Real-time Transaction Processing Using Two-stage Validation in Broadcast Disks*

Kwok-wa Lam[1], Victor C. S. Lee[1], and Sang H. Son[2]

[1] Department of Computer Science, City University of Hong Kong
csvlee@cityu.edu.hk
[2] Department of Computer Science, University of Virginia
son@cs.virginia.edu

Abstract. Conventional concurrency control protocols are inapplicable in mobile computing environments due to a number of constraints of wireless communications. In this paper, we design a protocol for processing mobile transactions. The protocol fits the environments such that no synchronization is necessary among the mobile clients. Data conflicts can be detected between transactions at the server and mobile transactions by a two-stage validation mechanism. In addition to relieving the server from excessive validation workload, no to-be-restarted mobile transactions will be submitted to the server for final validation. Such early data conflict detection can save processing and communication resources. Moreover, the protocol allows more schedules of transaction executions such that unnecessary transaction aborts can be avoided. These desirable features help mobile transactions to meet their deadlines by removing any avoidable delays due to the asymmetric property.

1 Introduction

Broadcast-based data dissemination becomes a widely accepted approach to communication in mobile computing environments [1], [2], [6], [8]. 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. 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] introduced the concept of Broadcast Disks (Bdisks), which 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.

Although there is a number of related research work in Bdisks environments [1], [3], [4], only a few of them support transactional semantics [5], [8]. In this

* Supported in part by Direct Allocation Grant no. 7100094 from City University of Hong Kong

paper, we propose a new approach to processing transactions in Bdisks environments where updates can be originated from the clients. In the new protocol, mobile clients share a part of the validation function with the server and are able to detect data conflicts earlier such that transactions are more likely to meet their deadlines [9].

2 Issues of Transaction Processing in Broadcast Environments

Since the bandwidth from mobile clients to server is limited, concurrency control protocols, which require continuous synchronization with the server to detect data conflicts during transaction execution 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. The eventual termination (commit or abort) of mobile transactions submitted to the server will be broadcast to the clients in the following 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 Bdisks environments. First, any "serious" conflicts which leads 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 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 [7].

3 Protocol Design

We assume a central data server that stores and manages all the data objects at the server. Updates submitted from the clients are subject to final 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 private workspace.

3.1 Broadcasting of Validation Information

We use the timestamp intervals of transactions to reduce the unnecessary aborts and exploit 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 is not supposed to 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. Thus, all transactions have to be submitted to the server for final validation. 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. One important issue is that the server and the clients should avoid repeating same part of the validation function. In other words, they should complement each other.

3.2 Timestamp Ordering

We assume that there are no blind write operations. For each data object, a read timestamp (RTS) and a write timestamp (WTS) are maintained. The values of $RTS(x)$ and $WTS(x)$ represent the timestamps of the youngest committed transactions that have read and written data object x respectively. 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 dependency 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, $TS(T_a)$, selecting 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. 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's examine the implication of data conflict resolution between a committed transaction, T_c, and an active transaction, T_a, on the dependency in the serialization order. 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) $RS(T_c) \cap WS(T_a) \neq \{\}$ (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) $WS(T_c) \cap RS(T_a) \neq \{\}$ (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 though 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 The New Protocol

4.1 Transaction Processing at Mobile 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 three functions are described by the algorithms **Process**, **Validate**, and **Submit** as shown below and the validation information consists of the following components.

- The *Accepted* and *Rejected* sets contain the identifiers of transactions successfully validated or rejected at the server in the last broadcast cycle.
- The $CT\_ReadSet$ and $CT\_WriteSet$ contain data objects that are in the read set and the write set of those committed transactions in the *Accepted* set.
- The $RTS(x)$, a read timestamp and, $FWTS(x)$ and $WTS(x)$, the first and the last write timestamps in the last broadcast cycle, 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.

