Using Separate Processing for Read-Only Transactions in Mobile Environment

Eddie Y.M. Chan, Victor C.S. Lee, Kwok-Wa Lam

Department of Computer Science
City University of Hong Kong,
83 Tat Chee Avenue, Kowloon,
Hong Kong.
{ymchan, kwlam}@cs.cityu.edu.hk; csvlee@cityu.edu.hk

Abstract. In mobile environment, the asymmetric communication between mobile clients and server is a distinguishing feature. The conventional query processing mechanisms cannot be directly applied. In this research, we investigate the approach of using separate processing for read-only transactions (ROTs). Since ROTs have a significant proportion in several applications like traffic information service, stock market and auction, using a separate algorithm to process ROTs from update transactions may reduce the degree of data contention. Thus, both the update and read-only transactions can be executed in a more efficient manner. Besides, using separate algorithm for ROTs allows higher degree of concurrency than standard concurrency control protocols. To require a group of ROTs to be conflict serializable with respect to the update transactions, a consistency requirement called Group Strong Consistency is defined. The performance of the protocol for Group Strong Consistency was examined through a series of simulation experiments.

1 Introduction

A read-only transaction (ROT) is a transaction that only enquires information from database, but does not update any data [17]. Many applications in mobile broadcast environment process relatively more ROTs than update transactions. Examples of these applications are online auction, road traffic information system and stock markets. Most users issue ROTs to enquire and monitor the current status rather than update the information inside the system.

ROTs can be processed with general concurrency control protocols that ensure serializability [10], [11], [12]. However, the concurrency control protocols do not differentiate ROTs from ordinary update transactions. In this approach, ROTs may be required to hold locks on large amount of data items for long periods of time, thus causing a high degree of data contention with update transactions.

In mobile environment, there are some characteristics [8] that make the system unique. The limited amount of bandwidth available for the clients to communicate with the server poses a new challenge to implement transaction processing efficiently. Traditional concurrency control protocols require large amount of interaction [14], [15] between a database server and clients. Those interactions are only made possible

M.-S. Chen et al. (Eds.): MDM 2003, LNCS 2574, pp. 106-121, 2003.

with symmetric communication bandwidth likes fixed network. Thus, these protocols become handicapped in mobile environment. Besides, large population size of mobile clients [13] may subscribe the services. The large amount of enquiry makes the processing of ROT an important performance issue in mobile environment.

Broadcast-based data dissemination [1], [2], [3], [9] becomes a widely accepted approach of communication in mobile environment. 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 asymmetric communication property [1], where the "downstream" (server to client) communication capacity is relatively much greater than the "upstream" (client to server) one.

Serializability is the standard notion of correctness in transaction processing [7]. Serializability preserves database consistency. That is, when transactions are processed in a serializable execution, the database is guaranteed to remain in a consistent state. Serializability requires that concurrent transactions, including read-only ones, be scheduled in a serializable way. While this strictness of serializability is necessary for update transactions to maintain database consistency, it may place unnecessary restriction on processing ROTs. Consequently, this may have a negative impact on the system performance.

In this paper, we investigate the approach of using separate algorithm for processing ROTs. The remainder of this paper is organized as follows. In Section 2, we discuss some related work. Section 3 describes the consistency requirements of ROTs. Section 4 introduces a separate algorithm for ROTs. In Section 5, an enhanced algorithm called Group Strong Consistency is proposed for mobile environment. We present preliminary performance results of Group Strong Consistency in Section 6. Finally, we conclude our study in Section 7.

2 Related Work

Transaction processing in mobile environment receives a lot of attention in these few years. Acharya et al proposed the Broadcast Disks [1], [2] (Bdisks) to deliver data in mobile environment. The basic idea is using the downstream to act as the I/O device for mobile clients. Server broadcasts the database repeatedly and mobile clients tune-in to retrieve their data of interest without contacting the server. Bdisks fits the asymmetric communication well because it uses the downstream efficiently and preserves the use of upstream bandwidth.

In [3], the authors consider the database to be partitioned into clusters, cluster is treated as the units of consistency. All data items inside a cluster must be fully consistent while data items stored in different clusters may allow bounded inconsistency. Besides, the cluster configuration is dynamic. Clusters may be created or merged based on different conditions. However, inconsistency may not allow in some applications. Whenever some clusters are merged, the conflicts between clusters have to be solved by roll-backing some transactions.

In [4], [5], [6], a number of algorithms is introduced to guarantee correctness of ROTs in broadcast environment. The multiversion broadcast approach broadcasts a number of versions for each data item along with the version numbers. For the conflict-serializability method, both the clients and server have to maintain a copy of

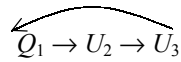
serialization graph for conflict checking. It incurs high overheads to maintain serialization graph. The integration of updates into the local copy of the serialization graph and the cycle detection may be too complicated for certain portable mobile computers. In the invalidation-only approach, a ROT is aborted if any data item that has been read by ROT was updated at the server. In [10], a similar protocol WoundCertifier is suggested and capable to process update transactions. But both methods lack concurrency.

