. □

To gain some idea of the run-time performance of MTC-SG, MTC-SQ, and MTC-Hybrid, we will compare their time complexities.

Lemma 3: Let n be the number of committed transactions. Assume that the number of read/write operations in a transaction is constant. The time complexity of MTC-SG is $O(n^2)$.

Proof: MTC-SG uses a serialization graph with n vertices to check whether an incoming commit request will cause a cycle. Since the number of operations in a transaction is constant, the time needed to check if two transactions conflict can also be considered as being constant. Thus, the time complexity of MTC-SG is the cycle detection time, which is $O(n^2)$. □

Lemma 4: Let n be the number of committed transactions. Assume that the number of read/write operations in a transaction is constant. The time complexity of MTC-SQ is $O(n)$.

Proof: Since MTC-SQ scans the committed transactions twice (to compute *low* and *up*), the time complexity of MTC-SQ is $O(n)$. □

Lemma 5: Let n be the number of committed transactions, and let p be the abort ratio of MTC-SQ. Assume that the number of read/write operations in a transaction is constant. The time complexity of MTC-Hybrid is $O(n + pn^2)$.

Proof: If a commit request passes the first test of MTC-Hybrid, it takes only $O(n)$. Otherwise, it has to take extra $O(n^2)$ for the second test. As the probability of failing the first test is the same as the abort ratio of MTC-SQ, the time complexity of MTC-Hybrid is $O(n + pn^2)$. □

Theorem 2: Let $T(S)$ denote the time complexity of a commit certification algorithm S. Then,

$$T(\textit{MTC-SQ}) \leq T(\textit{MTC-Hybrid}) \leq T(\textit{MTC-SG}).$$

Proof: This is straightforward from Lemmas 3, 4 and 5. □

From the above analysis, one can easily see that MTC-Hybrid is superior to MTC-SG under all circumstances because it has the same abort ratio but lower time complexity. Regarding MTC-Hybrid and MTC-SQ, MTC-Hybrid is actually reduced to MTC-SQ when data contention is 0 (i.e., all commit requests are granted by MTC-SQ). Thus, it makes no differences which one is chosen. However, when the data contention becomes higher, it is not clear which approach becomes dominant. The answers to the following two questions may help us decide.

Q1: Is the abort ratio of *MTC-SQ* very close to that of *MTC-Hybrid (MTC-SG)*?
Q2: Is the complexity of *MTC-Hybrid* very close to that of *MTC-SQ*?

If the answer to Q1 is yes for the desired operating region, then we should just use MTC-SQ because it has low processing overhead and sacrifices little in terms of the abort ratio. In contrast, if the answer to Q2 is yes, then one should just choose MTC-Hybrid because it has a lower abortion ratio and only incurs a little higher time complexity.

To answer the above two questions, we designed a simulation model to test the performance of the three algorithms. Table 1 describes the parameters defined in the simulation model and their settings. To exercise a particular system load, we first commit a fixed number of transactions (200 in our base settings), and then use the proposed commit certification algorithms to evaluate the commit requests of a number of randomly generated transactions. Note that these parameter settings, such as database size, may be smaller than we would find in practice. This is quite common in database performance evaluation and is a must in order to obtain performance results within a reasonable amount of time [4]. With proper system scaling, many factors, such as data contention, can model practical situations. Thus, the performance results obtained from the smaller system can reflect the performance of a larger system.

Table 1. System parameter settings.

Parameter	Settings
Database Size	4069 pages
Number of Read Operations in a Transaction	6
Number of Write Operations in a Transaction	0, 1, 2, 3, 4, 5, 6
Number of Committed Transactions	200

Note that resource contention is not considered in this paper. Thus, the various system variables, such as CPU time, disk access time, and communication time, are not modeled. While these system parameters may affect some performance metrics, such as response time and throughput, they are irrelevant to the abort ratio, the primary performance metric considered in our experiments.

Data collection in the experiments was based on the method of replications. Each experiment was repeated several times via different random seeds to obtain a 95% confidence interval. The following figures only show the mean values of the measures.

