

On Transaction Processing with Partial Validation and Timestamp Ordering in Mobile Broadcast Environments[†]

Victor C. S. Lee^{*}, Kwok-Wa Lam^{*}, Sang H. Son⁺, Eddie Y. M. Chan^{*}

^{*}Department of Computer Science, City University of Hong Kong

⁺Department of Computer Science, University of Virginia

Abstract

Conventional concurrency control protocols are inapplicable in mobile broadcast environments due to a number of constraints of wireless communications. Previous studies are focused on efficient processing of read-only transactions at the mobile clients, neglecting update transactions. In this paper, we design a new protocol for processing both read-only and update mobile transactions. The protocol can detect data conflicts at early stage at the mobile clients and resolve data conflicts flexibly using dynamic adjustment of timestamp ordering. Early data conflict detection saves processing and communication resources whilst dynamic adjustment of timestamp ordering allows more schedules of transaction executions such that unnecessary transaction aborts can be avoided. We performed extensive simulation studies to evaluate the effectiveness of these two features for the performance of the new protocol. The analysis of simulation results showed that both features are effective and contribute differently to the satisfactory performance of the protocol.

Index terms: transaction processing, concurrency control, partial validation, timestamp ordering, broadcast disks, mobile clients

1. Introduction

Recent advances in the development of portable computers and wireless communication networks have introduced the distributed mobile computing paradigm [5, 14]. Many emerging database applications that support a large community of concurrent users demand the broadcast mode for data dissemination. Examples of these applications are mobile auctions, stock trading, next generation road traffic management systems and electronic commerce applications. In stock trading, submitting buy/sell

[†] The work described in this paper was partially supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China [Project No. CityU 1152/01E] and was supported in part by NSF EIA-9900895, CCR-0098269, and IIS-0208758.

transactions on Internet has existed for some time. It should not be long for stock trading to go mobile. In electronic commerce applications such as airline ticketing services, customers carrying portable computer may purchase flight tickets from any airlines with their credit cards. These applications involve use of large amounts of data whose consistency must be preserved in spite of updates, some of which may originate from mobile clients. The updates of data must be disseminated promptly and consistently.

Broadcast-based data dissemination becomes a widely accepted approach of communication in mobile computing environments [1, 2, 3, 4, 22, 25]. In these applications, a server broadcasts data and clients tune in to the broadcast channel to retrieve their data of interest. The distinguishing feature of this broadcast mode is the communication bandwidth asymmetry, where the “downstream” (server to client) communication capacity is relatively much greater than the “upstream” (client to server) communication capacity. This asymmetry is caused by the large disparity between the transmission capacities of servers and clients. Moreover, the limited power capacity of some mobile systems requires them to have no I/O devices. The limited amount of bandwidth available for the clients to communicate with the broadcast server in such environments places a new challenge to implementing transaction processing efficiently.

Acharya *et al.* [1, 2, 3] introduced the concept of Broadcast Disks (Bdisks). The mechanism of the Bdisks uses communication bandwidth to emulate a storage device or a memory hierarchy in general for mobile clients of a database system. It exploits the abundant bandwidth capacity available from a server to its clients by broadcasting data to its clients periodically. In effect, it makes the broadcast channel as a set of disks from which the clients can fetch data in the air. Different data objects may be broadcast at different rates. Thus, hot data objects may be broadcast more often than cold data objects. This can be modelled as many broadcast disks with different speeds of rotation.

Recent studies [15, 18, 19, 20, 22] have focused on transaction processing in the asymmetric bandwidth environment. Most of these studies attempt to handle read-only transactions at the mobile clients efficiently. They used multiple versions of data so that read-only transactions can be processed locally and the number of transaction restarts is reduced. Some other studies attempt to handle update transactions that are required to send back to the server for validation. In summary, these studies used either validation of transactions at the mobile clients or validation at the server [10]. In our view, these studies could show only part of performance characteristics of the asymmetric bandwidth environment.

In this paper, we attempt to take an integrated approach that validation of transactions will be carried out both at the mobile clients and at the server. We also consider a more realistic running workload that read-only transactions and update transactions are specifically handled. At the mobile clients, partial validation will be performed to detect conflicts as soon as possible such that processing and bandwidth resources are saved. At the server, a sophisticated validation algorithm is devised to validate transactions submitted from the mobile clients. We have carried out a series of extensive

simulation studies to analyze the performance characteristics of our new protocol. In the analysis of these performance studies, we can identify the determinant impacts of different features of our protocol on the system in different workload ranges.

2. Related Work

Acharya *et al.* [1] discussed the tradeoffs between currency of data and performance issues when some of the broadcast data objects are updated by processes running at the server and the clients. Baruah *et al.* [6, 8] considered the design of broadcast programs with real-time and fault tolerance constraints for broadcast disks whose contents are occasionally updated. However, in all these work, the updates do not have transactional semantics associated with them either at the server or at the clients. The updates are made only by processes running at the server while the processes at the clients are assumed read-only. Previous work in the Bdisks environment was driven by wireless applications. They are focused on solving the problems associated with the limited number of uplink channels shared amongst a huge number of clients and the problems of voluntary disconnection where a remote client computer system must pre-load its cache before disconnection.

Only limited research work is dealt with transaction processing in the Bdisks environments. Herman *et al.* [13] discussed transactional support in the Datacycle architecture in high-speed networks, which is also an asymmetric bandwidth environment. Datacycle efficiently solves the problem of scalability but puts significant constraints on the database size. However, they use serializability as the correctness criterion that may be very expensive to achieve in such environments. In [22], the authors proposed a weaker correctness criterion to allow non-serializable execution. The weaker correctness criterion allows read-only transactions to read consistent data without synchronization with the server. However, since the weaker correctness criterion is based on view serializability, they showed that the determination of legality of histories with respect to this consistency requirement is NP-Complete. Therefore, to ensure the correctness, they needed to broadcast a huge amount of control information, which can be as large as an $n \times n$ matrix where n is the number of data objects. Moreover, they used traditional optimistic concurrency control to process the update transactions that may not suit the Bdisks environment.

In [18, 19], a number of broadcast methods are introduced to guarantee correctness of read-only transactions. The *multiversion broadcast* approach broadcasts a number of versions for each data item along with the version numbers. In [20], a number of means for storing versions are proposed. Multiple versions of data gives high flexibility to process read-only transactions, thus minimizing transaction

aborts. The performance results show that the overhead of maintaining versions can be justified by increasing concurrency. In the *invalidation-only broadcast*, a read-only transaction is aborted if any data item that has been read by the read-only transaction was updated at the server. For the *conflict-serializability* method, both the mobile clients and the server have to maintain a copy of the serialization graph for conflict checking.

In this paper, we propose a new approach to processing transactions in the Bdisks environments where the updates can be originated from the clients. In the new protocol, the mobile clients share a part of the validation function with the server and are able to detect data conflicts earlier. It is more suitable for the Bdisks environments. In our transaction processing framework, while transactions are processed at the server based on the notion of conflict serializability, the clients can determine whether active transactions can proceed or be aborted.

3. Issues of Transaction Processing in Broadcast Environments

In this section, we consider processing transactions in the Bdisks environments. Since the bandwidth from mobile clients to server is very limited, concurrency control protocols, which require continuous synchronization with the server to detect data conflicts during the execution of transactions such as two phase locking, become handicapped in these environments. This is the reason why almost all the protocols proposed for such environments are based on optimistic approach. In these protocols, transactions are processed unhindered at the clients and are submitted to the server for validation at the end of execution so that updates can be ultimately installed in the central database.

The key step in optimistic approach is the validation where a transaction's destiny is decided. Validation can be carried out basically in two ways: *backward validation* and *forward validation* [12]. In backward scheme, validation is done against committed transactions whereas in forward scheme, validation is carried out against currently running transactions. Forward validation provides flexibility for conflict resolution such that either the validating transaction or the conflicting active transactions may be chosen to restart. So, it is more popular for database systems.

After validation, the eventual termination (commit or abort) of the submitted mobile transactions will be broadcast to the clients in the subsequent broadcast cycles. If the mobile transaction submitted to the server could not pass the validation, it will take a long time for the client to be acknowledged and to restart the failed transaction. For a huge number of clients, this strategy will certainly cause intolerable delays and clutter the server. Consequently, it will have a negative impact on the system performance in terms of response time and throughput.

In addition, there are two major problems using conventional optimistic approach in the Bdisks environments. First, any “serious” conflicts that lead to a transaction abort can only be detected in the validation phase at the server. Therefore, some transactions, which are destined to abort when submitted to the server, are allowed to execute to the end. Such continuation of execution of these to-be-aborted transactions wastes the processing resources of the mobile clients as well as the communication bandwidth. Second, the ineffectiveness of the validation process adopted by the conventional protocols at the server leads to many unnecessary transaction aborts and restarts because they have implicitly assumed that committed transactions must precede the validating transaction in the serialization order. Consider the following example.