3 Analysis of Data Inconsistency Perceived by a ROT

3.1 Motivation for Relaxing Serializability

Serializability preserves database consistency. Under serializability, both update transactions and ROTs are required to serialize with each other on the same serialization order. If semantic knowledge about transactions is available, it is possible to devise concurrency control mechanisms that guarantee the consistency of the database while allowing non-serializable executions of concurrent transactions. Consider the following example:

Example 3.1
Q_1: r(x) r(y) U_2: r(x) r(y) w(x) U_3: r(y) w(y)
S_1: $r_1(x)$ $r_2(x)$ $r_2(y)$ $w_2(x)$ c_2 $r_3(y)$ $w_3(y)$ c_3 $r_1(y)$ c_1



Serialization Graph of S_1

If conflict-based serialization order is required, it follows that there is a cycle in the serialization graph of S_1. Hence, S_1 is not conflict serializable. Since ROT Q_1 does not change the database state, their presence can be ignored as long as database consistency is the only concern. The remaining update transactions U_2, U_3 would then form a serializable schedule and hence database consistency can be preserved in S_1.

Although S_1 has been proven to preserve database consistency, it may be an incorrect schedule unless Q_1 reads consistent data. Fortunately, this is the case. Since the derivation of new y that Q_1 reads from U_3 is independent of the new x written by U_2, the state of the database read by Q_1 is identical to that if U_2 was removed from S_1. Hence, S_1 is a correct (although non-serializable) schedule.

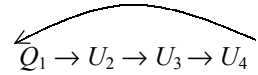
Normally, ROTs and update transactions are processed together with general concurrency control protocols that ensure conflict serializability. As seen from above, non-serializable schedules may still be correct. It is not difficult to see that a non-serializable schedule is still correct if it satisfies both the conditions below:

(i) the concurrent execution of all update transactions (non-ROTs) in a schedule is serializable, **and**
(ii) each ROT involved in the schedule does not read inconsistent data.

3.2 Data Inconsistency Perceived by a ROT in a Non-serializable Schedule

The following example illustrates a simple scenario in which a ROT reads inconsistent data.

Example 3.2
Q_1: r(x) r(z) U_2 : r(x) w(x) U_3: r(x) r(y) w(y) U_4: r(y) r(z) w(z)
S_2 : $r_1(x)$ $r_2(x)$ $w_2(x)$ c_2 $r_3(x)$ $r_3(y)$ $w_3(y)$ c_3 $r_4(y)$ r_4 (z) w_4 (z) c_4 $r_1(z)$ c_1



Serialization Graph of S_2

In the schedule S_2, a ROT Q_1 reads a data object x which is subsequently written by U_2. Then, U_3 reads the new value of x from U_2 and use this new value of x to update data object y. U_4 then reads this new value of y from U_3 and use this new value of y to update another data object z. Finally, Q_1 reads the new value of z. Since this new value of z read by Q_1 depends on the new value of x produced by U_2, it should be considered inconsistent with the original value of x read by Q_1. Hence, Q_1 reads inconsistent data and S_2 is incorrect, despite the database consistency is preserved by the serial execution of U_2, U_3 and U_4. In order to facilitate the following investigation, some definitions concerning derivation of data values are given below:

Definition 1: A value y_1 of a data object y is said to be *derived directly* from a value x_1 of another data object x if either of the following conditions hold:

(I) a committed update transaction has performed a read operation r(x_1) and then a write operation w(y_1), without updating the value of x, **or**

(II) a committed update transaction has performed write operations w(x_1) and w(y_1) after reading the initial value x_0 of the data object x.

Definition 2: A value y_1 of a data object y is said to be derived indirectly from a value x_1 of another data object x if both of the following conditions hold:

(I) the value y_1 of the data object y is derived directly from some value z_1 of another data object z, **and**

(II) the value z_1 is derived either directly or indirectly from the value x_1 of the object x.

For instance, the value of z written by U_4 in Example 3.2 is derived *directly* from the value of y written by U_3, which in turn is derived *directly* from the value of x written by U_2. Thus, the value of z written by U_4 is derived *indirectly* from the new value of x written by U_2. Q_1 reads inconsistent data because Q_1 have already read an older version of x.

If we assume that the consistency of database is preserved by serializable execution of update transactions, the following conclusion can be arrived:

Theorem 1: A read-only transaction Q reads inconsistent data if and only if:

(I) Q has read some data object x whose value is subsequently updated by an update transaction U, **and**

(II) Q later reads some data object y whose value is derived, either directly or indirectly, from the new value of x written by U.