We first evaluated the effect of data contention on the performance of the proposed three algorithms. Variation in data contention was achieved by changing the number of write operations in each transaction. Fig. 6 shows the abort ratios under the three commit certification algorithms. Overall, as the data contention increased, the abort ratio increased. In an environment where transactions seldom update data, as expected, the three algorithms all performed very well with the abort ratio being close to 0. When transactions were update-intensive, the three algorithms again had similar performance. As a matter of fact, in an environment where transactions write to all the data they read, i.e., *the number of read operations in a transaction = the number of write operations in a transaction*, the abort ratios of all three algorithms became the same. This is because, whenever a committing transaction T conflicts with any committed transaction T', a cycle involving solely T and T' in the serialization graph exists. Thus, T has to be aborted by all three algorithms. In contrast, if T does not conflict with any committed transactions, the commit request of T will be granted by all three algorithms. This explains why the abort ratios of all three algorithms are the same under this extreme condition.

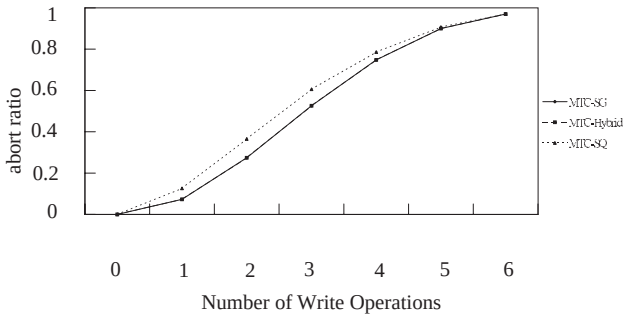


Fig. 6. Abort Ratio with Base Parameter Settings.

As can be seen from Fig. 6, when transactions did incur some but not many updates, MTC-Hybrid and MTC-SG performed significantly better than MTC-SQ. To see how the three algorithms behave under different system loads, we repeated the same experiment under various environments with all the parameter settings being the same except for the *number of committed transactions*. Fig. 7 shows our experimental results with the *number of committed transactions* being set to 70 to simulate an environment with a light load. It can be seen that all three algorithms performed approximately the same even when transactions comprises medium number of updates. This implies that a light transaction load has less impact on the difference in performance of the three algorithms.

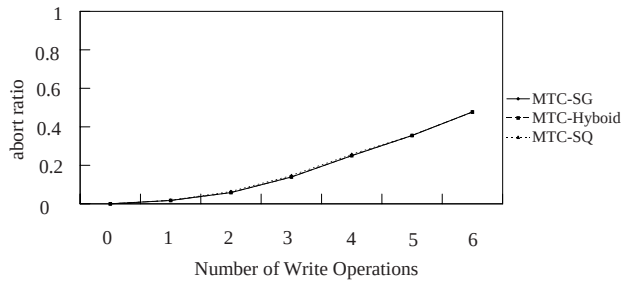


Fig. 7. Abort Ratio with Light System Load.

We then evaluated the effect of data contention on the running time of the three algorithms. Again, we varied the number of write operations in each transaction to reflect the effect of data contention. Time complexity was measured by actually collecting the execution times of 1000 randomly generated commit requests on a Sun IPC workstation. The time unit was one 60th of a second.

Fig. 8 shows the running time of the three algorithms. The running time of MTC-SQ was always lower and insensitive to data contention. Comparing MTC-SG and MTC-Hybrid, one can find that MTC-Hybrid consistently took less time than MTC-SG.

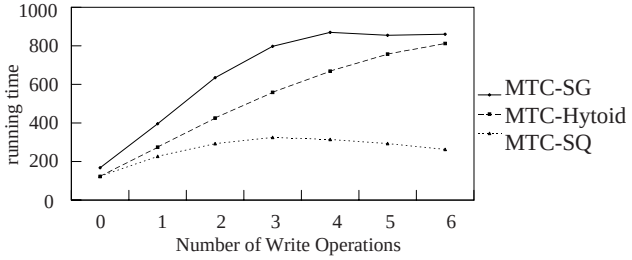


Fig. 8. Running Time of 1000 Transactions with Base Parameter

The choice of commit processing algorithm depends on the tradeoffs between the transaction abort ratio and algorithm running time as well as the system's operating region. In the following, we will suggest some guidelines for determining the best algorithm for different operating regions:

- In a read intensive environment, MTC-SQ and MTC-Hybrid are equally good.
- In an update intensive environment, MTC-SQ becomes the best choice.
- When the workload is light, i.e., the number of transactions executed within each broadcasting interval is small, MTC-SQ should be selected as it has the least running time and performs as well as the other two algorithms.
- When the workload becomes heavier and contains a mix of read-only and update transactions, there is no clean conclusion on which algorithm is the best. Under such an environment, MTC-Hybrid has a lower abort ratio while MTC-SQ incurs less processing time. Thus, it is up to the administrator to decide which factor is considered more important for the applications and/or organization.

5. EXTENSION AND DISCUSSION

In this section, we discuss issues related to extending our work to a more general environment, where multiple MSSs communicate with a single database server, as shown in Fig. 9. In such an environment, it is the database server, rather than the MSS, that handles commit requests. The MSSs serve as brokers in the sense that they just passes requests from mobile hosts on to the database server. Furthermore, this environment has the following characteristics that are different from those of the model we discussed in section 2.

1. Each MSS may have a different broadcasting interval.
2. A mobile host may move from the cell of a MSS to that of another while executing a transaction.
3. A mobile host may miss the broadcasts of some invalidation reports voluntarily or involuntarily.

There is a tradeoff in the length of the broadcasting interval and the degree of divergence in the cached data. If the length of the broadcasting interval is short, then the cached data is quite new, and according to our algorithm, a transaction is less likely to be aborted (because the set *CT* is small). However, the incurred disadvantage is that more downlink

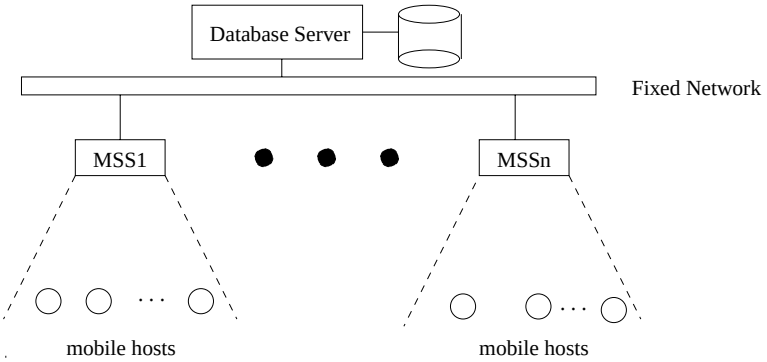


Fig. 9. A more general mobile environment.

communication is needed and mobile hosts have to listen more frequently to the data, which in turn consumes more battery power. Due to this tradeoff, it may be advantageous for the length of the broadcasting interval of an MSS to be decided based on the access pattern of its covering mobile hosts.

Since each MSS may have a different broadcasting interval, it becomes possible for a moving mobile host to never receive an invalidation report even if it listens to the broadcasting channel all the time. This happens when a mobile host constantly moves and stops coverage of each MSS before receiving its broadcast. One way to solve this problem is to constrain the length of the broadcasting interval in each MSS such that a mobile host always receives at least one invalidation report during its stay in the cell covered by an MSS.

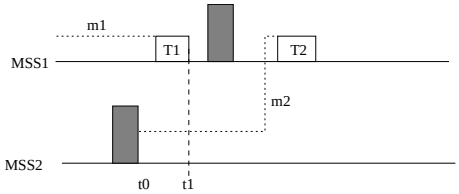
If a mobile host moves from the cell of an MSS to that of another with a different broadcasting interval during the execution of a transaction, problems may occur, as shown by the following example:

Example 4: Suppose the following transaction program is executed by two distinct mobile hosts:

$r(x)$ $x = x + 1$ $w(x)$ *commit*.

Let T_1 denote the first execution carried out by the mobile host m_1, and let T_2 be the second execution by m_2. Initially $x = 0$, and x is cached by both mobile hosts.