Functions: Process, Validate, and Submit at the Clients

Process (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);
      $TOR(T_a, x) := WTS(x)$;
      $Final\_Validate(T_a) := Final\_Validate(T_a) \cup \{x\}$; }
  }
  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);
      remove x from $Final\_Validate(T_a)$; }
  }
}

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 ∈ CT_WriteSet and x ∈ CWS(T_a) then
      abort T_a;
    if x ∈ CT_WriteSet and x ∈ Final_Validate(T_a) then
    { TI(T_a) := TI(T_a) ∩ [0, FWTS(x)];
      if TI(T_a) = [] then abort T_a;
      else remove x from Final_Validate(T_a); }
    if x ∈ CT_ReadSet and x ∈ CWS(T_a) then
    { TI(T_a) := TI(T_a) ∩ [RTS(x), ∞);
      if TI(T_a) = [] then abort T_a; }
  }
}

Submit (T_a)
{ Submitted := Submitted ∪ {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),
      Final_Validate(T_a), TOR(T_a, x)
  // x of TOR(T_a, x) ∈ (WS(T_a) ∪ Final_Validate(T_a));
}
```

4.2 The Server Functionality

The server continuously performs the following algorithm until it is time to broadcast the next cycle. In essence, 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 clients to determine whether the execution of transactions is globally serializable. Note that the server does not need to perform the validation for those read operations of the validating transactions that have already done at the clients. 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.

Validation at the Server
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 $Final\_Validated(T_v)$
  { 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 $TI(T_v) + \epsilon$ // ϵ is a sufficient small value
  for each x in $RS(T_v)$
    if $TS(T_v) > RTS(x)$ then
      $RTS(x) := TS(T_v)$;
  for each x in $WS(T_v)$
    $WTS(x) := TS(T_v)$;
  $Accepted := Accepted \cup \{T_v\}$;
  $CT\_WriteSet := CT\_WriteSet \cup WS(T_v)$;
  $CT\_ReadSet := CT\_ReadSet \cup \{RS(T_v) - WS(T_v)\}$;
}

5 Conclusions and Future Work

In this paper, we first discuss 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 transactions meeting their deadlines.

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 clients 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 of wireless communication. These are very desirable features in real-time applications where transactions are associated with timing constraints.

References

1. Acharya S., M. Franklin and S. Zdonik, "Disseminating Updates on Broadcast Disks," Proc. of 22nd VLDB Conference, India, 1996.
2. Acharya S., R. Alonso, M. Franklin and S. Zdonik, "Broadcast Disks: Data Management for Asymmetric Communication Environments," Proc. of the ACM SIGMOD Conference, U.S.A., 1995.
3. Baruah S. and A. Bestavros, "Pinwheel Scheduling for Fault-Tolerant Broadcast Disks in Real-Time Database Systems," Technical Report TR-1996-023, Computer Science Department, Boston University, 1996.
4. Bestavros A., "AIDA-Based Real-Time Fault-Tolerant Broadcast Disks," Proc. of the IEEE Real-Time Technology and Applications Symposium, U.S.A. 1996.
5. Herman G., G. Gopal, K. C. Lee and A. Weinreb, "The Datacycle Architecture for Very High Throughput Database Systems," Proc. of the ACM SIGMOD Conference, U.S.A. 1987.
6. Imielinski T and B. R. Badrinath, "Mobile Wireless Computing: Challenges in Data Management," Communication of the ACM, vol. 37, no. 10, 1994.
7. Lam K. W., K. Y. Lam and S. L. Hung, "Real-time Optimistic Concurrency Control Protocol with Dynamic Adjustment of Serialization Order," Proc. of the IEEE Real-Time Technology and Applications Symposium, pp. 174-179, Illinois, 1995.
8. Shanmugasundaram J., A. Nithrakashyap, R. Sivasankaran, K. Ramamritham, "Efficient Concurrency Control for Bdisks environments," ACM SIGMOD International Conference on Management of Data, 1999.
9. Stankovic, J. A., Son, S. H., and Hansson, J., "Misconceptions about Real-Time Databases," Computer, vol. 32, no. 6, pp. 29-37, 1999.