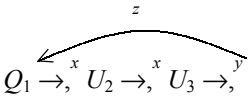
3.3 Data-Passing Graph (DP-graph)

DP-graph is defined in order to help visualizing the formation of inconsistent data read by a ROT. Each read-only transaction Q is associated with a DP-graph, which is a directed graph with Q and its dependent update transactions (transactions that may affect the database state read by Q) as its nodes. Directed edges are added to join two nodes under either of the following cases:

Case 1: When a transaction (Q or U_i) reads a data object x from an update transaction (U_j), a directed edge with label x is added from the latter to the former (*i.e.* $U_j \rightarrow^{x} Q$ or $U_j \rightarrow^{x} U_i$).

Case 2: When a ROT, Q, has read some data object x, and the value of x is subsequently updated by an update transaction U before Q commits, a directed edge with label x is added from Q to U (*i.e.* $Q \rightarrow^{x} U$).

Based on Theorem 1, a read-only transaction Q reads inconsistent data if there is a cycle in the DP-graph of Q. For example, consider the DP-graph of Q_1 in Example 3.2 below:



(1) the edge $Q_1 \rightarrow^{x} U_2$ shows that Q_1 has read x and the value of x is subsequently updated by U_1 before Q_1 commits, and

(2) the path $Q_1 \rightarrow^{x} U_2 \rightarrow^{x} U_3 \rightarrow^{y} U_4 \rightarrow^{z} Q_1$ shows that Q_1 has read a data object z whose value is derived indirectly from the new value of x written by U_2.

(3) Q_1 has read inconsistent data because there is a cycle in the DP-graph.

Based on the above analysis, new algorithms for processing ROTs separately from update transactions with data consistency preserved can be achieved by designing algorithms that never allow the formation of cycles in DP-graphs.

4 Separate Algorithm for ROTs

For the sake of exposition, we describe a basic separate algorithm to process a single ROT only, named the NRS-Algorithm. We adopt the following assumptions:

(a) It is possible to distinguish read-only transactions from update transactions upon their arrival.

(b) A transaction can read only those values that are written by committed transactions.

(c) There are no blind write operations, *i.e.* every write operation on a data object must be preceded by a read operation on that data object.
(d) A transaction does not read from or write to the same data object more than once.
(e) All update transactions are themselves globally serializable, which is guaranteed by the standard concurrency control protocols.

Since the serializability of update transactions is assumed to be guaranteed, the following discussion will only examine schedules in which all the update transactions are executed in a serial manner, just for the sake of simplicity. Moreover, transactions are assumed to read only those values that are written by committed transactions. It is thus reasonable as well as convenient to consider ROTs to have their read operations interleaved between two consecutive update transactions, but not between operations within an update transaction.

4.1 No-Read-Set of Read-Only Transactions

For each active read-only transaction Q, a **No-Read-Set** is maintained. We indicate this set by NRS(Q). During the execution of Q, all those data objects with values that are found to be inconsistent with those already read by Q are placed into NRS(Q).

Each NRS(Q) of a read-only transaction Q is initialized as ϕ. After then, NRS(Q) will be updated in either of the following holds:

Rule 1: An data object x is added to NRS(Q) if Q has read x and subsequently updated by a committed update transaction U.

Rule 2: If the value of a data object y is derived either *directly* or *indirectly* from value of some data objects that is already in NRS(Q), y is added to NRS(Q).

If the data object x is added to NRS(Q) according to *rule 1*, it corresponds to a path $Q \rightarrow^{x} U$ in the DP-graph of Q. On the other hand, if the data object y is added to NRS(Q) according to *rule 2*, it means that y is inconsistent with the value of x that Q has read before. It follows that Q should not be allowed to read those data objects already in NRS(Q), in order to read consistent data. In fact, *rule 1* and *rule 2* respectively handles *case 1* and *case 2* described in the previous section.

4.2 The NRS-Algorithm

Let CRS(Q) denote the current read set of a read-only transaction Q, *i.e.* the set of data objects that Q has already read at the moment. Let RS(U) and WS(U) be the read set and write set of an update transaction U respectively. In order to maintain the data consistency in the view of an active read-only transaction Q_a, the following NRS-Algorithm is implemented.

```
NRS(Qa):= φ ;
do while Qa is active
{
 Whenever an update transaction Uc commits, do
 {
  for each x ∈ WS(Uc)∩CRS(Qa)   do
    NRS(Qa):= NRS(Qa)∪{x};
  if RS(Uc)∩NRS(Qa) ≠ φ  then
    for each y ∈[WS(Uc)\CRS(Qa)]   do
      NRS(Qa):= NRS(Qa)∪{y};
 }
}
```

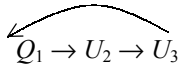
(Rule 1)

(Rule 2)

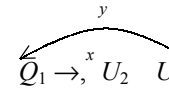
Abort Rule : Q_a aborts when it tries to read a data object in NRS(Q_a).

The following example illustrates the NRS algorithm.

