Efficient Concurrency Control for Broadcast Environments

*Jayavel Shanmugasundaram** *Arvind Nithrakashyap* *Rajendran Sivasankaran*
Krithi Ramamritham†
University of Massachusetts, Amherst
jai@cs.wisc.edu, {nithraka, sivasank, krithi}@cs.umass.edu

Abstract

A crucial consideration in environments where data is broadcast to clients is the low bandwidth available for clients to communicate with servers. Advanced applications in such environments do need to read data that is mutually consistent as well as current. However, given the asymmetric communication capabilities and the needs of clients in mobile environments, traditional serializability-based approaches are too restrictive, unnecessary, and impractical. We thus propose the use of a weaker correctness criterion called *update consistency* and outline mechanisms based on this criterion that ensure (1) the *mutual consistency* of data maintained by the server and read by clients, and (2) the *currency* of data read by clients. Using these mechanisms, clients can obtain data that is current and mutually consistent "off the air", i.e., without contacting the server to, say, obtain locks. Experimental results show a substantial reduction in response times as compared to existing (serializability-based) approaches. A further attractive feature of the approach is that if caching is possible at a client, weaker forms of currency can be obtained while still satisfying the mutual consistency of data.

1 Introduction

Many emerging database applications, especially those with numerous concurrent clients, demand the broadcast mode for data dissemination. For example, in electronic commerce applications, such as auctions, it is expected that a typical auction might bring together millions of interested parties even though only a small fraction may actually offer bids. Updates based on the bids made must be disseminated promptly and consistently. Fortunately, the relatively small size of the database, i.e., the current state of the auction, makes broadcasting feasible. But, the communication bandwidth available for a client to communicate with servers is likely to be quite restricted. Thus, an attractive approach is to use the broadcast medium to transmit the current state of the auction while allowing the clients to communicate their updates (to the current state of the auction) using low bandwidth uplinks to the servers. The problem addressed in this paper and the techniques outlined are motivated by such applications. In particular, we are concerned with the problem of providing readers with *current* and *mutually consistent* data while ensuring the consistency of updates.

Broadcast-based data dissemination is also likely to be a major mode of information transfer in mobile computing and wireless environments [Imi94, Ach95]. Many such systems have been proposed [She94, Oki93] and commercial systems such as Vitria [Vit] already support broadcasting. As these systems evolve, they will be used to run sophisticated applications, many of which will involve data whose consistency must be maintained in spite of updates, some of which may originate from mobile clients. Other applications of broadcasting, include stock trading, next generation road traffic management systems and automated industrial plants. Given the limited amount of bandwidth available for clients to communicate with the broadcast server, achieving data consistency efficiently is a challenging research issue.

[Her87] and [Ach96] are among the few papers motivated by similar considerations. Herman et. al. [Her87] discuss transactional support in the Datacycle architecture, which is also an asymmetric bandwidth environment. However, they use serializability as the correctness criterion, which we show is very expensive, restrictive, and unnecessary in such environments. In [Ach96], the authors discuss the tradeoffs between currency of data and performance issues when some of the broadcast data items are updated by processes running on the server. However, 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 on the server, while the

*Currently at the University of Wisconsin-Madison

†Also affiliated with Indian Institute of Technology, Bombay

processes on clients are assumed to be read-only. There has been some very recent concurrent related work [Pit99]. One of their approaches (based on serialization graph testing) is similar in functionality to ours, but it requires clients to be listening continuously to the broadcast. This approach, unlike ours, is thus intolerant to communication failures.

In this paper, we propose and evaluate protocols appropriate for broadcast environments. The protocols satisfy two properties: (1) *mutual consistency* and (2) *currency*. Mutual consistency ensures that (a) the server maintains mutually consistent data and (b) clients can read mutually consistent data. Currency ensures that clients see data that is current (as of the beginning of a broadcast cycle). We ensure mutual consistency by adapting a correctness criterion called *update consistency* in [Bob92] and external consistency in [Wei87][1] for transaction processing in broadcast environments. Our protocols, presented in the context of broadcast disks [Ach95], are intended for applications similar to the auction application, where the size of the database is relatively small but the number of clients is very large. These protocols permit read-only transactions running on mobile clients to always *read current and consistent values without contacting the server* (to acquire locks or to validate their reads), i.e., they are able to read current values "off-the-air". We also evaluate their performance and compare them with the algorithm used in *Datacycle*.

The rest of the paper is organized as follows. In section 2, we outline the characteristics of broadcast environments and demonstrate the inapplicability of existing concurrency control techniques. We then motivate the need for a consistency criterion weaker than serializability and show that update consistency is more appropriate. In section 3, we describe an efficent algorithm *APPROX* to check for update consistency and outline mechanisms to implement it in broadcast disk environments while also satisfying currency requirements. In section 4, we evaluate the performance of these mechanisms and in Section 5, we summarize the paper and outline future work.

2 Correctness in Broadcast Environments

In this section, we first outline the characteristics of broadcast environments and then describe why most existing concurrency control techniques are not applicable in broadcast environments. Finally, we identify the need to satisfy update consistency [Bob92, Wei87], a weakening of serializability, and currency of data seen by clients.

[1]For the rest of the paper, we refer to this correctness criterion as *update consistency*.

2.1 Characteristics of Broadcast Environments

In this section, we first describe broadcast disks, a particular type of broadcast environment and then use them to illustrate key characteristics of broadcast environments.

In the *broadcast disks* [Ach95] framework, a server periodically broadcasts all the data items in the database clients. The clients view this broadcast as a disk and can read the values of data items being broadcast and cache them locally. In order to write onto objects, the clients contact the server with the appropriate information. The size of the database being transmitted cannot be too large or intolerable delays may be experienced at the client waiting for a data item. Fortunately, for many applications like online auctions and traffic control, the number of data items being transmitted is not too large (of the order of hundreds, sometimes thousands, of objects).

In such environments, though the server to client bandwidth is relatively plentiful, the bandwidth from clients to the server is likely to be very limited because:

- The number of clients listening to a server could be of the order of millions, so a server cannot handle high bandwidth communication from all the clients.
- Battery power is a scarce resource for mobile clients. Since transmissions require substantial battery power (which is more than is needed for reception), transmissions from a client should be avoided, if possible .

Thus, techniques for concurrency control in broadcast environments must take this asymmetry in bandwidth into account. In the next section, we show that existing concurrency control techniques do not efficiently satisfy these requirements because they are not designed for such asymmetric communication environments.

2.2 Inapplicability of Serializability for Broadcast Environments

Serializability [Ber87] is the commonly accepted correctness criterion for transactions in database systems. It is, however, intrinsically a global property - the effect of concurrent transaction execution should be as though *all* the transactions executed in some serial order. It (a) requires excessive communication between the distributed entities, to obtain locks, for example, or (b) requires the underlying protocol to be overly conservative thus disallowing certain correct executions. The first alternative is expensive in broadcast environments because of the limited bandwidth that clients have available to communicate with the server. The second alternative leads to unnecessary transaction aborts, which again is undesirable. We now show that three fundamental techniques, at least one of which is used by virtually any proposed concurrency control mechanism to satisfy serializability in distributed/client-server environments, are inapplicable for broadcast environments.