At time t_0, m_2 receives an invalidation report at MSS_2. At time t_1, another mobile host m_1 executes T_1 at MSS_1. Then, m_2 moves to MSS_1 and executes T_2. The following depicts the scenario. The shaded rectangles represent the broadcasts of invalidation reports.



When m_2 requests to commit T_2, T_2 could be considered to have read the data written by T_1 (since m_2 executes T_2 after the broadcast of MSS_1). Thus, T_2 will commit successfully, and the final result of x will become 1, which is not acceptable.

The problem in the above example is that the database server treats T_2 as coming from MSS_1 whereas it should be viewed as coming from MSS_2. (Note that m_2 received its last invalidation report from MSS_2.) To handle this case as well as the case where a mobile host may skip some broadcasted invalidation reports, the timestamp of the last invalidation report a mobile host receives has to be kept. In addition, the database server has to maintain the set of transactions committed after *last*, the earliest of all MSS's last broadcast times. Consider the previous example; to validate the commit request of T_2, operations of T_1 have to be taken into account even though T_1 commits before the current broadcast of MSS_1. When the commit request of a transaction executed by some mobile host h arrives, the database server first compares the timestamp of the last invalidation report h receives with *last*. If the timestamp is earlier than *last*, we conclude the time at which h receives its last invalidation report is too old so that transactions committed around that time are not maintained by the database server any more. Thus, the committing transaction T has to be aborted. Otherwise, the committing transaction is compared to the committed transactions in a manner similiar to that previously discussed. Fig. 10 shows the modified algorithm of MTC-SG. Other algorithms can be modified similarly.

```
{CT is the set of transactions committed after last}
{SG is the serialization graph of CT.}
{last_T is the time the mobile host of T receives its last invalidation report.}
if last_T < last then reject this request;
else begin
        for each transaction T' in CT do
        begin
                if the read set of T overlaps the write set of T'
                then if (T' is committed before last_T) or
                        (T' and T are from the same mobile host)
                        then add T' → T to SG;
                        else add T → T' to SG;
                if the write set of T overlaps the read set or write set of T'
                then add T' → T to SG;
        end
        if SG contains any cycle then
        begin
                eliminate all edges induced by T from SG;
                reject the commit request;
        end else
        begin
                CT ← CT ∪ {T};
                accept the commit request;
        end
```

Fig. 10. MTC-SG for handling the commit request of a Transaction T in a general environment.

6. CONCLUSIONS

We have proposed a scheme for handling transactions optimistically in a mobile environment. This scheme assumes that an invalidation report is broadcasted periodically. Each mobile host listens to the broadcasted invalidation report, invalidates its cache, and executes transactions submitted by the user. All the operations of a transaction except commit are processed locally. Commit requests must be submitted to the database server on the wired network for certification. Three algorithms, namely MTC-SG, MTC-SQ, and MTC-Hybrid, for processing of commit requests by the MSS have been described and compared. It has been shown via analysis and simulation that MTC-Hybrid outperforms MTC-SG and takes less time under all circumstances. With respect to the performance of MTC-SQ and MTC-Hybrid, it has been found that both have similar performance and processing time when data contention is low. When data contention is higher, MTC-SQ has a higher abort ratio while MTC-Hybrid incurs higher processing overhead. When data contention becomes extremely high, both MTC-SQ and MTC-Hybrid have high abort ratios. However, MTC-SQ needs less time for processing. Thus, MTC-SQ is superior in this case.

Our future work includes exploring the transaction management issue in a mobile environment containing *low-end* mobile hosts, i.e., those that do not cache data. In such an environment, data is delivered to mobile users by the MSS via broadcasting. Existing protocols either do not handle transactions or do not consider the limitations of mobile systems (e.g., the Datacycle approach [11]). This suggests the need for new protocols.