Example 4.1
Q_1: r(x) r(y)
U_2: r(x) r(y) w(x) U_3 : r(y) w(y)
S_3: $r_1(x)$ $r_2(x)$ $r_2(y)$ $w_2(x)$ c_2 $r_3(y)$ $w_3(y)$ c_3 $r_1(y)$ c_1



Serialization Graph of S_3



DP-Graph of Q_1

S_3 is not serializable because there is a cycle in its serialization graph. Nevertheless, it is still a correct schedule since the update transactions are executed in a serial manner and the read-only transaction Q_1 does not see any inconsistent data (DP-Graph of Q_1 is acyclic). The development of the NRS of Q_1 during the successful commitments of the update transactions is shown in the following table:

operation	NRS(Q_1)
$r_1(x)$	ϕ
c_2	{ x }
c_3	{ x }
$r_1(y)$	{ x }
c_1	OK

Result: The algorithm allows Q_1 commit.
Explanation: When Q_1 executes r(x), NRS(Q_1) = ϕ and hence Q_1 reads x successfully. When U_2 commits, x is added to NRS(Q_1) since $x \in$ WS(U_2)∩CRS(Q_1). When U_3 commits, y is not added to NRS(Q_1) in both cases since $y \notin$ CRS(Q_1) and the only element in NRS(Q_1), x, is not read by U_3. Consequently, when Q_1 executes r(y), $y \notin$ NRS(Q_1) and hence Q_1 reads y successfully and finally it commits.

5 Group Strong Consistency

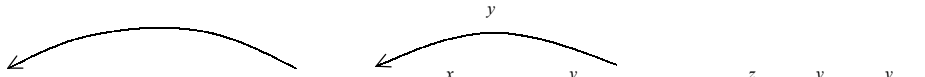
In this section, we consider more than one ROT that perceive the same serialization order with respect to the update transactions they directly or indirectly read from. In other words, ROTs need to be serialized with each other as well as the update transactions. This requirement is termed as **Group Strong Consistency**. The group concept is introduced to characterize all ROTs in the same group that meet the strong consistency requirement. Let us consider the following example.

Example 5.1

Q_1 : r(x) r(y) Q_2 : r(y) r(z)

U_3 : r(x) r(z) w(x) U_4 : r(z) w(z) U_5 : r(y) w(y) U_6 : r(y) w(y)

S_5: $r_1(x)$ $r_2(y)$ $r_3(x)$ $r_3(z)$ $w_3(x)$ c_3 $r_4(z)$ $w_4(z)$ c_4 $r_5(y)$ $w_5(y)$ c_5 $r_6(y)$ $w_6(y)$ c_6 $r_1(y)$ c_1 $r_2(z)c_2$



Serialization Graph of S_5 **DP-Graph of Q_1** **DP-Graph of Q_2**

Since the DP-graph of both Q_1 and Q_2 are acyclic, they do not read any inconsistent data themselves. However, Q_1 and Q_2 do not perceive the same serialization order with respect to the update transactions U_3 , U_4 , U_5 and U_6. So, in case that both Q_1 and Q_2 belong to the same group while group strong consistency is required, Q_2 should be aborted after Q_1 successfully commits.

In order to achieve group strong consistency within a certain group of ROTs, the DP-graphs have to capture the information about the read-write conflicts between update transactions. In particular, U_4 in the above example should be serialized after U_3 because of the read-write conflict between $r_3(z)$ and $w_4(z)$. To deal with this conflict, the following rule is added to the NRS algorithm:

Rule 3: When an update transaction U_c commits, the data objects in the read set of U_c will be added to the current read set, CRS(Q_a), of the active read-only transaction Q_a if (i) U_c updated the value of some data object after Q_a has read it, or (ii) U_c has read some data objects which is already in NRS(Q_a).

i.e. `if WS(U_c)∩CRS(Q_a)≠ϕ or RS(U_c)∩NRS(Q_a)≠ϕ then`
`CRS(Q_a):= CRS(Q_a)∪RS(U_c);`

Rule 3 is a little bit tricky because it adds data objects that are not yet read by Q_a to CRS(Q_a). The rationale behind the rule is as follows. When the conditions for rule 3 holds, there must be a path from Q_a to U_c in the DP-graph of Q_a. In other words, U_c must be serialized after Q_a. Suppose that U_c has read-write conflict with another update transaction U, *i.e.* U write a certain x after U_c has read it. In this case, U must be serialized after U_c and hence it must be also serialized after Q_a. So Q_a must be forbidden to read anything from U. If the data objects in RS(U_c) are added to CRS(Q_a) according to rule 3, any of them that is subsequently updated by U will be

placed into NRS(Q_a) and hence Q_a cannot read the new values of these data objects from U in a later stage. It must be emphasized that Q_a is still allowed to read those data objects in RS(U_c) as long as they are not updated by another update transaction.