- *Locking:* Many proposed concurrency control protocols [Car91, Wan91, Wil90, Fra93, Mol82] use locking even for read-only transactions. In broadcast environments, this would translate to acquiring read locks for every data item read by client transactions which would swamp the server with lock requests.
- *Cache Consistency Mechanisms:* These protocols [Car91, Wan91, Wil90, Fra93] predominantly assume that the server is aware of the data items cached at the clients so that changes to data items can be invalidated/propagated to clients. Clearly, these techniques would not be applicable in broadcast environments because (a) the server has to keep track of the caches of a large number of clients and (b) the clients would have to inform the server every time a data item is read, leading to high overhead even for read-only transactions. [Guk96] considers using old versions of data at the clients, which, in addition, compromises the currency of data items read.
- *Timestamp Mechanism:* Some mechanisms proposed for distributed systems [Wei87] use timestamp based concurrency control protocols that require an object to keep track of both read and write timestamps. This is infeasible in broadcast environments because this would require client to server communication for every read.

To the best of our knowledge, the Datacycle approach [Her87] is the only concurrency control technique proposed in the literature aimed at broadcast environments. The Datacycle approach ensures that all transactions executing at clients and the server are globally serializable. But, as we just argued, it would still lead to poor performance because serializability is very expensive to achieve in broadcast environments. Experimental results presented in Section 4 support this argument.

If we would like to avoid the (substantial) communication costs incurred by the interactions with the server – needed to ensure serializability – then, as we show now, clients have to be conservative, leading to unnecessary aborts.

Example 1. Assume that in a broadcast environment, clients only know the local transaction execution history and the history of updates at the server. Consider stock trading transactions t_1 and t_3 at two different clients A and B respectively that read the stock prices of IBM and Sun. Also let t_2 and t_4 be transactions at the server that update the prices of IBM and Sun respectively. Now consider the following execution history:

$$\mathtt{r_1(IBM)w_2(IBM)c_2r_3(IBM)r_3(Sun)w_4(Sun)c_4r_1(Sun)} \quad (1)$$

If transactions running on clients do not inform the server about the operations performed by them – a reasonable assumption given the limited uplink bandwidth from clients to the server – then the server would only be aware of the history of its own operations:

$\mathtt{w_2(IBM)\ c_2\ w_4(Sun)\ c_4}$

If a server broadcasts this history along with the data, client A would be aware of the history:

$\mathtt{r_1(IBM)\ w_2(IBM)\ c_2\ w_4(Sun)\ c_4\ r_1(Sun)}$

and Client B would be aware of the history:

$\mathtt{w_2(IBM)\ c_2\ r_3(IBM)\ r_3(Sun)\ w_4(Sun)\ c_4}$

If both t_1 and t_3 commit, then the server and both the clients see serializable histories (the serialization orders are $t_2;t_4$, $t_4;t_1;t_2$ and $t_2;t_3;t_4$ respectively). However, the global history is not serializable. Thus, either t_1 or t_3 must be aborted. However, since the read operations performed by a client transaction are not communicated to other clients or the server, and assuming that there exists no way to inform clients t_1 and t_3 except by expensive message passing, *both* t_1 and t_3 would have to be aborted – since each client must assume the worst-case history at the other. This is wasteful since the abortion of either t_1 or t_3 would have ensured a serializable history. Unnecessary aborts would occur even if the system history is

$$\mathtt{r_1(IBM)w_2(IBM)c_2w_4(Sun)c_4r_1(Sun)} \quad (2)$$

because Client A would be aware of this history and would not be able to distinguish it from (1). In this case too, t_1 must be aborted. A similar argument can be made for t_3. Essentially, in the absence of communication from read-only transactions, to preserve serializability, the read-only transactions will have to be aborted even in cases like history (2) because clients would have to assume worst case scenarios as in history (1).

Example 2. Consider the following history that is a modification of the history used in Example 1 (it has the additional operation $w_1(DEC)$, so that t_1 is not read-only any more, and a commit operation for transaction t_3). Again assume that transactions t_1 and t_3 are executed at two different clients and that transactions t_2 and t_4 are executed at the server.

$$\mathtt{r_1(IBM)w_2(IBM)c_2r_3(IBM)r_3(Sun)c_3w_4(Sun)c_4r_1(Sun)w_1(DEC)} \quad (3)$$

Let us assume that transaction t_1 now desires to commit. At the commit time, the server would be aware of the following history.

$$\mathtt{r_1(IBM)w_2(IBM)c_2w_4(Sun)c_4r_1(Sun)w_1(DEC)} \quad (4)$$

This is because the server needs to be aware of the reads and writes of all update transactions, whether they originate at the client or the server, in order to ensure consistency. In the absence of high client to server bandwidth, however, the server would not know about any read-only transactions. The sub history (4) is serializable when t_1 commits even though the entire history (3) is not. The implication of this example is that, even if the actual history of execution is like (4), update transactions such as t_1 would not be allowed to commit under serializability because a worst case scenario, such as (3), has to be assumed. Clearly this is wasteful since it causes unnecessary transaction aborts.

The above two examples illustrate that, in broadcast environments, serializability is very expensive to achieve, both for read-only and update transactions. This is because of the unnecessary aborts serializability might induce or because of the excessive communication it entails if such aborts are to be avoided.

In summary, we can see that serializability fails to be an appropriate correctness criterion because of two main reasons: (1) As exemplified by Example 1, all read-only transactions executing at possibly different clients are required to see the same serial order of update transactions (2) As exemplified by Example 2, read-only transactions are required to be serializable with respect to all the update transactions, even those whose updates do not affect the values read by read-only transactions. What seems to be required (and, as we shall show next, to be sufficient) is a correctness criterion that relaxes these notions.

2.3 In Search of Appropriate Correctness Criteria

There have been many weakened notions of serializability proposed in the literature for various applications [Ram96]. A correctness criterion proposed in the context of multiversion concurrency control, called update consistency in [Bob92] and external consistency in [Wei87], appears ideal for use in broadcast environments [2]. In this correctness criterion, a history is said to be consistent iff both of the following conditions hold (for a more formal definition, see [Bob93] or [Sha99]):

- All update transactions are serializable.
- Each read-only transaction is serializable with respect to the subset of update transactions it (directly or indirectly) reads from.

Though weaker than serializability, these conditions maintain consistency of the database and of the values read by transactions. In the rest of this section, we give the intuition behind these conditions using the examples introduced in the previous section and comment on the applicability of consistency based on these conditions to broadcast environments.

