Scalable Processing of Read-Only Transactions in Broadcast Push*

Evaggelia Pitoura
Department of Computer Science
University of Ioannina
GR 45110 Ioannina, Greece
email: pitoura@cs.uoi.gr

Panos K. Chrysanthis
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
email: panos@cs.pitt.edu

Abstract

Recently, push-based delivery has attracted considerable attention as a means of disseminating information to large client populations in both wired and wireless settings. In this paper, we address the problem of ensuring the consistency and currency of client read-only transactions in the presence of updates. To this end, additional control information is broadcasted along with data. A suite of methods is proposed that vary in the complexity and volume of the control information transmitted and subsequently differ in response times, degrees of concurrency, and space and processing overheads. The proposed methods are combined with caching to improve query latency. The relative advantages of each method are demonstrated through both simulation results and qualitative arguments. Read-only transactions are processed locally at the client without contacting the server and thus the proposed approaches are scalable, i.e., their performance is independent of the number of clients.

1 Introduction

In traditional client/server systems, data are delivered on demand. A client explicitly requests data items from the server. Upon receipt of a data request, the server locates the information of interest and returns it to the client. This form of data delivery is called *pull-based*. In wireless computing, the stationary server machines are often provided with a relative high-bandwidth channel which supports broadcast delivery to all mobile clients located inside the geographical region it covers. This facility provides the infrastructure for a new form of data delivery called *push-based* delivery. This broadcast infrastructure can also be found in wired networks. In push-based data delivery, the server repetitively broadcasts data to a client population without a specific request. Clients monitor the broadcast and retrieve the data items they need as they arrive on the broadcast channel.

Push-based delivery is important for a wide range of applications that involve dissemination of information to a large number of clients. Dissemination-based applications include information feeds such as stock quotes and sport tickets, electronic newsletters, mailing lists,

*University of Ioannina, Computer Science Department, Technical Report No: 98-026

road traffic management systems, and cable TV. Important are also electronic commerce applications such as auctions or electronic tendering. Finally, information dissemination on the Internet has gained significant attention (e.g., [8, 22]). Many commercial products have been developed that provide wireless dissemination of Internet-available information. For instance, the AirMedia's Live Internet broadcast network [3] wirelessly broadcasts customized news and information to subscribers equipped with a receiver antenna connected to their personal computer. Similarly, Hughes Network Systems' DirectPC [20] network downloads content directly from web servers on the Internet to a satellite network and then to the subscribers' personal computer.

The concept of broadcast data delivery is not new. Early work has been conducted in the area of Teletext and Videotex systems [4, 21]. Previous work also includes the Datacycle project [9] at Bellcore and the Boston Community Information System (BCIS) [13]. In Datacycle, a database circulates on a high bandwidth network (140 Mbps). Users query the database by filtering relevant information via a special massively parallel transceiver. BCIS broadcasts news and information over an FM channel to clients with personal computers equipped with radio receivers. Recently, broadcast has received considerable attention in the area of wireless computing because of the physical support for broadcast in both satellite and cellular networks.

In this paper, we address the problem of preserving the consistency of clients' read-only transactions, when the values of data that are being broadcasted are updated at the server. Providing transactional support tailored to read-only transactions is important for many reasons. First, the great majority of transactions in dissemination systems are read-only. Then, even if we allow update transactions at the client, it is more efficient to process read-only transactions with special algorithms. That is because consistency of queries can be ensured without contacting the server. This is important because even if a backchannel exists from the client to the server, this channel typically has small communication capacity. Furthermore, since the number of clients supported is large, there is a great chance of overwhelming the server with clients' requests. In addition, avoiding contacting the server decreases the latency of client transactions.

To this end, control information is broadcasted along with data that enables the validation of read-only transactions at the clients. We propose various methods that vary in the complexity and volume of control information, including transmitting invalidation reports, multiple versions per item, and serializability information. Caching at the client is also supported to decrease query latency. The performance of the methods is evaluated and compared through both qualitative arguments and simulation results. In all the methods proposed, consistency is preserved without contacting the server and thus the methods are scalable; i.e., their performance is independent of the number of clients. This property makes the methods appropriate for highly populated service areas. The methods are applicable in wired as well as in wireless settings.