Example 1: Consider the following two transactions.

$$T_1 : \quad r_1(x) \ w_1(x)$$

$$T_2 : \quad r_2(x) \ r_2(y)$$

Suppose the execution of the two transactions is as follows.

$$H_1 : r_1(x) \ w_1(x) \ r_2(x) \ c_1 \dots$$

In schedule H_1 , transaction T_1 reads data object x , subsequently updates x , and is sent to the server for validation before transaction T_2 completes. If either of the basic backward or forward validation schemes is used, for T_1 to commit, T_2 will be aborted since T_2 would be deemed to have read a “wrong” version of x . However, this transaction abort can be avoided if T_2 has no more conflicts with T_1 . If we let T_2 commit, we have the following schedule.

$$H_2 : r_1(x) \ w_1(x) \ r_2(x) \ c_1 \ r_2(y) \ c_2$$

The schedule, H_2 , is equivalent to the serialization order of $T_2 \rightarrow T_1$ and H_2 is a serializable schedule since there is no dependency cycle in H_2 [7]. In general, such aborts due to transaction T_2 , which reads a data that is subsequently updated by another successfully committed transaction T_1 , are unnecessary if T_2 does not read any data subsequently written by T_1 either directly or indirectly. In fact, no serialization order between T_1 and T_2 has been built except for the write-read conflict on x . Since the number of read operations is much higher than that of write operations in the Bdisks environments, a large fraction of data conflicts is of the read-write type, and a significant portion of them falls into this category.

Dynamic timestamp interval allocation has been proposed to address the read-write conflicts [9]. Whenever a transaction requests a data access, the system must check for and record the valid timestamp interval for each transaction on the data object accessed. This is necessary since a committing transaction may need to change the valid timestamp interval for other concurrently running transactions accessing the same data objects. The dynamic timestamp allocation only assigns timestamps to transactions when needed and avoids unnecessary aborts of transaction on the occurrence of data conflicts because it allows that the order of transaction commit is different from the serialization order. The timestamp interval

allocation is generally considered to exhibit better performance, as it is more flexible in capturing the partial ordering reflected in the serialization graph among the concurrent transactions. However, it requires many peer-to-peer communications for the adjustment of timestamp intervals of transactions, which cannot be directly applied in the Bdisks environments without any modification.

In this work, we have three objectives in designing the concurrency control protocol in the Bdisks environments.

- Data conflicts should be detected as soon as possible at the mobile clients to avoid any wastage of resources and to help improve the response time of mobile transactions.
- More schedules of transaction executions should be allowed to avoid unnecessary transaction aborts since the clients may suffer from a long delay to realize any transaction aborts and to restart the failed transactions.
- The protocol should not involve the clients to synchronize with each other and the server for data conflict detection.

4. Protocol Design

In this section, we examine an alternative way to adjust the timestamp intervals of transactions dynamically to meet the above three design objectives. We assume a central database server that stores and manages all the data objects at the server. Updates submitted from the clients are subject to validation so that they can be ultimately installed in the central database. Data objects are broadcast by the server and the clients listen to the broadcast channel to perform the read operations. When a client performs a write operation, it pre-writes the value of the data object in its workspace. The clients can have cached copies of the data objects to speed up the read operations. However, we do not pursue in the paper the impact of the cached copies on overall performance.

4.1 Broadcasting of Validation Information

As we mentioned before, it is impracticable for the mobile clients to continuously communicate with each other to exchange information for timestamp interval adjustment in the Bdisks environments. Instead, we make use of the capability of the server to broadcast validation information to the clients so that the clients can adjust the timestamp intervals of their respective active transactions. Clearly, this method cannot guarantee that the mobile transactions can be terminated (committed or aborted) locally at the clients. It is because the clients do not have a complete and up-to-date view of all conflicting

transactions. For instance, the information about the commitment of some conflicting transactions that are submitted to the server after the start of the last broadcast cycle is unknown to the mobile clients for the time being. Thus, all transactions have to be submitted to the server for final validation. Alternatively, the protocol could be implemented at the server by simply requiring the clients to send the information of their read and write sets to the server for one-off validation and timestamp adjustment such that broadcast of validation information can be avoided. However, for such a large population of mobile clients, this approach would certainly overload the server and cause intolerable delay.

Our new strategy places part of the validation function to the clients. In this way, the validation is implemented in a truly distributed fashion with the validation burden shared by the clients. The important issue of this strategy is that the server and the clients should avoid performing the same part of validation function repeatedly. In other words, they should complement each other. We will illustrate how they can complement each other in the following section.

4.2 Timestamp Ordering

To allow more schedules of transaction execution such that unnecessary transaction aborts can be avoided, we adapt the mechanism of timestamp ordering in broadcast environments. We assume that there are no blind write operations. That is, every write operation must be preceded by a read operation on the data object. For each data object, a read timestamp (*RTS*) and a write timestamp (*WTS*) are maintained. The $RTS(x)$ and $WTS(x)$ represent the youngest committed transactions that have read and written data object x . Each active transaction, T_a , at the clients is associated with a timestamp interval, $TI(T_a)$, which is initialised as $[0, \infty)$. The $TI(T_a)$ reflects the dependencies of T_a on the committed transactions and is dynamically adjusted, if possible and necessary, when T_a reads a data object or after a transaction is successfully committed at the server. If $TI(T_a)$ shuts out after the intervals are adjusted, T_a is aborted because a non-serializable schedule is detected. Otherwise, when T_a passes the validation at the server, a final timestamp, $TS(T_a)$, selected between the current bounds of $TI(T_a)$, is assigned to T_a . Let us denote $TI_{lb}(T_a)$ and $TI_{ub}(T_a)$ the lower bound and the upper bound of $TI(T_a)$ respectively. It is clear that whenever T_a is about to read (pre-write) a data object written (read) by a committed transaction T_c , $TI(T_a)$ should be adjusted such that T_a is serialized after T_c .

Let us examine the data conflict resolution between a committed transaction and active transactions on the dependencies of the serialization order. Suppose we have a committed transaction T_c and an active transaction T_a . There are two possible types of data conflicts that can induce the serialization order between T_c and T_a such that $TI(T_a)$ has to be adjusted.

- 1) $RS(T_c) \cap WS(T_a) \neq \emptyset$ (read-write conflict)

This type of conflict can be resolved by adjusting the serialization order between T_c and T_a such that $T_c \rightarrow T_a$. That is, T_c precedes T_a in the serialization order so that the read of T_c is not affected by T_a 's write. Therefore, the adjustment of $TI(T_a)$ should be : $TI_{lb}(T_a) > TS(T_c)$.

- 2) $WS(T_c) \cap RS(T_a) \neq \emptyset$ (write-read conflict)

In this case, the serialization order between T_c and T_a is induced as $T_a \rightarrow T_c$. That is, T_a precedes T_c in the serialization order. It implies that the read of T_a is placed before the write of T_c although T_c is committed before T_a . The adjustment of $TI(T_a)$ should be : $TI_{ub}(T_a) < TS(T_c)$. Thus, this resolution makes it possible for a transaction, which precedes some committed transactions in the serialization order, to be validated and committed after them.

4.3 Validation

The broadcasting of validation information allows the mobile clients to perform part of the validation function. Therefore, they can help to determine whether an active mobile transaction can proceed its execution and to ensure no active transaction that has data conflicts with a committed transaction will ever be submitted to the server. In addition to sharing of validation burden, our protocol also helps early detection of data conflicts and preserves processing resources for useful execution of transactions. In this section, we explain how the server and the clients can complement each other to complete the validation process.

For a data object x read by an active mobile transaction T_a , $TI(T_a)$ will be adjusted at two instances. The first instance is when x is read by T_a and the value of x is written by a committed transaction before T_a reads x . To show this write-read dependency, $TI_{lb}(T_a)$ is adjusted. The second instance is when x is written by another committed transaction after T_a has read x . $TI_{ub}(T_a)$ is adjusted to show this read-write dependency. Therefore, we need to consider two consecutive $WTS(x)$ s to adjust $TI(T_a)$. Once $TI(T_a)$ has been adjusted at both instances, the data object x read by T_a is not required for further validation at the client or at the server because the subsequent write operations will not affect these dependencies. We use $ValidateSet(T_a)$ to record the set of data objects that have been read by T_a and are required to perform final validation at the server. On the other hand, for each pre-write on x , we need to keep on adjusting $TI_{lb}(T_a)$ whenever $RTS(x)$ is larger than $TI_{lb}(T_a)$ to keep track of the dependency of T_a on those committed transactions due to read-write conflicts. After this part of the validation is done at the mobile clients, validation is still necessary at the server since some conflicting transactions may have completed their validation at the server since the last broadcast. This partial validation of pre-write operations at the mobile clients is useful for early detection of data conflicts and helps to prevent the to-be-aborted active transactions from wasting the resources.

In our new protocol, the clients will submit active transactions to the server for final validation if these transactions survive the local partial validation(s) to the end of their execution. Only some of the read operations of the validating transaction (*ValidateSet*) are required to be validated at the server in addition to the pre-write operations. When a transaction passes the validation at the server, a final timestamp, $TS(T)$, which is selected between the current bounds of $TI(T)$ of the validating transaction, is used to update the $RTS(x)$ and $WTS(x)$ for all data objects read and written by the transaction. Note that the value of the final timestamp of each committed transaction indicates the position of the transaction in the serialization order.

Alternatively, to shift the loading of validation from the mobile clients to the server, the pre-write operations can be deferred for validation at the server. However, it also implies to sacrifice early detection of data conflicts. Moreover, there is a trade-off between early detection of read-write conflicts and overheads of timestamp adjustment. In view of the high communication cost, we believe that early detection of read-write conflicts is justified in the Bdisks environments.

5. The Protocol

In this section, the new protocol, Partial Validation with Timestamp Ordering (PVTO), is introduced. In this paper, we only consider a single speed disk. Indices may be used in broadcast disks to help the clients to locate the requested data objects efficiently.

5.1 Transaction Processing at Mobile Clients

We now describe the concurrency control protocol to process transactions at the clients. The clients carry three basic functions:- (1) to process the read/write requests of active transactions, (2) to validate the active transactions using the validation information broadcast in the current cycle, (3) to submit the active transactions to the server for final validation. These functions are described by the algorithms *Process_MROT*, *Process_MT*, *Validate*, and *Submit* as shown in Figure 1, where MROT stands for read-only mobile transactions and MT stands for update mobile transactions.