Consider the history from Example 1 in Section 2.2. As mentioned earlier, if transactions t_1 and t_3 were to commit, then the history would not be serializable. However, the history would still be acceptable because the sub history involving only the update transactions t_2 and t_4 is serializable (the serialization order could be either $t_2; t_4$ or $t_4; t_2$). Further, each read-only transaction is serializable with respect to all the update transactions. For read-only transaction t_1, the serialization order is $t_4; t_1; t_2$ while for read-only transaction t_3, the serialization order is $t_2; t_3; t_4$.

Even though the history is not serializable, each transaction still reads consistent data because of update consistency. For instance, read-only transaction t_1 sees a consistent state of the database - the committed state corresponding to a transactional update to Sun's stock. Similarly, read-only transaction t_3 sees the update to IBM's stock. Update transactions, being serializable, also see and produce consistent database states. Thus, consistency is not compromised even though the two read-only transactions see different serial orders.

Consider now the history from Example 2 in Section 2.2. If transaction t_1 were allowed to commit, this history would not be serializable. However, the history would still be acceptable because all update transactions are serializable (the serialization order is $t_4; t_1; t_2$) and the read-only transaction t_3 is serializable with respect to the update transaction t_2 (the serialization order is $t_2; t_3$).

The fact that the histories in Examples 1 and 2 are acceptable implies that read-only transactions need not ever contact the server. This is because the server does not require this information to perform the validation of transactions while still maintaining (update) consistency. We thus see that achieving the update consistency of data at the server, on the one hand, and of data read by a client, on the other hand, addresses the problems with serializability as discussed in Section 2.2.

The only issue about update consistency that might seem to be of potential concern is the fact that transactions executing at the same client can see different serial orders of execution of update transactions. There are two cases to be considered here. In the first case, concurrently executing read-only transactions at a client see different serial orders of update transactions (consider Example 1 where t_1 and t_3 execute at the same client). This, however, does not lead to any inconsistencies because the transactions are allowed to see different orders of updates precisely because the updates are unrelated. Furthermore, because the mechanisms proposed to implement mutual consistency also satisfy the currency requirement, they ensure that if a read-only transaction (say t_i) starts executing after the completion of another read-only transaction (say t_j) at a client, then t_i and t_j see the serial ordering of transactions they both depend on (directly or indirectly) in a

[2] However, the concurrency control techniques outlined in [Bob92] and [Wei87], are not applicable to broadcast environments for reasons outlined in Section 2.1.

consistent fashion. Thus, two transactions, one of which is executed based on the results of another, will not see inconsistent histories.

The second case is when read-only transactions and update transactions see different serial orders of execution (all update transactions themselves are globally serializable). As explained in the previous case, it is acceptable for independent concurrently executing transactions to see different serial orders (consider Example 2 where t_1 and t_3 execute at the same client). In the case that the concurrently executing transactions are not independent, i.e., a read-only transaction reads from an update transaction, however, update consistency ensures that the read-only transaction and the update transaction see the same serial ordering. Thus, the problem arises only when (a) a read-only transaction starts executing after the completion of an update transaction or (b) an update transaction starts executing after the completion of an read-only transaction. The mechanisms that we use to enforce update consistency, however, satisfy the currency requirement and hence ensure that the two transactions see consistent serial orderings of the transactions they both depend on. Thus, update consistency along with the implementation proposed in this paper do not result in the two related transactions executing at the same client seeing different serial orders of execution of update transactions.

3 Mechanisms to Guarantee Correctness

We now outline *mechanisms* that ensure that (a) transactions are update consistent and (b) the data values read by transactions are current.

It can be shown (see [Sha99]) that even if all the update transactions are *serially* (not to be confused with serializably) executed, it is still NP-Complete to determine whether a history is update consistent. Thus, there probably does not exist an efficient way to determine whether a history is update consistent, when using virtually any serializability based concurrency control algorithm for update transactions. We hence use a polynomial time approximation algorithm, *APPROX*, to efficiently determine legal histories. A mechanism to implement this algorithm in broadcast disk environments, *F-Matrix*, is also described. We then propose, *R-Matrix*, a simpler (in terms of space and time) version of *F-Matrix*. Finally, we outline how *F-Matrix* and *R-Matrix* can be extended to exploit weak currency requirements by using client caching techniques.

3.1 A Simple Approximation Algorithm

Given the NP-completeness of determining update consistency, we now adapt a polynomial time algorithm [Bob92] that accepts a set of histories that is a proper subset of update consistent histories. The fact that only a proper subset of update consistent histories is accepted implies that the algorithm accepts only update consistent histories though some update consistent histories may not be accepted.

The following concepts are useful in defining the approximation algorithm. Let t be a transaction which executes in a history $\mathcal{H}$. Then, the *set of live transactions with respect to t in the history* $\mathcal{H}$, $LIVE_{\mathcal{H}}(t)$, is the minimal set closed under the following two rules: (a) t is in $LIVE_{\mathcal{H}}(t)$ and (b) If t' is in $LIVE_{\mathcal{H}}(t)$, then all transactions t'' such that t' reads the value of an object written by t'' in $\mathcal{H}$ are also in $LIVE_{\mathcal{H}}(t)$. Intuitively, the set of live transactions with respect to a transaction t is the set of transactions that t directly or indirectly reads from. The *update sub history* of a history $\mathcal{H}$, $\mathcal{H}_{update}$, is a projection of the history $\mathcal{H}$ which includes all and only the operations performed by transactions that perform a write operation in $\mathcal{H}$.

The approximation algorithm *APPROX* determines that a history is legal iff both of the following conditions hold: (1) $\mathcal{H}_{update}$ is conflict serializable. (2) For every read only transaction t_R in the history $\mathcal{H}$, $S_{\mathcal{H}}(t_R)$ is acyclic. Here $S_{\mathcal{H}}(t_R)$ is the serialization graph consisting of only the transactions in $LIVE_{\mathcal{H}}(t_R)$[3].

The intuition behind this algorithm is to replace *all* occurrences of view serializability [Ber87, Pap88] in the formal characterization of update consistency (see [Sha99])[4] with occurrences of conflict serializability. Since conflict serializability is an efficient alternative to view serializability, we expect the algorithm *APPROX* to be efficient. Indeed, it can be shown that *APPROX* is a polynomial time algorithm [Sha99].

3.2 Implementing APPROX for Broadcast Disk Environments

In this section, we describe an implementation of *APPROX*, namely *F-Matrix* (short for *Full Matrix*), that is appropriate for broadcast disk environments. We then propose a simpler algorithm that approximates *APPROX* but which is much more space efficient. Finally, we qualitatively compare the algorithms with the Datacycle concurrency control algorithm [Her87].

3.2.1 The *F-Matrix* Implementation

We now outline the server functionality, the client functionality, the nature of the control information transmitted from the server to the clients, the client read-only transaction validation protocol and the details on how the control information is computed at the server.

Server Functionality

The server performs the following functions:

[3] For a more precise definition of $S_{\mathcal{H}}(t_R)$, see [Sha99].

[4] The formal characterization of update consistency in [Bob93] is not in terms of view serializability and is thus less general than our characterization.