REFERENCES

1. S. Acharya, R. Alonso, M. Franklin, and S. Zdonik, "Broadcast disks: data management for asymmetric communication environments," in *Proceedings of ACM SIGMOD Interational Conference on Management of Data*, 1995, pp. 199-210.
2. S. Acharya, M. Franklin, and S. Zdonik, "Prefetching from a broadcast disk," in *Proceedings of 12th Interational Conference on Data Engineering*, 1996, pp. 276-285.
3. S. Acharya, M. Franklin, and S. Zdonik, "Balancing push and pull for data broadcast," in *Proceedings of ACM SIGMOD Interational Conference on Management of Data*, 1997, pp. 183-194.
4. R. Agrawal, M.J. Carey, and M. Livny, "Concurrency control performance modeling: Alternatives and implications," *ACM Transactions on Database Systems*, Vol. 12, No. 4, 1987, pp. 609-654.
5. D. Barbara and T. Imielinski, "Sleepers and workaholics: Caching strategies in mobile environment," in *Proceedings of ACM SIGMOD Interational Conference on Management of Data*, 1994, pp. 1-12.
6. P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison Wesley Publishing Company, 1987.
7. B.Y. Chan, A. Si, and H. V. Leong, "Cache management for mobile databases: Design and evaluation," in *Proceedings of the 14th Interational Conference on Data Engineering*, 1998, pp. 54-63.
8. A. Datta, A. Celik, J. Kim, and D. VanderMeer, "Adaptive broadcast protocols to support power conservant retrieval by mobile users," in *Proceedings of 13th Interational Conference on Data Engineering*, 1997, pp. 124-133.

9. A. Elmagarmid, J. Jing, and O. Bukhres, "An efficient and reliable reservation algorithm for mobile transactions," in *Proceedings of 4th Interational Conference on Information and Knowledge Management*, 1995, pp. 90-95.
10. C. Fong, J. Lui, and M. Wong, "Quantifying complexity and performance gains of distributed caching in a wireless network environment," in *Proceedings of 13th Interational Conference on Data Engineering*, 1997, pp. 104-114.
11. G. Herman, G. Gopal, K. C. Lee, and A. Weinrib, "The datacycle architecture for very high throughput database systems," Technical Report, Bell Communications Research Inc., NJ, 1987.
12. Y. Huang, Prasad Sistla, and O. Wolfson, "Data replication for mobile computers," in *Proceedings of ACM SIGMOD Interational Conference on Management of Data*, 1994, pp. 13-24.
13. T. Imielinski and B. R. Badrinath, "Mobile wireless computing: Challenges in data management," *Communications of the ACM*, Vol. 37, No. 10, 1994, pp. 18-28.
14. T. Imielinski, S. Viswanathan, and B. R. Badrinath, "Data on air: organization and access, " *IEEE Transactions on Knowledge and Data Engineering*, Vol. 9, No. 3, 1997, pp. 353-372.
15. J. Jing, O. Bukhres, and A. Elmagarmid, "Distributed lock management for mobile transactions," in *Proceedings of the 15th Interational Conference on Distributed Computing Systems*, 1995, pp. 118-126.
16. J. Jing, O. Bukhres, A. Elmagarmid, and R. Alonso, "Bit-sequences: A new cache invalidation method in mobile environments," Technical Report, Department of Computer Science, Purdue University, IN, 1995.
17. W. C. Lee and D. L. Li, "Using signature techniques for information filtering in wireless and mobile environments," *Distributed and Parallel Databases: An International Journal*, Vol. 4, No. 3, 1996, pp. 205-228.
18. S. Mahajan, M. Donahoo, S. Navathe, M. Ammar, and S. Malik, "Grouping techniques for update propagation in intermittently connected databases," in *Proceedings of the 14th Interational Conference on Data Engineering*, 1998, pp. 46-53.
19. E. Pitoura and B. Bhargava, "Maintaining consistency of data in mobile distributed environments," in *Proceedings of the 15th Interational Conference on Distributed Computing Systems*, 1995, pp. 404-413.

San-Yih Hwang (¶ÀªT¸q) received the B.S. and M.S. degrees from National Taiwan University, Taiwan, in 1984 and 1988, respectively, and the Ph.D. degree from the University of Minnesota, Minneapolis, in 1994, all in computer science.

He is presently an associate professor in the Department of Information Management, National Sun Yat-Sen University, which he initially joined in 1995. Between 1994 and 1995, he was with the Computer and Communication Laboratory, Industrial Technology Research Institute (CCL/ITRI), Taiwan. His current research interests include workflow systems, data management aspects of mobile computing, and parallel IO.