In addition, rule 4 is needed:

Rule 4: Whenever a read-only transaction Q commits, the conflict between Q and active read-only transaction Q_a that belongs to the same group as Q will be checked. If Q has read some data objects in NRS(Q_a) and Q_a has read some data objects in NRS(Q), Q_a should be aborted. If Q has read some data objects in NRS(Q_a) only, the data objects in NRS(Q) are inherited to the NRS(Q_a). Moreover, the data objects that Q has read are added to CRS(Q_a).

```
i.e. Whenever a ROT Q of the same group as Qa commits, do
        if RS(Q)∩NRS(Qa) ≠ φ  then {
          if NRS(Q) ∩ RS(Qa) ≠ φ  then {
            abort Qa;
          }
          NRS(Qa):= NRS(Qa)∪NRS(Q);
          CRS(Qa):= CRS(Qa)∪RS(Q);
        }
```

The rule 4 serves to find any cycle when the DP-graph of Q and Q_a are combined together. If Q has read some data objects in NRS(Q_a), a path from Q_a to Q is found in the serialization graph. If Q has read some data objects in NRS(Q_a) and Q_a has read some data objects in NRS(Q), it means there is a path from Q_a to Q and from Q to Q_a. Obviously, there is a cycle in the serialization graph after combining Q and Q_a. In this case, Q_a should be aborted.

The NRS-Algorithm for Group Strong Consistency

```
        RS(Qa):= φ ;
        do while Qa is active {
          Whenever an update transaction Uc commits, do
          {
(Rule 3)    if WS(Uc)∩CRS(Qa)≠φ or RS(Uc)∩NRS(Qa)≠φ then
              CRS(Qa):= CRS(Qa)∪RS(Uc);
          for each x ∈ WS(Uc)∩CRS(Qa)   do
            NRS(Qa):= NRS(Qa)∪{x};
          if RS(Uc)∩NRS(Qa) ≠ φ  then
            for each y ∈[WS(Uc)\CRS(Qa)]   do
              NRS(Qa):= NRS(Qa)∪{y};
        }
```

(Rule 4)

```
Whenever a ROT Q of same group as Q_a commits, do
  if RS(Q)∩NRS(Q_a) ≠ φ  then {
    if NRS(Q) ∩ RS(Q_a) ≠ φ  then {
         abort Q_a;
    }
    NRS(Q_a):= NRS(Q_a)∪NRS(Q);
    CRS(Q_a):= CRS(Q_a)∪RS(Q);
  }
}
```

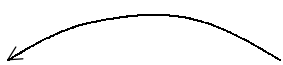
Abort Rule: Q_a aborts when it tries to read a data object in NRS(Q_a).

Example 5.1 (Cont'd)

Q_1: r(x) r(y) Q_2: r(y) r(z)

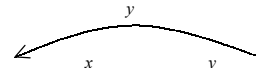
U_3: r(x) r(z) w(x) U_4: r(z) w(z) U_5: r(y) w(y) U_6: r(y) w(y)

S_5: $r_1(x)$ $r_2(y)$ $r_3(x)$ $r_3(z)$ $w_3(x)$ c_3 $r_4(z)$ $w_4(z)$ c_4 $r_5(y)$ $w_5(y)$ c_5 $r_6(y)$ $w_6(y)$ c_6 $r_2(z)$ c_2 $r_1(y)c_1$



$Q_1 \rightarrow U_3 \rightarrow U_4 \rightarrow Q_2 \rightarrow U_5 \rightarrow U_6$

Serialization Graph of S_5



$Q_1 \rightarrow, U_3 \ U_5 \rightarrow, U_6$

DP-Graph of Q_1

$U_4 \rightarrow, Q_2 \rightarrow, U_5 \rightarrow, U_6$

DP-Graph of Q_2

The following table shows the development of NRS(Q_1) and NRS(Q_2), using the algorithm for group strong consistency.

operation	NRS(Q_1)	NRS(Q_2)
$r_1(x)$	ϕ	ϕ
$r_2(y)$	ϕ	ϕ
c_3	{ x }	ϕ
c_4	{ x, z }	ϕ
c_5	{ x, z }	{ y }
c_6	{ x, z }	{ y }
$r_2(z)$	{ x, z }	{ y }
c_2	{ x, y, z }	OK
$r_1(y)$	{ x, y, z }	
c_1	aborts	

(Q_1 and Q_2 belong to the same group.)

Result: Q_2 commits but Q_1 aborts.

Explanation: When Q_1 executes r(x), NRS(Q_1) = ϕ and hence Q_1 successfully reads x. When Q_2 executes r(y), NRS(Q_2) = ϕ and Q_2 reads y successfully too. When U_3 successfully commits, z is firstly added to CRS(Q_1) since WS(U_3) $\cap$ CRS(Q_1) $\neq \phi$ and $z \in$ RS(U_3). After then, x is added to NRS(Q_1) since $x \in$ WS(U_3) $\cap$ CRS(Q_1). When U_4 commits, z is added to NRS(Q_1) since $z \in$ WS(U_4) $\cap$ CRS(Q_1). Similarly, when U_5 commits, y is added to NRS(Q_2) since $y \in$ WS(U_5) $\cap$ CRS(Q_2). Finally, when Q_2 executes r(z), $z \notin$ NRS(Q_2) and hence Q_2 reads z successfully and then

commits. Since Q_1 and Q_2 belongs to the same group and $z \in \text{RS}(Q_2) \cap \text{NRS}(Q_1)$, $y \in \text{NRS}(Q_2)$ is inherited to $\text{NRS}(Q_1)$ when Q_2 commits. Consequently, when Q_1 executes $\text{r}(y)$, $y \in \text{NRS}(Q_1)$ and hence Q_1 aborts.