In order to process transactions at the mobile clients, the server is required to transmit the validation information consisting of the following components.

- *Accepted* and *Rejected* sets contain the identifiers of transactions successfully validated or rejected at the server in the last broadcast cycle;

- $CT_ReadSet$ and $CT_WriteSet$ contain the identifiers of data objects that are in the read set and the write set of the committed transactions in the *Accepted* set;
- $RTS(x)$ is a read timestamp and $FWTS(x)$ and $WTS(x)$ are the first and the last write timestamps in the last broadcast cycle respectively that are associated with each data object x in $CT_ReadSet$ and $CT_WriteSet$. $FWTS(x)$ is used to adjust $TI_{ub}(T_a)$ of an active transaction T_a for the read-write dependency while $WTS(x)$ is used to adjust $TI_{lb}(T_a)$ for the write-read dependency.

The reason for maintaining an additional write timestamp, $FWTS(x)$, is for correct dynamic adjustment of timestamp intervals of transactions at the mobile clients. Note that $WTS(x)$ indicates the last transaction that writes the data object x in the last broadcast cycle. In a broadcast cycle period, there may be more than one transactions updating the same data object x . When an active transaction T_a at the client has read data object x in the last broadcast cycle, we need to adjust $TI_{ub}(T_a)$ to handle the read-write dependency. To capture this read-write dependency, we need to identify the first write operation that immediately follows the read operation of T_a , denoted by $FWTS(x)$. Otherwise, the adjustment of timestamp intervals based on $WTS(x)$ cannot correctly capture this read-write dependency. Obviously, $FWTS(x)$ is equal to $WTS(x)$ if there is only one transaction that has written data object x in the last broadcast cycle.

At the mobile clients, in addition to the timestamp interval $TI(T_a)$ and $ValidateSet(T_a)$ associated with each active transaction T_a , as described in the last section, the following information is maintained to carry out the basic functions.

- $TOR(T_a, x)$ is used to record the value of $WTS(x)$ when T_a reads data object x .
- $CWS(T_a)$ is the set of data objects prewritten by T_a .

5.1.1 Process

The clients process the read/write requests of active transactions and adjust their timestamp intervals. When a read or write request is processed, $TI_{lb}(T_a)$ will be set to $WTS(x)$ or $RTS(x)$ respectively. If $TI(T_a)$ shuts out, T_a is aborted. The value of $WTS(x)$ will be recorded into $TOR(T_a, x)$ and x is put into $ValidateSet(T_a)$ in case of a successful read operation, for the final validation at the server. For a write operation, a pre-write operation is carried out and the new value of x will be stored into T_a 's private workspace.

5.1.2 Validate

The clients listen to the broadcast channel for the validation information periodically. Note that the clients do not need listen to the broadcast channel for the validation information at all times. They need to do so only during a part of the broadcast cycle, possibly at the beginning of the cycle. By

examining the sets of *Accepted* and *Rejected*, they can report to the transaction users about the termination of the transactions submitted in the previous broadcast cycles. In case of failure, the transaction is restarted.

For each data object x in $CT_WriteSet$, the clients check the following conditions. First, if an active transaction, T_a , has pre-written x , which is in the current write set of T_a , $CWS(T_a)$, the transaction is aborted. It is because no timestamp ordering can be made between the committed transactions and T_a for this data conflict type. Second, if x is in $ValidateSet(T_a)$, $TI_{ub}(T_a)$ will be set to $FWTS(x)$ due to the write/read conflict. If T_a can proceed, x is removed from $ValidateSet(T_a)$. For each x in $CT_ReadSet$, if T_a has pre-written x , $TI_{ub}(T_a)$ will be set to $RTS(x)$ due to the read/write conflict. In case of disconnection [21] such that the mobile client is unable to consult the validation information, the mobile client may conservatively set $TI_{ub}(T_a)$ to the time of disconnection or the current upper bound whichever is smaller before disconnection. When the mobile client reconnects to the system, it can continue to perform the remaining operations.

5.1.3 Submit

When an active transaction can proceed to the end of execution, it will be submitted to the server for final validation together with the following information.

- current TI values;
- read/write sets (RS/WS);
- the pre-written values of the data objects in the write set (New_Value);
- the timestamps of data objects in the read set (TOR); and
- the set of data objects that have to perform final validation at the server ($ValidateSet$).

```

Process_MROT ( $T_a, x$ )
{
     $TI(T_a) := TI(T_a) \cap [WTS(x), \infty)$ ;
    if  $TI(T_a) = []$  then abort  $T_a$ ;
    else
    {
        Read( $x$ );
         $ValidateSet(T_a) := ValidateSet(T_a) \cup \{x\}$ ;
    }
    if EndOfTransaction( $T_a$ ) = TRUE then Submit( $T_a$ )
}

```

```

Process_MT ( $T_a, x, op$ )
{
    if ( $op = READ$ )
    {
         $TI(T_a) := TI(T_a) \cap [WTS(x), \infty)$ ;
        if  $TI(T_a) = []$  then abort  $T_a$ ;
        else

```

```

        {
            Read(x);
            TOR( $T_a$ , x) := WTS(x);
            ValidateSet( $T_a$ ) := ValidateSet( $T_a$ )  $\cup$  {x};
        }
    }
    if (op = WRITE)
    {
        TI( $T_a$ ) := TI( $T_a$ )  $\cap$  [RTS(x),  $\infty$ );
        if TI( $T_a$ ) = [] then abort  $T_a$ ;
        else
        {
            Pre-write(x);
            CWS( $T_a$ ) := CWS( $T_a$ )  $\cup$  {x};
            ValidateSet( $T_a$ ) := ValidateSet( $T_a$ ) - {x};
        }
    }
    if EndOfTransaction( $T_a$ ) = TRUE then Submit( $T_a$ )
}

Validate
{
    // results of previously submitted transactions
    for each  $T_v$  in Submitted
    {
        if  $T_v \in Accepted$  then
        {
            mark  $T_v$  as committed;
            Submitted := Submitted - {  $T_v$  };
        }
        else
        {
            if  $T_v \in Rejected$  then
            {
                mark  $T_v$  as aborted;
                restart  $T_v$ ;
                Submitted := Submitted - {  $T_v$  };
            }
        }
    }

    for each active transaction ( $T_a$ )
    {
        if  $x \in CT\_WriteSet$  and  $x \in CWS(T_a)$  then
            abort  $T_a$ ;
        if  $x \in CT\_ReadSet$  and  $x \in CWS(T_a)$  then
        {
            TI( $T_a$ ) := TI( $T_a$ )  $\cap$  [RTS(x),  $\infty$ );
            if TI( $T_a$ ) = [] then abort  $T_a$ ;
        }
        if  $x \in CT\_WriteSet$  and  $x \in ValidateSet(T_a)$  then
        {
            TI( $T_a$ ) := TI( $T_a$ )  $\cap$  [0, FWTS(x)];
            if TI( $T_a$ ) = [] then abort  $T_a$ ;
            else ValidateSet( $T_a$ ) := ValidateSet( $T_a$ ) - { x };
        }
    }
}

```

```

Submit ( $T_a$ )
{
    Submitted := Submitted  $\cup$  {  $T_a$  };
    Submit to the server for global final validation
        with  $TI(T_a)$ ,  $RS(T_a)$ ,  $WS(T_a)$ ,  $New\_Value(T_a, x)$ ,
             $ValidateSet(T_a)$ ,  $TOR(T_a, \bar{x})$ 
    //  $x$  of  $TOR(T_a, x) \in (WS(T_a) \cup ValidateSet(T_a))$ ;
}

```

Figure 1: Functions at the Clients

5.2 The Server Functionality

The server performs two basic functions: (1) to broadcast the latest committed values of all data objects and the validation information and (2) to validate the submitted transactions to ensure the serializability. One objective of the validation scheme at the server is to complement the local validation at the clients to determine whether the execution of transactions is globally serializable. As we mentioned before, the server does not need to perform the validation for some read operations of validating transactions that have already done at the clients. Instead, only the part of validation that cannot be guaranteed by the clients is required to be performed. At the server, we maintain a validating transaction list that enqueues the validating transactions submitted from the clients, but not yet processed.

The server maintains the following information: a read timestamp $RTS(x)$ and a write timestamp $WTS(x)$ for each data object x . Each data object x is associated with a list of k write timestamp versions, which are the timestamps of the k most recently committed transactions that wrote x . For any two versions, $WTS(x, i)$ and $WTS(x, j)$, if $i < j$, then $WTS(x, i) < WTS(x, j)$. The latest version is equal to $WTS(x)$. Note that this is not a multiversion protocol as only one version of the data object is maintained.

The server continuously performs the algorithm described in Figure 2 until it is the time to broadcast next cycle. The server constructs the validation information containing *Accepted*, *Rejected*, *CT_WriteSet*, and *CT_ReadSet* sets that are going to be broadcast in the next cycle by processing the transactions in the validating transaction list one by one. For each data object x pre-written by a validating transaction T_v , if $WTS(x) > TOR(T_v, x)$, it means that at least one committed transaction has written the data object. Therefore, T_v has to be aborted. Otherwise, $TI_{lb}(T_v)$ is adjusted to $RTS(x)$. For each data object x in $ValidateSet(T_v)$, the server locates the timestamp version, $WTS(x, i)$ that is equal to $TOR(T_v, x)$. If $WTS(x, i)$ is not the latest version, $TI_{ub}(T_v)$ is adjusted to $WTS(x, i+1)$. If the server cannot locate the timestamp version for $TOR(T_v, x)$, it means that the version read by T_v is too old such that there is no sufficient information to validate it. Thus, T_v has to be aborted. If $TI(T_v)$ shuts out after the timestamp adjustment, T_v has to be aborted. If T_v is aborted, it is inserted into the *Rejected* set.