1. During each cycle, broadcasts the latest *committed* values of all data items at the beginning of the cycle. Note that this implies that the server has to maintain two versions of objects: the latest committed version and the last written version[5].
2. Ensures the conflict serializability of all transactions submitted to the server (some of these may originate at the clients and be submitted for validation as described later). The exact information a client must provide along with its update transaction is discussed under client functionality. We do assume that if a transaction commits, then all transactions from which that transaction reads have previously committed.
3. Transmits a *control matrix* during each cycle that helps clients determine whether read-only transactions read consistent values. The control matrix will be described shortly.

Client Functionality

Clients handle two types of transactions: read-only transactions and update transactions. Read, write, commit and abort operations performed by a transaction are handled as follows:

- *Read Operation:* Before a read operation is performed on a data item broadcast during a cycle, the control information transmitted during that cycle is consulted to determine whether the read operation can proceed (the exact details about the nature of this check will be described shortly). If the read operation cannot proceed the transaction is aborted.
- *Write Operation:* When a data item is written, the write is performed on a local copy of the data item in the client. No checks are made.
- *Commit:* If the transaction has not performed any write operation, then the commit operation does not have to do anything and the commit succeeds. In case the transaction has performed a write operation, a list of all the objects written and the values written are sent to the server. In addition, the list of all read operations performed and the cycle numbers in which they are performed are sent to the server. The server checks to see whether the update transaction can be committed and communicates the result to the client. If so, the transaction is committed, else it is aborted. This method of handling update transactions is similar to the method proposed in [Kum97].
- *Abort:* If the transaction has not performed any write operation, then the abort does nothing. In case the transaction has written to a data item, then all the copies of the data items written to are discarded and further execution of the transaction is stopped.

[5]The maintenance of two versions of objects has some commonality with multiversion concurrency control [Ber87]. Our concern here is about *clients* which maintain only a single version.

Nature of Control Information

We now describe the nature of the control information transmitted by the server and show how it is updated at the end of each cycle. The control information matrix at any point in time is an $n \times n$ matrix, C, where n is the number of objects. If objects are assumed to be have ids ob_1 through ob_n, each entry $C(i,j)$ is set to a cycle number determined as follows.

Let $\mathcal{H}$ be the history of execution of the committed update transactions at the server. Also, let t_j be the last committed update transaction that wrote ob_j. We assume that a transaction t_0 writes all data items at cycle 0 (before the beginning of the broadcast). Then:

- $C(i,j) = \max_{t' \in LIVE_{\mathcal{H}}(t_j) \ \wedge \ \text{t' writes } ob_i} (c_{t'})$ where $c_{t'}$ = cycle number in which t' committed.

As defined earlier, $LIVE_{\mathcal{H}}(t_j)$ refers to the set of transactions (including t_j) that t_j directly or indirectly reads from. Thus, $LIVE_{\mathcal{H}}(t_j)$ is the set of transactions that "affect" the latest committed value of ob_j (because t_j writes the latest committed value of ob_j). $C(i,j)$ is thus the latest cycle number in which some transaction that affects the latest committed value of ob_j and also writes to ob_i, commits. The following example illustrates how the entries in the C matrix are determined.

Example 4: Consider the following history:

$w_1(ob_1)w_1(ob_2)c_1r_2(ob_1)w_2(ob_1)c_2r_3(ob_2)w_3(ob_2)c_3$

and assume that the commit operation c_i occurs during the i^{th} broadcast cycle .

In the above scenario $C(1,1) = 2$ because t_2 was the last transaction to write onto ob_1 (thus "affecting" the value of ob_1) and it committed during cycle 2 of the broadcast. For similar reasons, $C(2,2) = 3$. The value of $C(1,2) = 1$. This is because t_3 was the last committed transaction to write onto ob_2 and $LIVE_{\mathcal{H}}(t_3) = \{t_1, t_3\}$ and t_1 is the only transaction in $LIVE_{\mathcal{H}}(t_3)$ to write onto ob_1 and it does so during cycle 1 of the broadcast. For similar reasons, $C(2,1) = 1$.

So far, we have assumed that each entry of the C matrix has to store the cycle number relative to the first cycle ever broadcast. This could be avoided if we know the maximum number of cycles that a transaction could span (max_cycles). In that case, we need to store only cycle numbers from 0 to max_cycles and perform modulo $max\_cycles + 1$ arithmetic and comparisons. This would reduce the size of each entry in the C matrix. The issue of how the control information is to be transmitted during a broadcast cycle also needs to be addressed. One effective way to partition the C matrix is to broadcast the j^{th} column right after broadcasting the object ob_j. This would minimize the amount of time the client would have to wait for control information, as will become apparent in the next section.

Validating Client Reads

For a read-only transaction t at the client, the following protocol is followed before each read operation. Let R_t be the set of $(ob_i, cycle)$ pairs such that transaction t previously read object ob_i from broadcast cycle $cycle$, i.e., t read the latest committed value of ob_i at the beginning of cycle $cycle$. Then a read operation on an object ob_j is allowed to proceed in a cycle iff the following condition, *read-condition*(ob_j), holds (thereby preserving conflict serializability):

$$\forall (ob_i, cycle) \in R_t \quad (C(i,j) < cycle)$$

where C is the matrix at the beginning of the current cycle. If the *read-condition* fails, the transaction is aborted. Intuitively, a read operation performed by a transaction t on object ob_j is allowed iff no transaction in $LIVE(t)$ wrote onto ob_i after t read from the ob_i.

Theorem 1 A read-only transaction t_R is allowed to commit by the protocol described above iff $S(t_R)$ is acyclic. (See [Sha99] for proof of this theorem.)

Computing Control Information

As a transaction commits, the entries affected by it are updated by the server. For simplicity, we consider only the simple case where the entries are updated as per a serialization order (see Section 5 for other option).

We now show how the matrix C can be incrementally constructed. At the beginning of the broadcast, each entry in C is set to the cycle number 0. Let C_{old} be the matrix that considers all the committed transactions in a serial order up to a certain point in time in a broadcast cycle c_1. Now, let t be a newly committed transaction that occurs immediately after all previously committed transactions in the update transaction serialization order. Also, let $c_2 \geq c_1$ be the cycle in which t commits. Let RS_t be the set of objects read by t and let WS_t be the set of objects written by t. Then the new matrix C_{new} is computed as follows:

1. $C_{new}(i,j) = c_2$ if $ob_i, ob_j \in WS_t$

   Since transaction t writes to objects ob_i, ob_j and it is the last transaction to write onto ob_j in the serial order of execution, t is the last transaction that affects the latest value of ob_j and also writes to ob_i. Thus, the entry is set to the cycle in which t commits.

2. $C_{new}(i,j) = \max_{ob_k \in RS_t}(C_{old}(i,k))$ if $ob_i \notin WS_t$ and $ob_j \in WS_t$

   Since transaction t reads from objects in RS_t, the transactions t depends on is the set of all transaction that directly or indirectly affected the latest values of items written to the objects in RS_t. Thus, each entry is updated to reflect this fact. If the transaction t does not read any object, then this C entry is set to 0.