6 Performance Evaluation

The simulation experiments are aimed at studying the performance of our proposed algorithm, Group Strong Consistency (GS), and the WoundCertifier [10] in broadcast disk environments. For WoundCertifier (WC), a certification report is broadcasted periodically. An active transaction is aborted if the readset is overlapped with certification report's writeset. When a transaction is ready to commit, client sends a request-to-commit message to the server and the server validates the request with other committed transactions.

We choose WC for comparison because it performs partial validation check at client side. This is a favorable feature for mobile transaction since the mobile client can detect the data conflict without consulting the server. We believe that the comparison is fair as both algorithms have the ability to detect data conflict in the early stage at the client side. We do not consider the effects of caching in the performance study. In other words, a ROT may have to wait for the requested data object in the next broadcast cycle if the data object is missed (broadcast) in the current broadcast cycle.

The performance of these algorithms is compared by miss rate, which is the percentage of transactions missing their deadline. Other performance metrics including restart rate, response time and throughput are also collected. In order to have a better understanding on the efficiency of the algorithms, we collect local-restart rate which is the percentage of transaction restarts caused by partial validation checking at mobile clients.

6.1 Experimental Setup

The simulation model consists of a server, 300 mobile clients and a broadcast disk for transmitting both the data objects and the required control information. It is implemented using the simulation tool OPNET [16]. These 300 mobile clients are assumed belonging to the same group. The mobile clients only process ROTs while the server processes update transactions as well. For mobile clients, the transactions are submitted one after another. After a transaction has committed, there is an inter-transaction delay before next transaction is submitted.

We consider a soft real-time database system in broadcast environments where a ROT is processed until it is committed even though the deadline is missed. The deadline of a ROT is assigned to be (current time + slack factor × predicted execution time) where predicted execution time is a function of transaction length, mean inter-operation delay and broadcast cycle length. The data objects accessed by a transaction are uniformly distributed in the database containing 300 data objects. A relatively small database size is used to model the hotspot effect of a real database system. It helps to intensify data conflicts between transactions at server and mobile clients.

A range of server transaction arrival rate is used to simulate different level of server loading. Thus, we can study the performance of the algorithms under different degree of data contention. The server transaction arrival rate is the number of server transaction issued per bit-time at the server. For example, 4e-6 means 1 server transaction per 250,000 bit-times. Table 1 lists the baseline setting for the simulation experiments. The time unit is in bit-time, the time to transmit a single bit. For a broadcast bandwidth of 64Kbps, 1 M bit-times is equivalent to approximately 15 seconds and the mean inter-operation delay and the mean inter-transaction delay is 1 second and 2 seconds respectively.

Table 1. Baseline Setting

Parameter	Value
Mobile Clients	
Number of Mobile Client	300
Transaction Length (Number of read operations)	6
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)
Concurrency Control Protocol	Group Strong Consistency
Server	
Number of Server	1
Transaction Length (Number of operations)	8
Write Operation Probability	0.6
Number of Data Objects in Database	300
Size of Data Objects	8,000 bits
Concurrency Control Protocol	Optimistic Concurrency Control with Forward Validation
Priority Scheduling	Earliest Deadline First

6.2 Performance Results

Figure 1 gives the miss rate of ROTs issued by mobile clients when the server transaction arrival rate varies. The figure indicates that the miss rate of both algorithms increases when the server transaction arrival rate increases. It also shows that the performance of GS is comparatively better than WC. Although WC is able to detect data conflict in the read phase, the validation algorithm does not allow high degree of concurrency. The mobile transactions suffer from excessive restart when there is a high server loading. It can be observed that GS improves the system performance more than WC. The separate algorithm for ROTs allows higher degree of concurrency than WC. In other words, GS allows some transaction schedules which cannot be produced by WC.

Figure 2 gives the restart rate of the transactions. It can help to investigate the effectiveness of the algorithms. Despite both algorithms perform similar in light server loading, we note that WC is sensitive to the server transaction arrival rate. The performance of WC drops sharply when the server is in high loading. Since WC aims to detect data conflicts by comparing with the certification report, it will restart the transaction if any possible data conflict is detected. In case of GC, a ROT will not be restarted as far as the data read by the ROT are consistent. Thus, GS can avoid some unnecessary restarts and accept more schedules than WC.