If a validating transaction is allowed to commit, a final timestamp indicating its position in the serialization order is assigned to it. The final timestamp is equal to $TI_{lb}(T_v) + \varepsilon$ where ε is a sufficiently small value. For each data object x in the read set and the write set of T_v , $RTS(x)$ and $WTS(x)$ and the timestamp list are updated. The read set and the write set of T_v are added into $CT_ReadSet$ and $CT_WriteSet$ accordingly. When the next broadcast cycle comes, the server broadcasts the validation information.

```

Global_Validate ( $T_v$ )
{
  Dequeue a transaction in the validating transaction list.
  for each  $x$  in  $WS(T_v)$ 
  {
    if  $WTS(x) > TOR(T_v, x)$  then
    {
      abort  $T_v$ ;
       $Rejected := Rejected \cup \{ T_v \}$ ;
    }
    else
    {
       $TI(T_v) := TI(T_v) \cap [RTS(x), \infty)$ ;
      if  $TI(T_v) = []$  then
      {
        abort  $T_v$ ;
         $Rejected := Rejected \cup \{ T_v \}$ ;
      }
    }
  }
  for each  $x$  in  $ValidateSet(T_v)$ 
  {
    Locate  $WTS(x, i) = TOR(T_v, x)$ 
    if FOUND then
    {
      if  $WTS(x, i+1)$  exists then
         $TI(T_v) := TI(T_v) \cap [0, WTS(x, i+1)]$ ;
      if  $TI(T_v) = []$  then
      {
        abort  $T_v$ ;
         $Rejected := Rejected \cup \{ T_v \}$ ;
      }
    }
    else
    {
      abort  $T_v$ ;
       $Rejected := Rejected \cup \{ T_v \}$ ;
    }
  }
  // transaction passes the final validation
   $TS(T_v) :=$  lower bound of  $TI(T_v) + \varepsilon$ 
  //  $\varepsilon$  is a sufficiently small value
  for each  $x$  in  $RS(T_v)$ 
  if  $TS(T_v) > RTS(x)$  then
  {

```

```

        RTS(x) := TS(Tv);
        if x not in WS(Tv)
            CT_ReadSet := CT_ReadSet ∪ { x };
    }
    for each x in WS(Tv)
    {
        WTS(x) := TS(Tv);
        CT_WriteSet := CT_WriteSet ∪ { x };
    }
    Accepted := Accepted ∪ { Tv };
}

```

Figure 2: Validation at the Server

6. Proof of Correctness and Properties

In this section, we prove the correctness of our new protocol, Partial Validation with Timestamp Ordering (PVTO). That is, the PVTO protocol ensures serializability. In the correctness proof, we prove that all committed schedules S produced by the PVTO protocol are serializable. In other words, the serialization graph, $SG(S)$, does not contain any cycle [7].

Lemma 1: Suppose a transaction T_i is aborted by the partial validation at a mobile client. T_i would have been aborted by the global validation at the server if the client had not performed the partial validation on T_i .

Proof: There are three cases in the partial validation for T_i to be aborted by a mobile client. Suppose that the client does not perform the partial validation to abort T_i and T_i is sent to the server for the global validation. We show that the server will also abort T_i in the global validation.

Case 1: if $x \in CT_WriteSet$ and $x \in CWS(T_i)$ then
 abort T_i ;

In this case, the server will find that $TOR(T_i, x) < WTS(x)$ is true and abort T_i .

Case 2: if $x \in CT_ReadSet$ and $x \in CWS(T_i)$ then

```

{
    TI(Ti) := TI(Ti) ∩ [RTS(x), ∞);
    if TI(Ti) = [] then abort Ti;
}

```

In this case, the server will also abort T_i because it uses the same checking as the client.

Case 3: if $x \in CT_WriteSet$ and $x \in ValidateSet(T_i)$ then

```

{
    TI(Ti) := TI(Ti) ∩ [0, FWTS(x)];
    if TI(Ti) = [] then abort Ti;
}

```

```

        else ValidateSet( $T_i$ ) := ValidateSet( $T_i$ ) - { $x$ };
    }

```

In this case, the server will find $WTS(x, i)$, which is equal to $TOR(T_i, x)$ for x in $ValidateSet(T_i)$, and $WTS(x, i+1)$, which is equal to $FWTS(x)$ and abort T_i . The Lemma follows.

The Lemma 1 helps simplify the correctness proof of the new protocol as we can consider only the global validation part at the server.

Lemma 2: Let T_i and T_j be two committed transactions in a schedule S produced by the PVTO protocol. If there is an edge $T_i \rightarrow T_j$ in $SG(S)$, then $TS(T_i) < TS(T_j)$.

Proof : If there is an edge $T_i \rightarrow T_j$ in $SG(S)$, there must exist one or more conflicting operations of one of the following types on data object x .

Case 1: $r_i[x] \rightarrow w_j[x]$

In the PVTO protocol, the serialization order among transactions may not be the same as the chronological order of transaction commitment, T_j may commit before T_i does. Therefore, two possible cases of commit order have to be considered.

a) T_i commits before T_j reaches validation test

When T_j reaches the validation test, for $w_j[x]$, the timestamp interval of T_j will be adjusted: $TI(T_j) := TI(T_j) \cap [RTS(x), \infty)$. It ensures that the final timestamp of T_j , $TS(T_j)$, which is obtained by adding a sufficiently small value to the lower bound of $TI(T_j)$, is greater than $RTS(x)$, which is equal to or greater than $TS(T_i)$ or T_j aborts if the interval shuts out. Therefore, if T_j can be committed, then $TS(T_i) \leq RTS(x) < TS(T_j)$ and $TS(T_i) < TS(T_j)$.

b) T_j commits before T_i reaches validation test

When T_i reaches the validation test, for $r_i[x]$, the timestamp interval of T_i will be adjusted: $TI(T_i) := TI(T_i) \cap [0, WTS(x)]$ where $WTS(x)$ is equal to $TS(T_j)$. It ensures that $TS(T_i)$ is smaller than $TS(T_j)$ or T_i aborts if the interval shuts out. Therefore, if T_i can be committed, then $TS(T_i) < TS(T_j)$.

Case 2 : $w_i[x] \rightarrow r_j[x]$

a) T_i commits before T_j reaches validation test

This case happens when T_i commits before T_j reads x . When T_j reads x , the timestamp interval of T_j will be adjusted: $TI(T_j) := TI(T_j) \cap [WTS(x), \infty)$ where $WTS(x)$ is equal to $TS(T_i)$. It ensures that $TS(T_j)$ is greater than $TS(T_i)$ or T_j aborts if the interval shuts out. Therefore, if T_j can be committed, then $TS(T_i) < TS(T_j)$.

b) T_j commits before T_i reaches validation test

For this case, once T_j commits, an edge $T_j \rightarrow T_i$ would have existed instead of the edge $T_i \rightarrow T_j$. Given this type of data conflict and the serialization order $T_i \rightarrow T_j$, T_j cannot commit before T_i reaches the validation test.

Theorem 3: If S is a committed schedule produced by the PVTO protocol, then S is serializable.

Proof: Consider $SG(S)$, if $T_i \rightarrow T_j$ is an edge of $SG(S)$, then there must exist conflicting operations $p_i[x]$, $q_j[x]$ in S such that $p_i[x]$ precedes $q_j[x]$. Hence, by Lemma 2, $TS(T_i) < TS(T_j)$. If a cycle $T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_n \rightarrow T_1$ existed in $SG(S)$, then by induction, $TS(T_1) < TS(T_1)$. This is a contradiction. So, $SG(S)$ is acyclic and S is serializable. Thus, the Theorem follows.

Lemma 4: All transactions that are committed by the OCC protocol are also committed by the PVTO protocol.

Proof: Assume a transaction T_i is committed by the OCC protocol but is aborted by the PVTO protocol. Let $RC(T_i)$ be the set of recently committed transactions that are committed between the time when T_i starts its execution and the time at which it reaches the validation phase. Since T_i is committed by the OCC protocol, it must be true that $RS(T_i) \cap WS(T_{RC}) = \emptyset$ for each transaction $T_{RC} \in RC(T_i)$. If T_i is aborted by the PVTO protocol, it implies that $TI(T_i)$ shuts out and T_i must read some data objects written by at least some transactions $T_{RC} \in RC(T_i)$. Hence, $RS(T_i) \cap WS(T_{RC}) \neq \emptyset$, which is in contradiction to the assumption, $RS(T_i) \cap WS(T_{RC}) = \emptyset$, that T_i is committed by the OCC protocol. Thus, the Lemma follows.

Lemma 5: All transactions that are aborted by the PVTO protocol are also aborted by the OCC protocol.

Proof: Assume a transaction, T_i is aborted by the PVTO protocol but is committed by the OCC protocol. Since T_i is aborted by the PVTO protocol, it must be true that $TI(T_i)$ shuts out and T_i must read some data objects written by at least some transactions $T_{RC} \in RC(T_i)$. That is, $RS(T_i) \cap WS(T_{RC}) \neq \emptyset$ where $T_{RC} \in RC(T_i)$. If T_i is committed by the OCC protocol, it must be true that $RS(T_i) \cap WS(T_{RC}) = \emptyset$ for each transaction $T_{RC} \in RC(T_i)$, which is in contradiction to the assumption that T_i is aborted by the PVTO protocol. Thus, the Lemma follows.