3. $C_{new}(i,j) = C_{old}(i,j)$ otherwise

   If an object ob_j is not updated, entries in column j are untouched.

Theorem 2 The above incremental algorithm preserves the semantics of the C matrix. (See [Sha99] for proof of this theorem.)

There are a few potential drawbacks of *F-Matrix*. One is that computing the matrix may be expensive in terms of server time. We, however, do not expect this problem to be severe because broadcast environments cater to read-mostly workloads and read-only transactions are not involved in matrix construction. The other potential drawback of the *F-Matrix* mechanism is that the bandwidth required to transmit the C matrix during each broadcast cycle. The space requirements for the C matrix in *F-Matrix* is $n^2 \times \log(max\_cycles)$ bits per broadcast cycle, where n is the number of objects in the database and max_cycles is the maximum number of broadcast cycles spanned by a transaction. This cost can be significant for large values of n. Quantitatively, if the size of each object is s bits, then the fraction $\frac{n^2 \times \log(max\_cycles)}{n^2 \times \log(max\_cycles) + n \times s} = \frac{n \times \log(max\_cycles)}{n \times \log(max\_cycles) + s}$ of the broadcast cycle time is spent on control information. If s is small, this overhead can be significant. The overhead may not be significant for large values of s. This problem is actually slightly worse than it seems. It can be shown that regardless of the compression technique used, in the worst case, the number of bits to be transmitted in order to represent the C matrix during each broadcast cycle is quadratic in the number of objects broadcast [Sha99].

This result, however, assumes for simplicity that the entire C matrix is transmitted in each broadcast cycle. The number of bits to be transmitted may be drastically reduced if we transmit only changes (deltas) over the previous C matrix transmission. This approach, however, has the disadvantage that the client has to store the previous transmitted C matrix. More importantly, the client should listen to the last broadcast of the C matrix and the subsequent deltas regardless of whether it wanted to read any data in those broadcast cycles, thus increasing the usage of scarce client resources (like battery power). We do not investigate the details of this approach in detail in this paper but plan to study it as part of future work.

3.2.2 *R-Matrix* and *Datacycle* Implementations

In the previous section, we described the *F-Matrix* mechanism to implement the *APPROX* algorithm in broadcast disk environments and also outlined some of the potential overheads of computing and transmitting the C matrix. In order to get around the drawbacks of *F-Matrix* mentioned above, *F-Matrix* could be modified as follows. Instead of viewing each object as a separate entity, we can partition them into groups of objects. The C matrix will now not be a $n \times n$ matrix, where n is the number of database objects, but would be a $n \times g$ matrix, where g is the number of groups in the database, thus reducing the size of the control information. Further, the fact that there are fewer entries to update would make it more efficient to update the matrix. Each entry of the modified C matrix, $MC(i, s)$, where s is a set of objects, is defined by:

- $MC(i, s) = \max_{ob_j \in s}(C(i, j))$

For the purposes of the matrix, we consider all database items in the set s to be indistinguishable. At the client, the check is performed as in the *F-Matrix* case, with the following modification to account for the grouping of objects. Let R_t be the set of $(ob_i, cycle)$s such that transaction t previously read object ob_i in cycle *cycle*. Then *read-condition(ob_j)* becomes:

$\forall (ob_i, cycle) \in R_t, \quad MC(i, s_j) < cycle)$

where MC is the matrix at the beginning of the current cycle and s_j is the database partition to which ob_j belongs. Since the number of partitions is tunable, we have a spectrum of mechanisms implemented by varying the size of the database partitions (in the extreme case, where all database partitions are singleton sets, we have the implementation *F-Matrix*). There is a tradeoff in choosing the size of the partition: increasing the partition size would mean that certain unnecessary conflicts would result, while reducing the partition size would imply increased control information overhead. In this paper, we concentrate on the two extreme cases to illustrate this basic tradeoff: (a) all database partitions are singleton sets (*F-Matrix*) and (b) all database items fall under one partition.

In case (b), the matrix entry associated with each object is just the latest cycle number in which a committed value was written to it (i.e., the matrix is in fact just a vector). Thus, *read-condition(ob_j)* is simplified to:

$\forall (ob_i, cycle) \in R_t \quad (MC(i, db) < cycle)$

where db is the set of all the objects in the database. In other words, a transaction can proceed to read an object only if *no* previously read value has been updated. This corresponds to the Datacycle approach for concurrency control [Her87] and ensures serializability. For the rest of this paper, we call this approach *Datacycle*. The *Datacycle* approach, however, does not fully realize the potential of the matrix reduction. We can in fact weaken *read-condition(ob_j)* for a read on an object ob_j performed by a transaction that read the first data item at cycle c_1 to:

$\forall (ob_i, cycle) \in R_t$
$(MC(i, db) < cycle) \quad \vee \quad (MC(j, db) < c_1)$

where db is the set of all the objects in the database. We call this modified algorithm *R-Matrix* (for reduced matrix). The idea behind this algorithm is that a transaction reads consistent values if either (a) no values it has previously read have been overwritten by other transactions or (b) (even though previously read values may have been overwritten by other transactions) the current value being read has not been overwritten since the beginning of the transaction. (a) ensures that the transaction sees the database state at the time of the last read while (b) ensures that the transaction sees the database state at the time of the first read. It can be shown (see [Sha99]) that *R-Matrix* accepts only schedules accepted by *APPROX*.

Optimizations similar to those done in *Datacycle* could also be done for *R-Matrix*. Thus, a bit could be set by hardware if any of the previously read values of a transaction are changed. For a future read of an object, if the bit is set and if the object being read has been changed during or after the cycle in which the first read operation was performed by the transaction, the transaction is aborted. Otherwise, the read operation is allowed to proceed. Besides reducing the number of unnecessary aborts of read-only transactions, this mechanism has a nice "stability" property unlike *Datacycle* where a transaction may be aborted even if it does not perform any further reads.

3.3 Weak Currency Requirements

In the discussion above, we have assumed that the read operations of client transactions would like to see data that is current to at least the beginning of the broadcast cycle in which the read operation was performed. Thus, each read operation had to read data items off the broadcast. This is not necessary, however, if the client currency requirement is such that data items read have to be current to only within T time units, where T could be much larger than the broadcast cycle time. In this case, data items read off the broadcast disk could be cached at clients so that future accesses to the same data items can just be read from the local cache[6] without having to wait for the data item to appear again in the broadcast. A data item is removed from the cache as soon as the currency of the data item cached exceeds T time units. Note that this decision to invalidate items in the cache can be done locally by the client without need for any communication.