A similar trend is observed in response time (Figure 3). The response time of WC increases relatively more sharply when the server transaction arrival rate increases. It can be easily noted that when the number of restart is high, there will be a longer response time. Even if WC is able to detect data conflicts and restart transactions locally, it does not help to improve the response time due to the high number of restart. Conflict resolution appears to be more efficient in GS, a better performance is observed under different server transaction arrival rate. The response time of both algorithms is consistent with the number of restart.

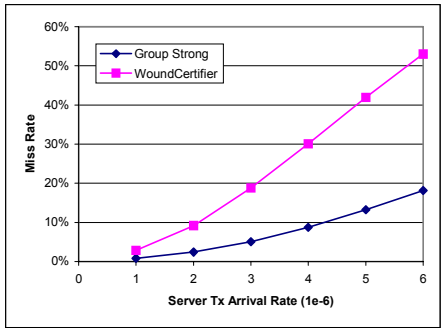


Figure 1: Miss Rate

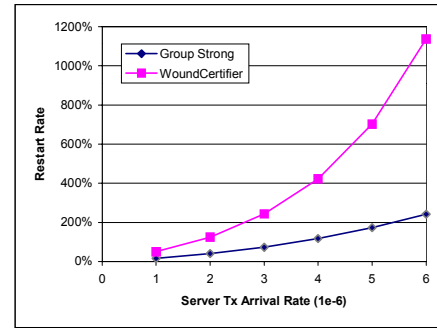


Figure 2: Restart Rate

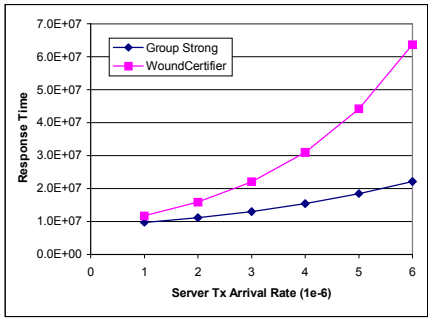


Figure 3: Response Time

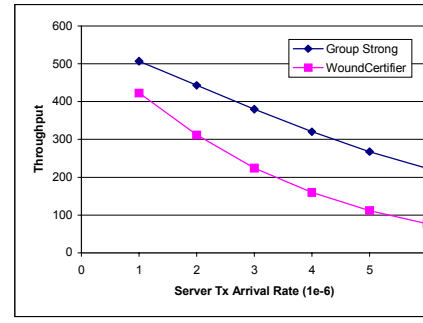


Figure 4: Throughput

Figure 4 gives the throughput of the algorithms. GS consistently performs better than WC. It can be explained in three ways. Firstly, both algorithms use the same amount of time to read data objects. The major factor is the efficiency of algorithm in resolving data conflict. From the view of restart rate, GS gets a lower restart rate than WC. A lower restart rate results in using shorter time to process a successfully-

committed transaction. It is easily to reason that GS has a higher throughput than WC. Secondly, the restart rate of WC increases sharply when the arrival rate of server transaction is high. It is reasonable to notice that the throughput of WC drops more deeply than GS. On the contrary, the restart rate of GS is increased proportionally with the server transaction arrival rate. Thus, the throughput of GS drops proportionally. The final factor is ability to detect irresolvable conflict as soon as possible. We will explain with the aid of local-restart rate.

The local-restart rate is used to measure the ability of the algorithms to detect data conflicts at the mobile client and restart the transaction locally. We believe that it is a favorable feature of mobile client in transaction performance. Since a late restart will prolong a transaction lifetime, it is better to detect data conflicts at client side and restart immediately. Thus, a high rate of local-restart is preferred when processing mobile transactions. Figure 5 records the percentage of restart happened at mobile client out of total number of restart. The general trend of GS shows a better performance than WC. GS almost detects all data conflicts at the client side. WC indicates a notable gain when the server loading is increased. However, it does not mean that it will have a great performance gain since WC suffers from excessive restart in high rate of data conflict. On the whole, GS performs better than WC.

Finally, we show the throughput of GS and WC with different mobile transaction lengths in Figure 6. It can be observed that the performance difference between the two algorithms decreases as mobile transaction length increases. It implies that it is less likely for a long ROT to read consistent data. On the whole, the results of this series of simulation experiments show that GC outperforms WC under different server loadings and transaction lengths.