Theorem 6: The set of transactions committed by the OCC protocol is the subset of transactions committed by the PVTO protocol.

Proof: The theorem follows directly from the Lemma 4, 5 and Example 1 in which T_2 is aborted by the OCC protocol, but is committed by the PVTO protocol.

7. Performance Evaluation

The simulation experiments are aimed at studying the performance of our proposed protocol, partial validation with timestamp ordering (PVTO), and the conventional optimistic concurrency control (OCC) protocol in real-time broadcast disk environments. The major performance of these protocols is the miss rate, which is the percentage of transactions missed their deadlines [24]. Other performance metrics including restart rate, response time and throughput are also collected. The statistics of read-only mobile transactions (MROT) and update mobile transactions (MT) under PVTO and OCC are collected separately.

In order to analyse the relative effectiveness of the partial validation and timestamp ordering mechanisms, we construct another two models. One model employs only the partial validation (PV) mechanism. There is no timestamp ordering and the mobile clients will perform the partial validation to determine whether an active mobile transaction can proceed. If any mobile transaction fails the partial validation in the course of its execution at the mobile client, it will be aborted and restarted. Otherwise, the mobile transaction will be sent to the server for final validation at the end of its execution. Another model employs only timestamp ordering (TO) mechanism. There is no partial validation. A mobile transaction will adjust its timestamp intervals in the course of its execution. Since there is no validation information broadcast by the server, all mobile transactions will be sent to the server for one-off validation at the end of execution.

7.1 Parameter Setting

The simulation model is based on the model in [16] and is implemented using the simulation tool OPNET [17]. It consists of a server, a client, and a broadcast disk for transmitting both the data objects and the required control information. A mobile transaction is processed until it is committed, even the deadline is missed. The deadline of a mobile transaction is assigned as (current time + slack factor \times predicted execution time), where the predicted execution time is a function of transaction length, mean inter-operation delay and broadcast cycle length. A small database helps to intensify data conflicts for creating hot-spot effect. Table 1 lists the baseline setting for the simulation experiments. The baseline setting is similar to the one presented in [22]. These values were chosen in order to create a scenario with high resource utilization and data contention. The number of read / write operations of a transaction is specified by the transaction length. Update transactions at both the mobile client and the server update a data object with 50% probability. At the client, 70% of the transactions are read-only transactions. The

time unit used in the model is in bit-time (the time to transmit a single bit). Therefore, for a broadcast bandwidth of 64 kbps, the inter-operation delay is equivalent to 1 second and the inter-transaction delay is equivalent to 2 seconds. The delays are exponentially distributed. The server has a database of 300 data objects and the size of each data object is 8,000 bits. The data objects that a transaction accesses are uniformly distributed in the database.

Parameter	Value
Mobile Client	
Transaction Length (Number of operations)	4
Read Operation Probability (for update transactions)	0.5
Fraction of ROTs	70%
Mean Inter-Operation Delay	65,536 bit times (exponentially distributed)
Mean Inter-Transaction Delay	131,072 bit times (exponentially distributed)
Slack Factor	2.0 – 8.0 (uniformly distributed)
Server	
Transaction Length	8
Transaction Arrival Rate	1 per 2,000,000 bit-times to 1 per 250,000 bit-times
Read Operation Probability	0.5
Number of Data Objects in Database	300
Size of Data Objects	8,000 bits
Concurrency Control Protocol	OCC with Forward Validation
Priority Scheduling	EDF

Table 1: Baseline setting

7.2 Simulation Results

Figures 3 and 4 show the miss rate of mobile read-only transactions (MROT) and mobile transactions (MT) respectively under different concurrency control mechanisms with varying loading at the server. It can be seen that the performance of the conventional OCC protocol is relatively poor. We observed that the poor performance was due to the major issues discussed in Section 3. All transactions including those “to-be-restarted” transactions are submitted to the server for validation. Subsequently, it takes a long time for the mobile clients to realize any transaction aborts and the ineffective validation mechanism at the server also causes many unnecessary transaction restarts. All these factors hinder the mobile transactions from meeting their deadlines.

It can be observed that the partial validation mechanism improves the system performance. It helps to detect any data conflicts at early stage at the mobile client side. With the timestamp ordering mechanism, any unnecessary transaction restarts can be avoided and therefore number of transaction restarts can be greatly reduced. In particular, it improves the performance of MROT's significantly even without partial validation. In case of MTs, the integration of the partial validation and the timestamp ordering mechanisms helps to push the performance further. On the whole, we find that the PVTO protocol significantly improves the system performance in terms of miss rate.

One of the reasons for the mobile transactions to meet their deadlines is due to reduced number of restarts. Figures 5 and 6 show the restart rate under different concurrency control mechanisms. The results are consistent with those of miss rate. The PVTO protocol effectively reduces the restart rate of both kinds of mobile transactions. However, we can notice that the major contributor is the timestamp ordering mechanism, which removes any unnecessary transaction restarts. The partial validation mechanism alone cannot help to reduce the restart rate. In fact, the mechanism of partial validation is to abort and restart those mobile transactions at early stage at the mobile client side. Other useful performance results such as response time and throughput are shown in Figures 7 to 10. In all cases, the PVTO protocol performs the best. It attains the shortest response time and the largest throughput.

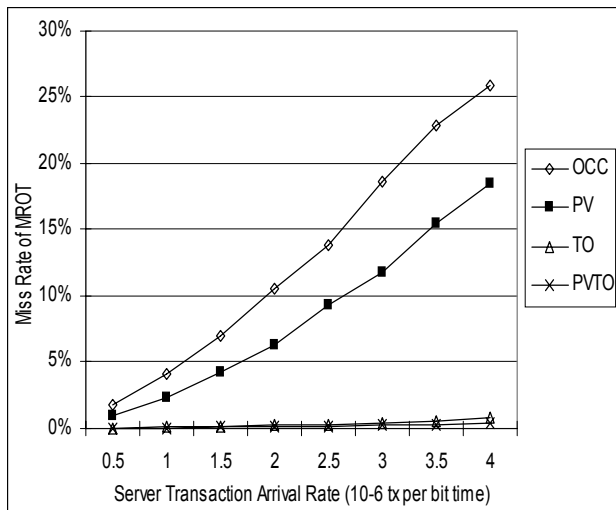


Figure 3: Miss Rate of MROT

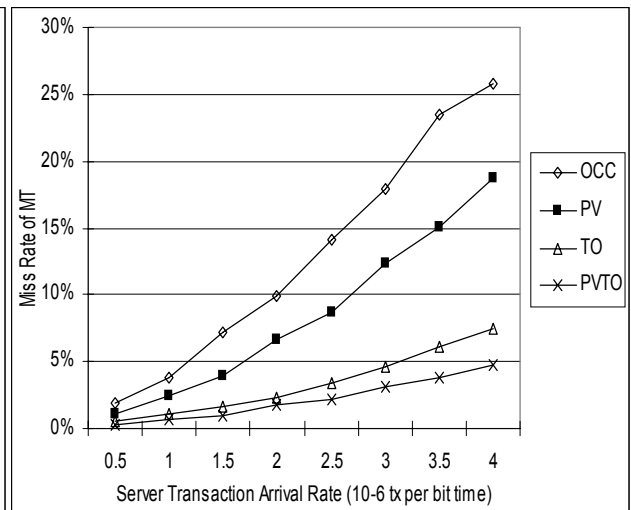


Figure 4: Miss Rate of MT

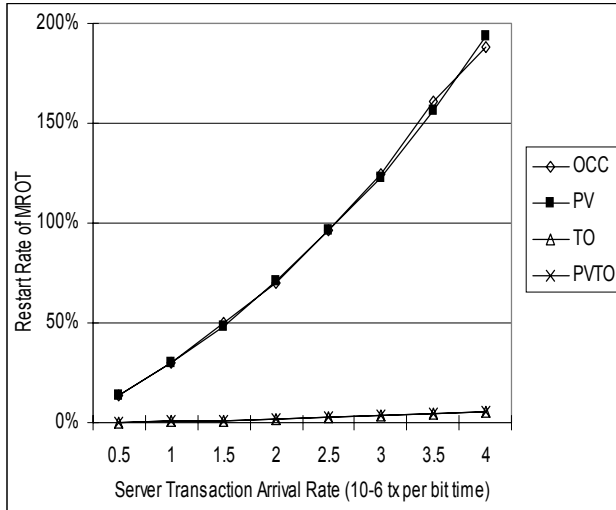


Figure 5: Restart Rate of MROT

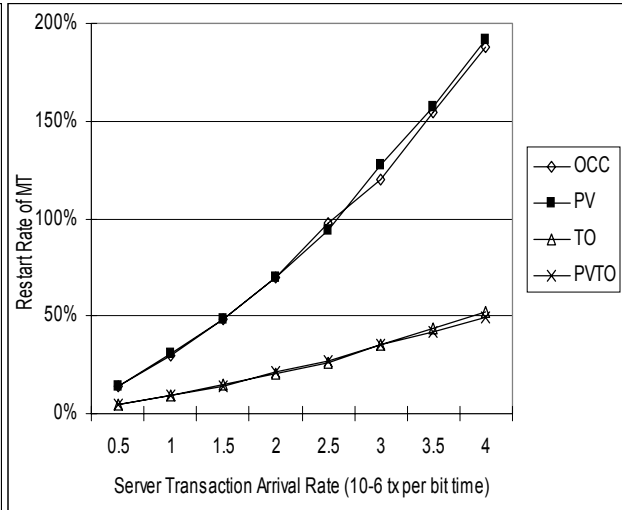


Figure 6: Restart Rate of MT

To better understand the effectiveness of the partial validation mechanism, we collect the early abort rate of mobile transactions. At the mobile client, transactions may be aborted at two different instances. Figures 11 and 12 show the early abort rate of transactions that are aborted when the transactions read the requested data objects from the broadcast cycle. Figures 13 and 14 show the early abort rate of transactions that are aborted when the transactions consult the validation information at the beginning of a broadcast cycle. At both instances, a mobile transaction, which finds that its read phase is affected by the write operations of other committed transactions, will be aborted at the client side. From these figures, we note that the early abort rate under the PV protocol can be up to more than 70% for both types of mobile transactions when the loading at the server is high. On the one hand, it demonstrates the effectiveness of the partial validation mechanism. On the other hand, it illustrates the importance of detecting data conflicts at early stage. Otherwise, resources and bandwidth will be wasted on those destined-to-be-aborted transactions if they were sent to the server for validation. Combined with the timestamp ordering mechanism, the early abort rate is greatly reduced under the PVTO protocol. In spite of this, partial validation is important in terms of saving resources. For instance, in Figure 14, partial validation helps to detect data conflicts at early stage and about 30% of mobile transactions are aborted when the loading at the server is high.