Different clients may have different currency requirements and even for a given client, there may be different currency requirements for different data items. Since the invalidation of the cache at clients is purely local, the invalidation interval can be tailored on a per client, per object basis and the invalidation performed accordingly. Thus, clients with vastly different currency requirements can coexist in a broadcast medium with-

[6] In case the client cache is not large enough to hold all the date items needed by transactions, cache replacement policies similar to those suggested in [Ach96] could be employed.

out any need for extra communication. This caching technique is a specific instance of a more general technique called quasi-caching, introduced in [Alo90].

The main problem that is be solved in this context is to ensure that transactions see mutually consistent data even when they read cached data items. Previous approaches for maintaining the consistency of cached data items [Alo90, Ach96] do not ensure the mutual consistency of objects read by transactions. We achieve this by storing the columns of the matrix corresponding to the data items cached, along with the cycle at which the data items were cached. A transaction that accesses cached data items has all the information necessary for validation when using *F-Matrix* or *R-Matrix*.

In summary, weak currency requirements can be effectively exploited in broadcast environments using client caching. Our mechanisms can be extended to maintain inter-object consistency by storing the relevant columns of the concurrency control matrix.

4 Simulation Results

Our simulation based experiments are aimed at comparing the performance of four different algorithms (*Datacycle*, *F-Matrix*, *R-Matrix* and *F-Matrix-No*) for concurrency control of client read-only transactions in broadcast disk environments. The first three algorithms are described in the previous section. The last algorithm is used as a baseline and implements the functionality of *F-Matrix*, but ignores the cost of broadcasting the control information associated with each object in the database. Hence, the broadcast cycle lengths are shorter for runs of this algorithm as compared to those for *F-Matrix*. We do not consider the effects of caching in this performance study.

The performance of these algorithms can be compared relative to the following metrics:

- **Transaction Response Time** – the time the transaction is submitted by a client to the time the transaction commits. This includes the time involved in restarting the transaction (perhaps more than once, if necessary), if it aborts.
- **Transaction Restart Ratio** – the number of times a transaction is restarted because the data it read was inconsistent.

4.1 Experimental Setup

Our simulator consists of a server, a client, a broadcast disk structure and an event queue. Only read-only transactions execute on the client. The server executes update transactions. Only one client was simulated because the performance of the concurrency control mechanisms for read-only transactions is independent of the number of clients. The objects that the transactions access are determined using a uniform distribution on the objects in the database. All these objects are updatable and hence correspond to the "hot spots" of the broadcast database. Table 1 lists the simulation parameters. Client transaction length indicates the number of read operations performed at the client and server transaction length indicates the number of read/write operations at the server. The delays are modeled as exponential distributions. For a broadcast medium with bandwidth of 64 Kbits/s, the inter-operation delay translates to 1 second and the client inter-transaction delay translates to 2 seconds.

The server fills the broadcast disk with data at the beginning of a cycle. To *simulate* the database objects being broadcast at different times, a time entry exists for each object. This is set to the time at which the object is actually broadcast. The client waits until this time to read the corresponding object. Each cycle consists of a broadcast of all the objects along with the associated control information. For *F-Matrix*, the control information consists of an $(n \times n)$ matrix, where n is the number of objects. Each column of the matrix is broadcast along with the corresponding object. For *R-Matrix* and *Datacycle*, the control information consists of an array of length n. Each element in the array is broadcast along with the corresponding object.

The time to broadcast one bit (one bit-unit) is used as the unit of time. All response times are measured in terms of this unit. We evaluate the algorithms with respect to different settings of the first 5 parameters, varying a parameter at a time. Each experiment consists of 1000 client transactions and the simulations run until all of them commit. The data was derived from the last 500 transactions to ensure that it was "steady-state" data. In these experiments, 95% confidence intervals were obtained, whose widths were less than 10% of the point estimates.

For *F-Matrix* the fraction of the cycle used for the control information (in bits) is given by $\frac{n^2 \times TS}{n^2 \times TS + n \times OBJ}$ $= \frac{n \times TS}{n \times TS + OBJ}$, where TS is the size of a timestamp and OBJ is the size of an object. For a timestamp size of 8 bits and an object size of one kilo Byte and 300 objects in the database, the overheads work out to about 23%. For *R-Matrix* and *Datacycle*, the fraction of cycle time used for control information is $\frac{n \times TS}{n \times TS + n \times OBJ}$ $= \frac{TS}{TS + OBJ}$. For the same parameters, the overheads in this case are only about 0.1%.

4.2 Effect of Client Transaction Length

Until client transaction length 4, all the algorithms have similar performance (see Figures 1(a) and 1(b)). After that, especially beyond a value of 6, the *Datacycle* algorithm performs very poorly so much so that for a length of 10, its response time was outside the limits of the Y-axis. Even though *R-Matrix* is far better than *Datacycle*, *F-Matrix* shows even better behavior. For a client transaction of 8, *R-Matrix* has a response time

Parameter	Default Value
Client Transaction Length	4
Server Transaction Length	8
Transaction Rate at Server	1 in 2.5×10^5 bit-units
Number of Objects in Database	300
Size of Objects in Database	1 KB
Server Read Operation Probability	0.5
Client Inter-Operation Delay	65536 bit-units
Client Inter-Transaction Delay	131072 bit-units
Client Restart Delay	0 bit-units
Timestamp Size	8 bits

Table 1: Parameters for the Simulation

of 122.68×10^6 while *F-Matrix* has a response time of 14.6×10^6 units (about 12% of *R-Matrix*'s response time). This is because *F-Matrix* reduces the number of client transaction aborts to almost zero (see Figure 1(b)) by using a weaker read condition. Also, note that the performance curve for *F-Matrix* is flat, indicating that it scales very well with client transaction length.

The small differences between the performance of *F-Matrix* and *F-Matrix-No* are magnified in Figure 3(a). These differences arise because the control information transmission time is ignored in *F-Matrix-No*. So, read transactions wait for a shorter time for the broadcast data to be available. The number of update transactions per cycle is also reduced and hence, there are lesser read conflicts in each cycle.

As can be seen from Figures 1(a) and 1(b), there is a high correlation between the response time and the number of aborts. Thus, for the rest of this paper, we concentrate on response times.

4.3 Effect of Server Transaction Length

Longer server transactions lead to more updates at the server for each cycle. Hence, the response time increases with server transaction length in Figure 2(a). However, *F-Matrix* shows very little increase in response time compared to *Datacycle* and even *R-Matrix*. Once again this demonstrates the scalability of *F-Matrix*.

4.4 Effect of Transaction Rate at Server

In Figure 2(b), the server transaction rate (X-axis) decreases as we go from left to right. This graph shows that, as expected, response time improves with a decrease in server transaction rate. The performance of *F-Matrix* nearly matches that of *F-Matrix-No* and is better than *R-Matrix* and much better than *Datacycle*. Also, *F-Matrix* displays almost no degradation in performance as server transaction rate increases.