The remainder of this paper is organized as follows. In Section 2, we introduce the problem of supporting consistent read-only transactions in the presence of updates. In Section 3, various methods for handling the problem are presented. The methods proposed are extended to support caching at the client in Section 4. In Section 5, the performance of the read-only transaction processing methods is compared through both qualitative arguments and simulation results. In Section 6, we review briefly related research. Finally, in Section

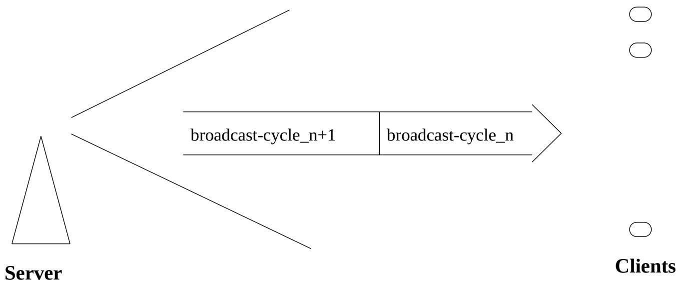


Figure 1: Broadcast-based data delivery

7, we offer conclusions and present our plans for future work.

2 The Model

The server periodically broadcasts data items to a large client population. Each period of the broadcast is called a broadcast *cycle* or *bcycle*, while the content of the broadcast is called a *bcast*. Each client listens to the broadcast and fetches data as they arrive (Figure 1). This way data can be accessed concurrently by any number of clients without any performance degradation. However, access to data is strictly sequential, since clients need to wait for the data of interest to appear on the channel. We assume that all updates are performed at the server and disseminated from there. Clients access data from the broadcast in a read-only mode. We do not make any particular assumptions on transaction processing, i.e., concurrency control or recovery, at the server.

2.1 Broadcast Organization

Clients do not need to listen to the broadcast continuously. Instead, they could tune-in to read specific items. Selective tuning is important especially in the case of portable mobile computers, since they most often rely for their operation on the finite energy provided by batteries and listening to the broadcast consumes energy. However, for selective tuning, clients must have some prior knowledge of the structure of the broadcast that they can utilize to determine when the item of interest appears on the channel. Alternatively, the broadcast can be self-descriptive, in that, some form of directory information is broadcasted along with data. In this case, the client first gets this information from the broadcast and use it in subsequent reads. Techniques for broadcasting index information along with data are given for example in [14, 10].

The smallest logical unit of a broadcast is called *bucket*. Buckets are the analog to blocks for disks. Each bucket has a header that includes useful information. The exact content of the bucket header depends on the specific broadcast organization. Information in the header usually includes the position of the bucket in the bcast as an offset from the beginning of the bcast as well as the offset to the beginning of the next bcast. The offset to the beginning of the next bcast can be used by the client to determine the beginning of the next bcast when the size of the broadcast is not fixed. Data items correspond to database records (tuples). We assume that users access data by specifying the value of one attribute of the record, the search key. Each bucket contains several items.

2.2 Consistency of Read-Only Transactions

We assume that the server broadcasts the content of a database. A database consists of a finite set of data items. A database state is typically defined as a mapping of every data to a value of its domain. Thus, a databases state, denoted DS, can be defined as a set of ordered pairs of data items in D and their values. In a database, data are related by a number of restrictions called integrity constraints that express relationships of values of data that a database state must satisfy. A database state is consistent if it does not violate the integrity constraints [7].

While data items are being broadcasted, transactions are executed at the server that may update the values of the items broadcasted. We assume that the contents of the broadcast at each cycle is guaranteed to be consistent. In particular, we assume that the values of data items that are broadcasted during each broadcast cycle correspond to the state of the database at the beginning of the broadcast cycle, i.e., the values produced by all transactions that have been committed by the beginning of the cycle. Thus, a read-only transaction that reads all its data within a single cycle can be executed without any concurrency overhead at all. We make this assumption for clarity of presentation, we later discuss how it can be eliminated.

Since the set of items read by a transaction is not known at static time and access to data is sequential, transactions may have to read data items from different bcasts, that is values from different database states. As a very simple example, say T be a transaction that corresponds to the following program:

```
if a > 0 then read b else read c
```

and that b and c precede a in the broadcast. Then, a client's transaction has to read a first and wait for the next cycle to read the value of b or c.

We define the *span* of a transaction T, $span(T)$, to be the maximum number of different broadcast cycles from which T reads data. The above example shows that the order in which transactions read data affects the response time of queries. A form of transaction optimization that orders requests for data based on the order according to which they are broadcasted can be employed to keep the transaction's span small.