To better understand the performance of the mechanisms relative to OCC, Figures 15 and 16 show the commit probability of mobile transactions. By marking the transactions that would be aborted under OCC, the commit probability of transactions evaluates the probability for transactions to commit if they were processed by OCC with respect to the mechanism to be compared with. For example, if the commit probability of transactions with respect to TO is 80%, that means that 8 out of 10 transactions that

are committed under TO would be able to commit if they were processed by OCC. The results show that the probability decreases as the loading at the server increases. For the partial validation mechanism, a mobile transaction that can be committed under PV should be able to commit under OCC too because the partial validation mechanism only helps to detect data conflict at early stage. It does not help to avoid data conflict. So, the commit probability under OCC with respect to PV is 100%.

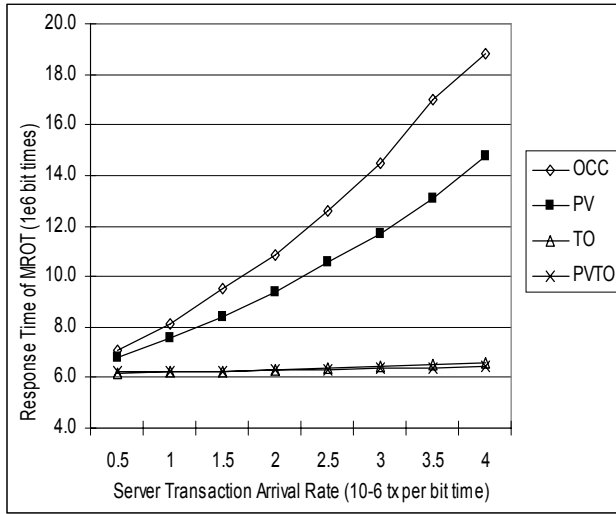


Figure 7: Response Time of MROT

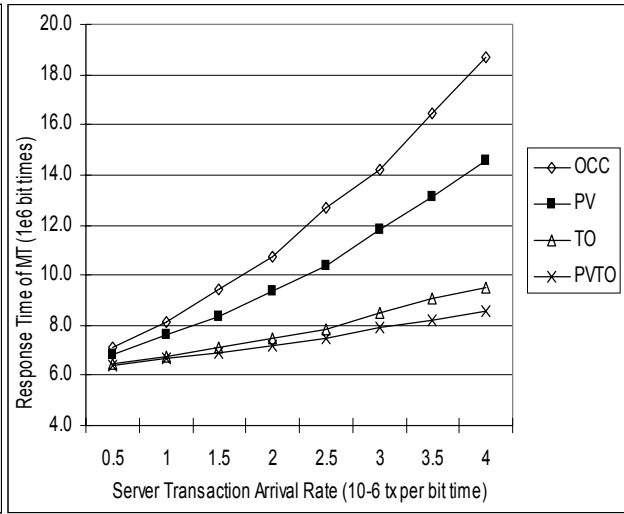


Figure 8: Response Time of MT

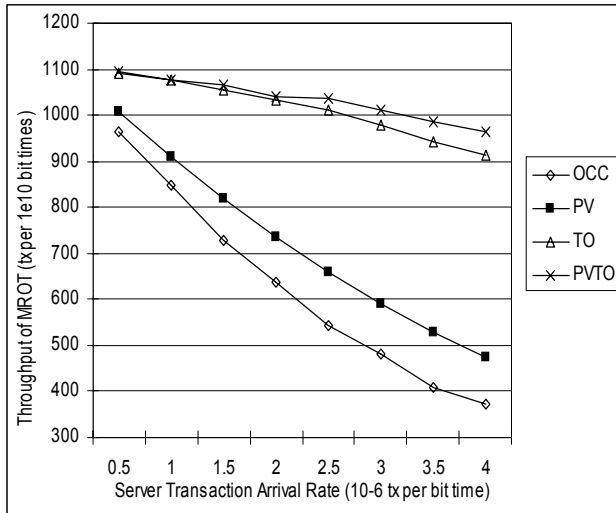


Figure 9: Throughput of MROT

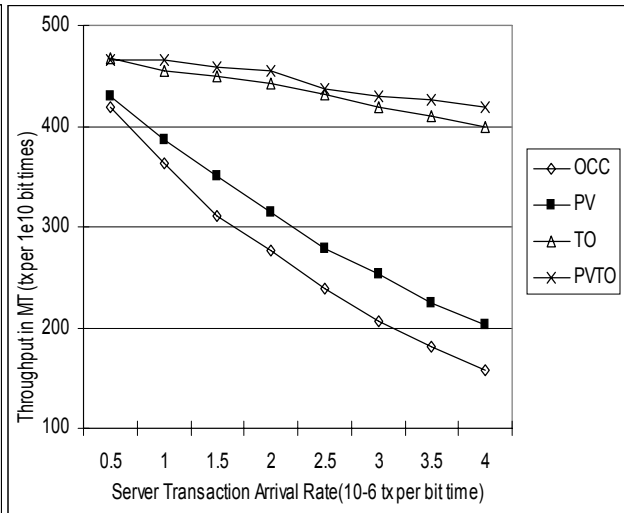


Figure 10: Throughput of MT

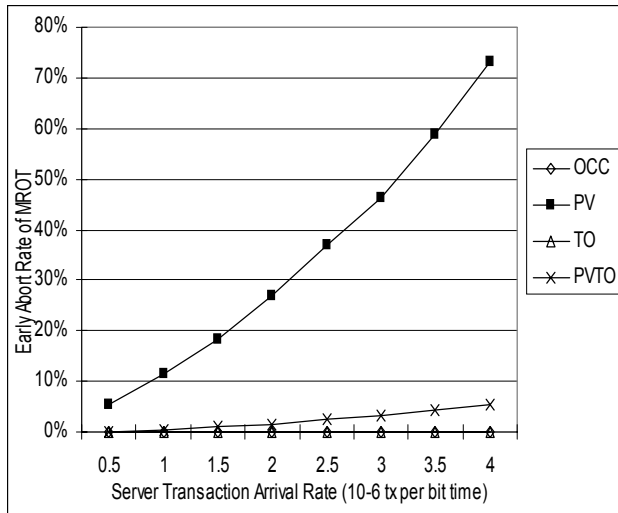


Figure 11: Early Abort Rate of MROT

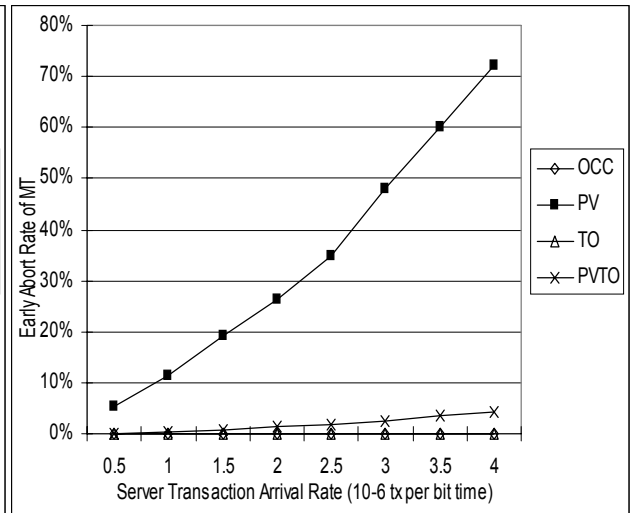


Figure 12: Early Abort Rate of MT

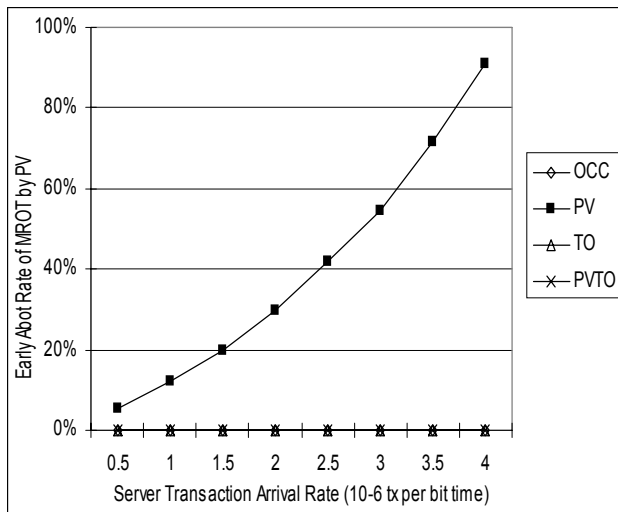


Figure 13: Early Abort Rate of MROT by PV

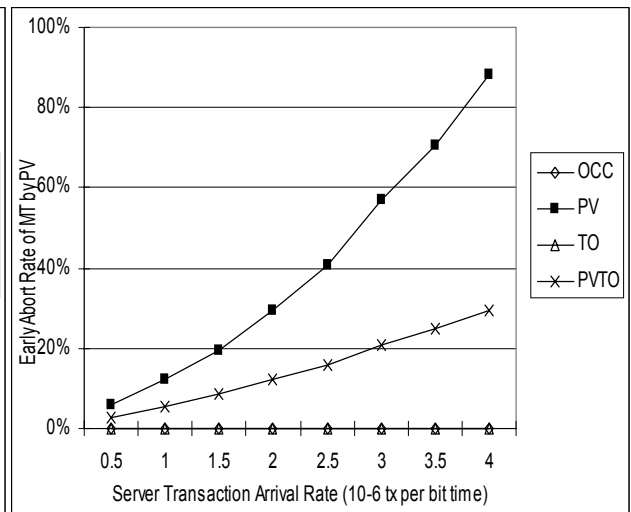


Figure 14: Early Abort Rate of MT by PV

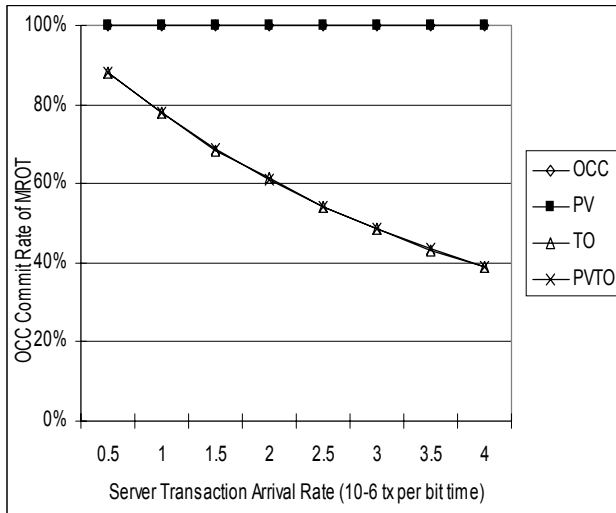


Figure 15: Commit Prob. of MROT in OCC

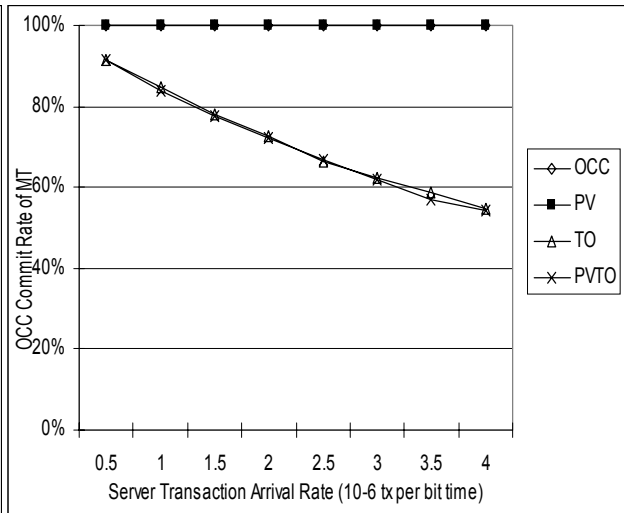


Figure 16: Commit Prob. of MT in OCC

Figures 17 and 18 show the miss rate of MROT and MT respectively with different server transaction write probability given a high loading at the server. When the write probability is low, data conflict, in particular, the read-write type conflict described in Section 3, is rare and the miss rate is low. When the write probability is high, high probability of data conflict leads to a high miss rate. The mechanisms still function satisfactorily and the PVTO protocol performs the best to help mobile transactions to meet their deadlines.

In addition to the server transaction write probability, the database size also affects the data conflict rate. On the one hand, a large database with uniform data access decreases the probability of data conflict. On the other hand, a large database increases the length of the broadcast cycle, which in turn increases the waiting time of mobile transactions and prolongs their execution. As a result, the number of server transactions concurrently executing with the mobile transactions will be increased and the probability of data conflict will be increased accordingly. To see the effect of these two counter-factors, Figures 19 and 20 show the miss rate with different database size given a high loading at the server. The results show that the miss rate remains more or less the same across different database size and suggest that the effect of the two factors is offset by each other. Although there is little impact on miss rate, the response time of mobile transactions is increased linearly with the database size as shown in Figures 21 and 22.