4.5 Effect of Number of Objects

As the number of objects in the database increases, the probability of transactions accessing an object decreases. The length of the cycle, however, increases with the number of objects. The increase in the control information to be sent also increases the length of the broadcast cycle. This increases the number of server transactions per cycle and hence, increases the number of possible conflicts. Hence, the response times increase with the size of the database. As can be seen from Figure 3(a), the relative ordering of the four protocols remains the same with *F-Matrix* displaying the best response times (about 9.6×10^6 units for a database size of 400, as compared to nearly 11.3×10^6 for *R-Matrix*) among the three practical protocols. The rate of increase in response time, though almost constant for both *R-Matrix* and *F-Matrix*, is smaller for *F-Matrix*.

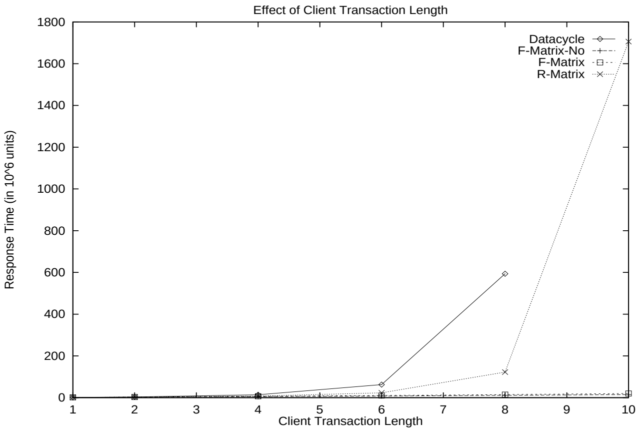
4.6 Effect of Object Size

The relative behavior of the algorithms with increasing size of objects is shown in Figure 3(b). The length of the broadcast cycle increases with object size and hence the response time also increases. Note that at small object sizes, *F-Matrix* and *R-Matrix* perform about the same because the space overhead of *F-Matrix* increases (relative to the size of the broadcast cycle) and offsets the associated gain in concurrency. At very small object sizes, *F-Matrix* would thus perform worse than *R-Matrix*. As object size increases, however, the effect of the control information tends to decrease. Hence, the performance of *F-Matrix* is better than that of *R-Matrix* and *Datacycle* and approaches that of *F-Matrix-No*. Here again, response time increases at a smaller rate for *F-Matrix* than for the other two practical protocols.

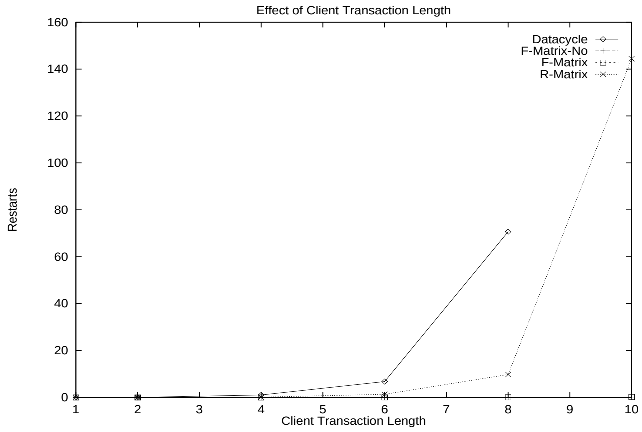
4.7 Summary of Results

R-Matrix outperforms *Datacycle* and *F-Matrix* outperforms *R-Matrix* in all the experiments. Thus, the experimental results confirm the hypothesis that a weaker abort condition would lead to better response times. Furthermore, *F-Matrix* is highly *scalable* with respect to client transaction length, server transaction length, and server transaction rate. All the protocols display increased response times with increase in broadcast cycle lengths, but the rate of increase in the case of *F-Matrix* is lower than that of *R-Matrix* as well as *Datacycle*. These performance features of *F-Matrix* are observed in spite of *F-Matrix* requiring more bits to transmit the control information in each cycle. Also, in many cases its performance profile is very close to *F-Matrix-No*.

F-Matrix and *R-Matrix* represent two ends of a spectrum. *F-Matrix* has the overhead of transmitting control information of size $O(n^2)$, where n is the number of objects. *R-Matrix* presents a cheaper alternative, though it does not perform or scale as well as *F-Matrix*. In general, the choice of the algorithm to be used would depend on the capacity of the broadcast medium as well as the profile (number of objects, frequency of updates, etc.) of the database that is to be used.