Since client transactions read data from different cycles, there is no guarantee that the values they read are consistent. We define the *readset* of a transaction T, denoted $Read\_Set(T)$, to be the set of items it reads. In particular, $Read\_Set(T)$ is a set of ordered pairs of data items and their values that T read. Our correctness criterion for read-only transactions is that each transaction reads consistent data. Specifically, the readset of each read-only transaction must form a subset of a consistent database state [18]. We assume that each server transaction preserves database consistency. Thus, a state produced by a serializable execution (i.e., an execution equivalent to a serial one [7]) of a number of server transactions produces a consistent database state. The goal of the methods presented in this paper is to ensure that the readset of each read-only transaction corresponds to such a state.

3 Read-Only Transaction Processing Schemes

In this section, we introduce a suite of processing schemes for read-only transactions. The first method is based on broadcasting invalidation reports, the second one on broadcasting older versions and the last one on broadcasting serializability information.

3.1 The Invalidation-Only Method

Each bcast is preceded by an invalidation report in the form of a list that includes all data items that were updated at the server during the previous broadcast cycle. For each active read-only transaction R, the client keeps a set $RS(R)$ of all data items that R has read so far. At the beginning of each bcast, the client tunes in and reads the invalidation report. A read transaction R is aborted if an item $x \in RS(R)$ was updated, that is if x appears in the invalidation report. Clearly, in the absence of disconnections (we will discuss disconnections in a later section)

Theorem 1 *The invalidation only method produces correct read-only transactions.*

Proof. Let c_c be the cycle during which a committed read-only transaction R performed its last read operation and DS^{c_c} be the database state broadcasted at cycle c_c. Then, the values read by R correspond to the database state DS^{c_c}. For the purposes of contradiction, assume that a value of a data item x read by R corresponds to a database state broadcasted at a previous cycle, then an invalidation report should have been transmitted for x and thus R should have been aborted. □

As indicated by the proof above, in the invalidation-only method, a read-only transaction R reads the most current values, that is the values produced by all transactions committed by the beginning of the broadcast cycle at which R commits. The increase in the size of the broadcast is equal to $\lceil \frac{u\,k}{b} \rceil$, where u is the number of items that were updated, k is the size of the *key* and b the bucket size.

3.2 Multiversion Broadcast

In order to minimize the number of invalidated and aborted read-only transactions, older versions of data items may be retained temporarily. In particular, if for each data item, its S previous values, i.e., the values that the item had during the previous S bcycles, are available, where S is the maximum transaction span among all read-only transactions, then, read-only transactions can proceed safely by reading older versions of data when necessary. To implement the scheme, the server, instead of broadcasting the last committed value for each data item, maintains and broadcasts multiple versions for each data item. Multiversion schemes, where older copies of items are kept for concurrency control purposes, have been successfully used to speed-up processing of on-line read-only transactions in traditional pull-based systems (e.g., [16]).

At least one value (the current one) is broadcasted for each data item. At each bcycle k, the server discards the $k - S$ version from the bcast. An additional value is added to the broadcast, for those data items that were updated during the previous broadcast cycle. Let c_0 be the broadcast cycle during which a client transaction R performs its first read

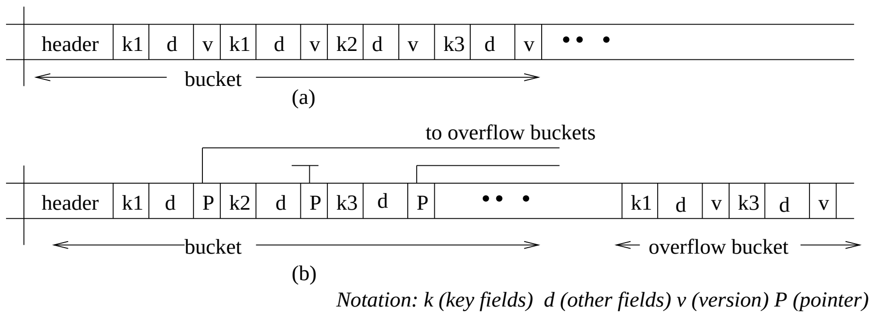


Figure 2: Multiversion broadcast with $S = 3$: (a) simple (b) with overflow buckets.

operation. During c_0, transaction R reads the most up-to-date value for each data item, that is, the value having the largest version number. In later cycles, R reads the value with the largest version number c_n, such that $c_n \leq c_0$.

Theorem 2 *The S-multiversion broadcast method produces correct read-only transactions.*

Proof. Let R be a read-only transaction, c_0 the cycle at which R performs its first read operation and DS^{s_0} the database state broadcasted at cycle c_0. We will show that the values read by R correspond to the database state DS^{s_0} which is consistent and thus R is correct. For any data item $x \in RS(R)$, R reads the largest version c_n of x, such that $c_n \leq c_0$. This value is the most recent value of x produced before cycle c_0, that is the value that the item had at DS^{s_0}. $\square$.