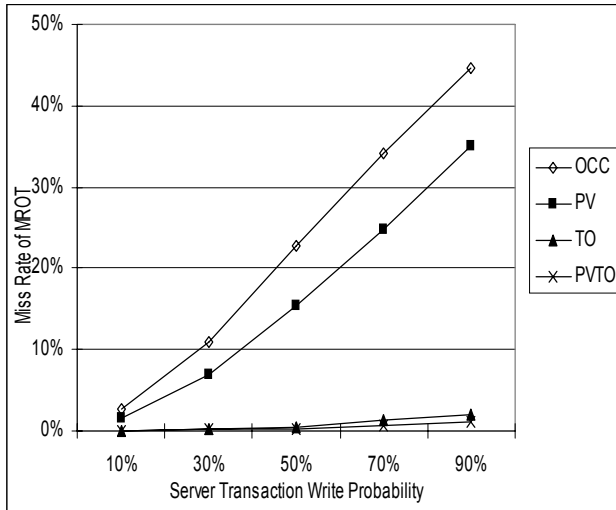


Figure 17: Miss Rate of MROT

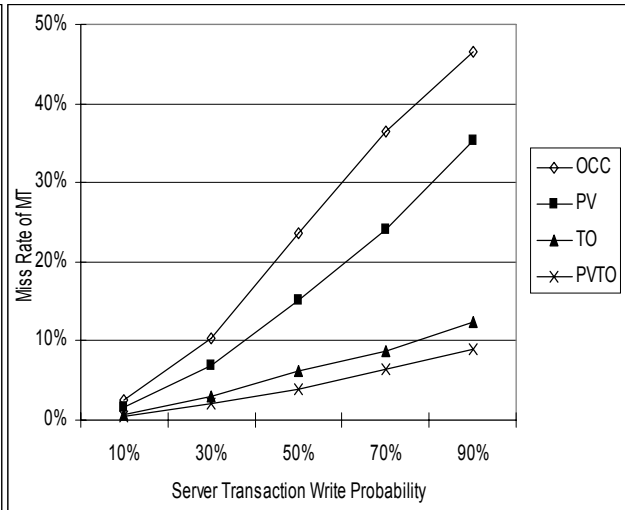


Figure 18: Miss Rate of MT

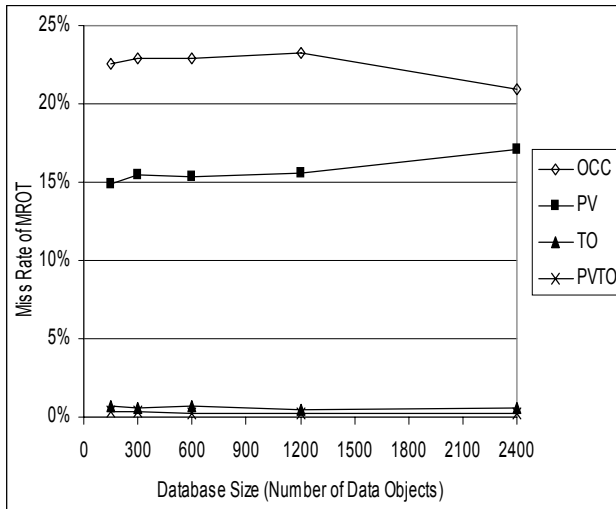


Figure 19: Miss Rate of MROT

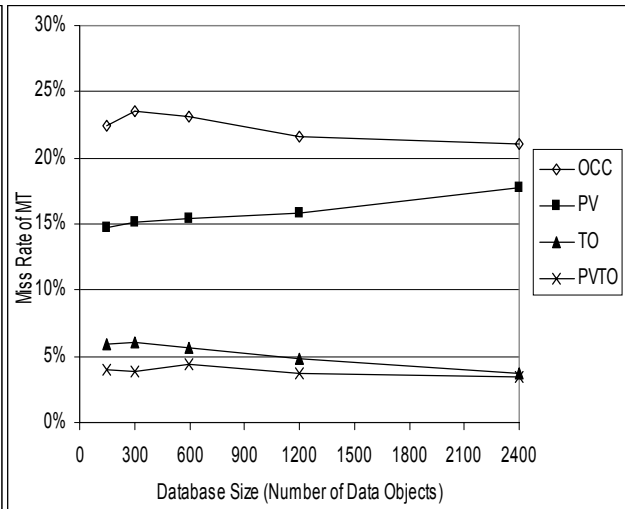


Figure 20: Miss Rate of MT

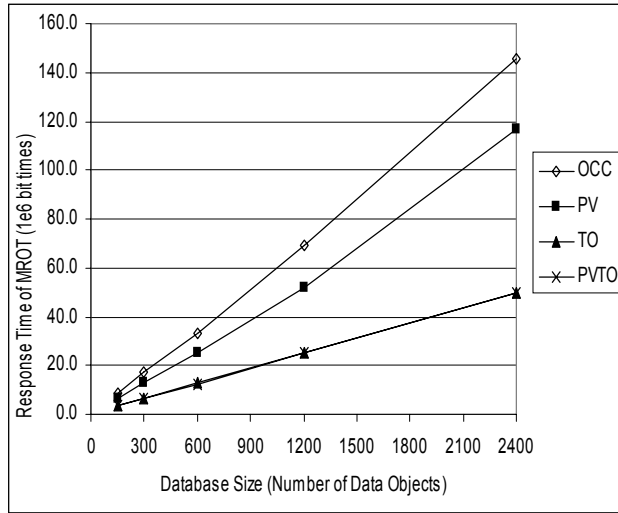


Figure 21: Response Time of MROT

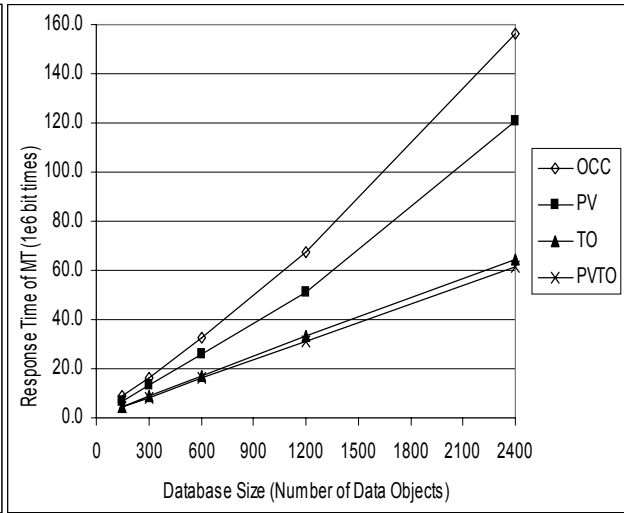


Figure 22: Response Time of MT

8. Conclusions and Future Work

In this paper, we first discussed the issues of transaction processing in broadcast environments. No one conventional concurrency control protocol fits well in these environments due to a number of constraints in the current technology in wireless communication and mobile computing equipment. Recent related research on this area is mainly focused on the processing of read-only transactions. Update mobile transactions are submitted to the server for single round validation. This strategy suffers from several deficiencies such as high overhead, wastage of resources on to-be-restarted transactions, and many unnecessary transaction restarts. These deficiencies are detrimental to the system performance.

To address these deficiencies, we have designed a concurrency control protocol in broadcast environments with three objectives. Firstly, data conflicts should be detected as soon as possible (at the mobile client side) such that both processing and communication resources can be saved. Secondly, more schedules of transaction executions should be allowed to avoid unnecessary transaction aborts and restarts since the cost of transaction restarts in mobile environments is particularly high. Finally, any synchronization or communication among the mobile clients or between the mobile clients and the server should be avoided or minimized due to the asymmetric property in wireless communication.

We have performed a series of extensive simulation experiments to evaluate the performance of the new partial validation with timestamp ordering (PVTO) protocol. The analysis of these simulation results reveals that both partial validation at the mobile clients and dynamic adjustment of timestamp ordering prove to be effective and contribute differently to the performance of the new PVTO protocol.

While the partial validation mechanism (PV) helps to detect data conflict at early stage at the mobile client side and therefore saves resources, the dynamic adjustment of timestamp ordering mechanism (TO) helps to eliminate unnecessary transaction restarts. Combining these two mechanisms greatly improves the system performance in terms of various metrics including miss rate and throughput.

A number of areas are worth further investigation as extension of our work. One area is to consider deadlines in making commitment decision in order to further improve the real-time performance. Another area is the impact of cached copies of data objects at the mobile clients. Caching [11, 23] probably reduces the latency of client transactions by storing a copy or a certain version of hot data objects in the client cache. Without any modification, the proposed protocol can support caching as far as the required information (WTS and RTS) is stored along with each data object to be cached. Moreover, the validation information at the beginning of every broadcast cycle can help to invalidate any stale cached version. The other area is the issue of disconnection where a mobile client voluntarily disconnects a connection with the server. We have briefly mentioned a pessimistic way of handling disconnection with our new protocol in Section 5. We believe that it could be an interesting topic to be investigated further to enhance the system performance.

Reference

- [1] Acharya, S., Franklin, M. and Zdonik, S., "Disseminating Updates on Broadcast Disks," *Proc. of 22nd VLDB Conference*, pp. 354-365, India, 1996.
- [2] Acharya, S., Alonso, R., Franklin, M. and Zdonik, S., "Broadcast Disks: Data Management for Asymmetric Communication Environments," *Proc. of the ACM SIGMOD Conference*, pp. 199-210, U.S.A., 1995.
- [3] Acharya, S., Franklin, M. and Zdonik, S., "Prefetching from a Broadcast Disk," *Proc. of the IEEE Conference on Data Engineering*, pp. 276-285, U.S.A., 1996.