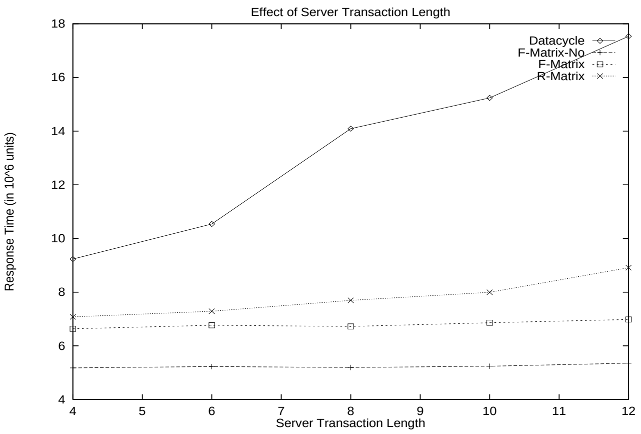


(a) Comparison of Response Times

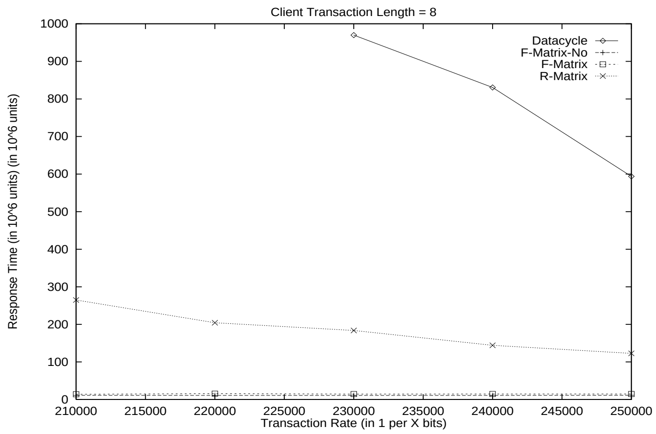


(b) Comparison of Number of Aborts

Figure 1: Effect of Client Transaction Length

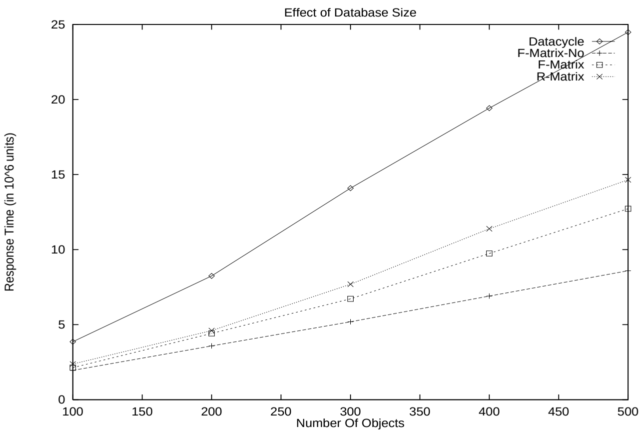


(a) Effect of Server Transaction Length

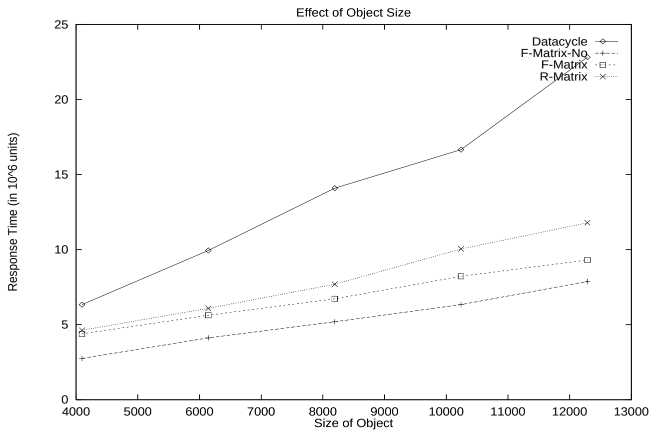


(b) Effect of Server Transaction Rate

Figure 2: Effect on Response Time



(a) Effect of Number of Objects



(b) Effect of Object Size

Figure 3: Effect on Response Time

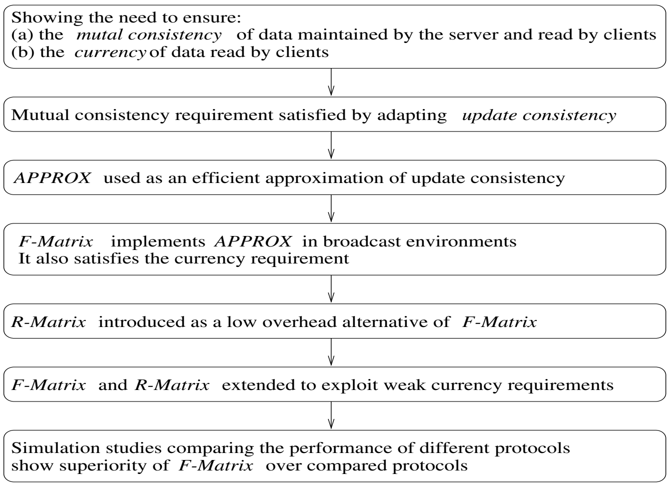


Figure 4: Development of Ideas

5 Conclusions and Future Work

There are several possible applications of broadcast based database systems (such as stock trading and traffic information) that impose consistency and currency requirements [Xua97]. Catering to such requirements through efficient concurrency control is the subject of this paper. Figure 4 shows the development of ideas.

In the future, we plan to study efficient parallel computation and incremental transmission of the control matrix, performance in the presence of caching, and extensions to optimize for client update transactions.

References

[Ach95] S. Acharya, et. al. Broadcast Disks: Data Management for Asymmetric Communications Environments. *Proceedings of the ACM SIGMOD Conference*, California, May 1995.

[Ach96] S. Acharya, M. Franklin and S. Zdonik. Disseminating Updates on Broadcast Disks *Proceedings of the VLDB Conference*, Mumbai(Bombay), India, 1996.

[Alo90] R. Alonso, D. Barbara and H. Garcia-Molina. Data Caching Issues in an Information Retrieval System. *ACM Transactions on Database Systems*, 15(3), September 1990.

[Ber87] P. Bernstein, V. Hadzilacos and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Massachusetts, 1987.

[Bob92] P.M. Bober and M.J. Carey. Multiversion Query Locking. *Proceedings of the VLDB Conference*, Vancouver, Canada, August 1992.

[Bob93] P.M. Bober and M.J. Carey. Multiversion Query Locking. Computer Science Technical Report TR 1160, University of Wisconsin-Madison, 1993.

[Car91] M.J. Carey, et. al. Data Caching Tradeoffs in Client-Server DBMS Architectures. *Proceedings of the ACM SIGMOD Conference*, Denver, June 1991.

[Fra93] M.J. Franklin. Caching and Memory Management in Client-Server Database Systems. Ph.D. Thesis, University of Wisconsin-Madison, 1993.

[Guk96] S. Gukal, E. Omiecinski and U. Ramachandran. Transient Versioning for Consistency and Concurrency in Client-Server Systems. *Proceedings of the Conference on Parallel and Distributed Information Systems (PDIS)*, Florida, December 1996.

[Her87] G. Herman, et. al. The Datacycle Architecture for Very High Throughput Database Systems *Proceedings of the ACM SIGMOD Conference*, 1987.

[Imi94] T. Imielinski and B.R. Badrinath. Mobile Wireless Computing: Challenges in Data Management. *Communications of the ACM*, 37(10), 1994.

[Kum97] S. Kumar, E. Kwang and D. Agrawal. Caprera: An Activity Framework for Transaction Processing on Wide-Area Networks *Proceedings of the VLDB Conference*, Athens, Greece, August 1997.

[Mol82] H. Garcia-Molina and G. Wiederhold. Read-Only Transactions in a Distributed Database. *ACM Transactions on Database Systems*, 7(2), 1982.

[Oki93] B. Oki, et. al. The Information Bus - An Architecture for Extensible Distributed Systems. *Proceedings of the SOSP Conference*, North Carolina, December 1993.

[Pap88] C.H. Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, 1988.

[Pit99] E. Pitoura and P. Chrysanthis, Scalable Processing of Read-Only Transactions in Broadcast Push, *IEEE International Conference on Distributed Computing Systems*, Austin, 1999

[Ram96] K. Ramamritham and P. Chrysanthis. A Taxonomy of Correctness Criteria in Database Applications. *VLDB Journal*, 5(1), January 1996.

[Sha99] J. Shanmugasundaram, et. al. Efficient Concurrency Control for Broadcast Environments Univ. of Massachusetts Technical Report 1999.

[She94] S. Shekar and D. Liu. Genesis and Advanced Traveler Information Systems (ATIS): Killer Applications for Mobile Computing. MOBIDATA Workshop, New Jersey, 1994.

[Vit] White Paper, http://www.vitria.com.

[Wan91] W. Wang and L. Rowe. Cache Consistency and Concurrency Control in a Client/Server DBMS Architecture. *Proceedings of the ACM SIGMOD Conference*, June 1991.

[Wei87] W. Weihl. Distributed Version Management for Read-Only Actions. *IEEE Transactions on Software Engineering*, 13(1), January 1987.

[Wil90] W. Wilkinson and M.A. Nemat. Maintaining Consistency of Client Cached Data. *Proceedings of the VLDB Conference*, Australia, August 1990.

[Xua97] P. Xuan, et. al. Broadcast on Demand - Efficient and Timely Dissemination of Data in Mobile Environments. *IEEE Real-Time Technology and Applications Symposium*, June 1997, pp. 38-48.