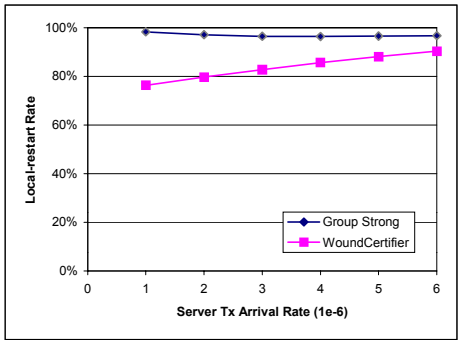


Figure 5: Local-restart Rate

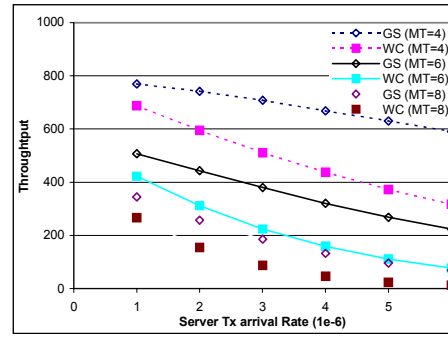


Figure 6: Effect of Mobile Transaction Length

7 Conclusion

In this paper, we discuss the issues of transaction processing in mobile environment. We examine the conditions which a read-only transaction (ROT) reads inconsistent data. Data-passing graph is devised as a tool to check whether a ROT has read inconsistent data. We first suggest NRS-algorithm to process ROT separate from the update transactions and prevent ROT from reading inconsistent data by maintaining the No-Read-Set at the client. Second, Group Strong Consistency is proposed to

restrict ROTs within a group to perceive the same serialization order of update transactions, as required by the serializability. Partial validation is performed at the client side to detect data conflicts at an early stage and restart the transaction locally.

In our simulation experiments, a significant performance improvement of Group Strong Consistency is showed when comparing with WoundCertifier. Besides, we investigate the advantages of applying partial validation at the mobile client. It can be concluded that using separate processing for ROTs is a viable approach to transaction processing in Bdisk environment. Thus, the system can scale up with more clients without weakening the performance.

References

1. S. Acharya, R. Alonso, M. Franklin and S. Zdonik, "Broadcast Disks: Data Management for Asymmetric Communications Environments", *Proceedings of the ACM SIGMOD Conference*, California, pp. 199-210, May 1995.
2. S. Acharya, M. Franklin and S. Zdonik, "Disseminating Updates on Broadcast Disks", *Proceedings of 22nd VLDB Conference*, Mumbai(Bombay), India, pp. 354-365, 1996.
3. E. Pitoura, B. Bhargava, "Maintaining Consistency of Data in Mobile Distributed Environments", *Proceedings of the 15 th International Conference on Distributing Computer System*, Vancouver, British Columbia, Canada, pp. 404-413, 1995.
4. E. Pitoura, "Supporting Read-Only Transactions in Wireless Broadcasting", *Proceedings of the DEXA98 International Wordshop on Mobility in Databases and Distributed Systems*, pp. 428-433, 1998.
5. E. Pitoura, "Scalable Invalidation-Based Processing of Queries in Broadcast Push Delivery", *Proceedings of the Mobile Data Access Workshop, in coopeation with the 17th International Conference on Conceptual Modeling*, pp. 230-241, 1999.
6. E. Pitoura, and P. K. Chrysanthis, "Scalable Processing of Read-Only Transactions in Broadcast Push," *Proceedings of the 19th IEEE International Conference on Distributed Computing System*, pp. 432-439, 1999.
7. P.A. Bernstein, V. Hadzilacos, N. Goodman, "Concurrency Control and Recovery in Database Systems", Addison-Wesley, Reading, Massachusetts.
8. D. Barbara, "Mobile Computing and Databases – A Survey", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 11, No. 1, pp. 108-117, 1999.
9. M.S. Chen, P.S. Yu, K.L. Wu, "Indexed Sequential Data Broadcasting in Wireless Mobile Computing", *17th International Conference on Distributed Computing Systems*, pp. 124-131, 1997.
10. D. Barbara, "Certification Reports: Supporting Transactions in Wireless Systems", *Proc. 17th International Conference on Distributed Computing Systems*, Baltimore, pp. 466-473, May 1997.
11. J. R. Haritsa, M. J. Carey, M. Livny, "Data Access Scheduling in Firm Real-time System", *Real-Time Systems*, Vol. 4, No. 3, pp. 203-241, 1992.
12. J. Huang, J. A. Stankovic, K. Ramamritham, D. Towley, "Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes", *Proceedings of the 17th International Conference on Very Large Data Bases*, pp. 35-46,1991.
13. M. Franklin, S. Zdonik, "Dissemination-Based Information Systems", *IEEE Data Engineering Bulletin*, Vol.19, No.3, pp.: 20-30, September, 1996.
14. M. Franklin, "Concurrency Control and Recovery", *Handbook of Computer Science and Engineering,* A. Tucker, ed., CRC Press, Boca Raton, 1997.

15. H. T. Kung, John T. Robinson, “On Optimistic Methods for Concurrency Control”, *ACM Transactions on Database System,* Vol. 6, No. 2, pp. 213-226, June 1981.
16. OPNET Modeler / Radio 6.0 (c), MIL3, Inc., 1987-1999.
17. H. Garcia-Molina, G. Wiederhold, “Read-only transactions in a distributed database”, *ACM Transactions on Database Systems*, Vol. 7, No. 2, pp. 209-234, June 1982.