- [4] Aksoy, D. and Franklin, M., "R×W: A Scheduling Approach for Large-Scale On-Demand Data Broadcast," *IEEE/ACM Transactions on Networking*, vol. 7, no. 6, pp. 846-860, December 1999.
- [5] Alonso, R. and Korth, H., "Database System Issues on Nomadic Computing," *Proc. of the ACM SIGMOD Conference*, pp. 388-392, U.S.A., 1993.
- [6] Baruah, S. and Bestavros, A., "Pinwheel Scheduling for Fault-Tolerant Broadcast Disks in Real-Time Database Systems," Technical Report TR-1996-023, Computer Science Department, Boston University, 1996.
- [7] Bernstein, P. A., Hadzilacos, V. and Goodman, N., "Concurrency Control and Recovery in Database Systems," Addison-Wesley, Reading, Massachusetts, 1987.
- [8] Bestavros, A., "AIDA-Based Real-Time Fault-Tolerant Broadcast Disks," *Proc. of the IEEE Real-Time Technology and Applications Symposium*, pp. 49-58, U.S.A., 1996.

- [9] Boksenbaum, C., Cart, M., Ferrie, J. and Pons, J. F., "Concurrent Certifications by Intervals of Timestamps in Distributed Database Systems," *IEEE Transactions on Software Engineering*, vol. 13, no. 4, pp. 409-419, 1987.
- [10] Das, A. and Kai, K. Y., "Tradeoff between client and server transaction validation in mobile environment," *International Symposium on Database Engineering & Applications*, pp. 265-272, 2001.
- [11] Franklin, M. J., Carey, M. J. and Livny, M., "Transactional Client-Server Cache Consistency: Alternatives and Performance," *ACM Transactions on Database Systems*, vol. 22, no. 3, pp. 315-363, September 1997.
- [12] Haerder, T., "Observations on Optimistic Concurrency Control Schemes," *Information Systems*, vol. 9, no. 2, pp. 111-120, 1984.
- [13] Herman, G., Gopal, G., Lee, K. C. and Weinreb, A., "The Datacycle Architecture for Very High Throughput Database Systems," *Proc. of the ACM SIGMOD Conference*, U.S.A., pp. 97-103, 1987.
- [14] Imielinski, T. and Badrinath, B. R., "Mobile Wireless Computing: Challenges in Data Management," *Communication of the ACM*, vol. 37, no. 10, pp. 18-28, 1994.
- [15] Lee, SangKeun, Kitsuregawa, M. and Hwang, Chong-Sun, "Efficient processing of wireless read-only transactions in data broadcast," *Proc. of the Twelfth International Workshop on Research Issues in Data Engineering: Engineering E-Commerce/E-Business Systems RIDE-2EC*, pp. 101 –111, 2002.
- [16] Lee, V. C. S., Lam, Kwok-wa and Son, S. H., "Real-time Transaction Processing with Partial Validation at Mobile Clients," *Proc. of the Seventh International Conference on Real-Time Computing Systems and Applications (RTCSA 2000)*, pp. 473-477, South Korea, December 2000.
- [17] OPNET Modeler/Radio 6.0 (c) 1987-1999 MIL 3, Inc.
- [18] Pitoura, E., "Supporting Read-Only Transactions in Wireless Broadcasting," *Proc. of the DEXA98 International Workshop on Mobility in Databases and Distributed Systems*, pp. 428-433, 1998.
- [19] Pitoura, E. and Chrysanthis, P. K., "Scalable Processing of Read-Only Transactions in Broadcast Push," *Proc. of the 19th IEEE International Conference on Distributed Computing System*, pp. 432-439, 1999.
- [20] Pitoura, E. and Chrysanthis, P. K., "Exploiting Versions for Handling Updates in Broadcast Disks," *Proc. of the 25th VLDB Conference*, pp. 114-125, Scotland, 1999.
- [21] Pitoura, E. and Bhargava, B., "Data Consistency in Intermittently Connected Distributed Systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 11, issue 6, pp 896-915, Nov/Dec 1999.
- [22] Shanmugasundaram, J., Nithrakashyap, A., Sivasankaran, R. and Ramamritham, K., "Efficient Concurrency Control for Bdisks environments," *ACM SIGMOD International Conference on Management of Data*, pp. 85-86, 1999.
- [23] Srinivasa, R. and Son, S. H., "Quasi-consistency and Caching with Broadcast Disks," *Proc. of Second International Conference on Mobile Data Management*, pp. 133-144, 2001.
- [24] Stankovic, J. A., Son, S. H. and Hansson, J., "Misconceptions about Real-Time Databases," *Computer*, vol. 32, no. 6, pp. 29-37, 1999.

- [25] Wu, Simon, Lee, V. C. S. and Lam, Kwok-wa, "Broadcast Transaction Scheduling in Mobile Computing Environments," *Proc. of the 3rd International Conference on Mobile Data Management*, pp. 161-162, Singapore, January 2002.