As shown from the proof above, R is serialized after all transactions that were committed prior to c_0 and before all transactions that were committed after bcycle c_0. In terms of currency, the data items read by T correspond to the database state at the beginning of the bcycle c_0.

Instead of broadcasting all S previous versions, the server may broadcast V older versions, for some constant $V \leq S$. In this case, the number V of older versions that are retained can be seen as a property of the server. In that sense, a V-multiversion server, i.e., a server that broadcasts the previous V values, is one that guarantees the consistency of all transactions with span V or smaller. Transactions with larger spans can proceed on their own risk; their consistency cannot be guaranteed. V, i.e., the amount of broadcast reserved for old versions, can be adapted depending on various parameters, such as the allowable bandwidth, feedback from clients, or update rate at the server.

Multiversion Broadcast Organization

One way to structure the broadcast is to broadcast all versions of an item successively (Figure 2(a)). In such an organization, the location of each data item is not the same at all bcasts, thus clients can not anymore utilize a locally stored directory to determine the position of items in the broadcast. Consequently, prior to each cycle, the server must reconstruct an index structure and broadcast it along with data, further increasing the overall size of the broadcast. The client must first tune in to get such index information.

An alternative organization in which the position of each data item in the bcast remains fixed, is to broadcast all old versions at the end of the bcast. In particular, instead of broadcasting with each data item all its versions, a single version, the most recent one, is broadcasted along with a pointer. The pointer associated with each data item points to its older versions, if any, that are broadcasted in reverse chronological order at the end of each bcast in *overflow* buckets (Figure 2(b)). This way, for each data item, the offset of its position in the bcast from the beginning of the bcast remains fixed. Thus, the server needs not recompute and broadcast an index, at each broadcast cycle. Instead, the client uses its locally stored directory to locate the first appearance of a data item in the broadcast. After reading the item, if it needs an older version, it uses the pointer to locate it in the overflow bucket.

The drawback of the overflow approach is that long-running read-only transactions that must read old versions are penalized since they have to wait for the end of the bcast to complete their operations. However, transactions that are satisfied with current versions do not suffer from a similar increase in latency. On the contrary, in the first approach, in which all versions of an item are clustered together, the overhead in latency is equally divided among all transactions.

Let v be the size of the version number, k the size of the key, d the size of the other attributes, u the mean number of updates, and S the maximum transaction span. The first approach produces a bcast of size $\lceil \frac{D(k+v+d)+u(S-1)(d+k+v)}{b} \rceil$, where b is the bucket size. With this approach, since the position of each item from the beginning of the bcast is not fixed, additional space may be allocated to broadcast index information. In the second approach, the size of the data buckets is $D(k+d+P)$, where P is the size of the pointer, while the total size of the overflow buckets is $B = \lceil \frac{u(S-1)(k+v+d)}{b} \rceil$. The pointer can be kept as the offset of the beginning of the overflow bucket from the end of the bcast, and thus be analog to the number of overflow buckets, in particular $P = log(B)$. To allocate less space for version numbers, instead of broadcasting the number of the bcycle at which the data item was created, we can broadcast the difference between the current bcycle and the bcycle in which the value was created, i.e., how old the value is. For example, if the current bcycle is cycle 30, and a version was created during bcycle 27, we broadcast 3 as the version of the data value instead of 27. In this case, $log(S)$ bits are sufficient for v.

3.3 Serialization-Graph Testing

Both the invalidation-only and the multiversion schemes ensure that transactions read consistent values, i.e., values produced by a serializable execution, by enforcing transactions to read values that correspond to the content of a single bcast. In the case of the invalidation-only scheme, this is the bcast at the end of the transaction, while in the case of the multiversion scheme this is the bcast at the beginning of the transaction. However, it suffices for transactions to read values that correspond to any consistent database state not necessarily one that is broadcasted. To this end, we use a conflict serialization graph testing (SGT) method.

The serialization graph for a history H, denoted $SG(H)$, is a directed graph whose nodes are the committed transactions in H and whose edges are all $T_i \rightarrow T_j$ $(i \neq j)$ such that one of T_i's operations precedes and conflicts with one of T_j operations in H [7]. According to the serialization theorem, a history H is serializable iff $SG(H)$ is acyclic. We assume that each transaction reads a data item before it writes it, that is, the readset of a transaction includes its writeset. Then, in the serialization graph, there can be two types of edges $T_i \rightarrow T_j$ between any pair of transactions T_i and T_j *dependency* edges that express the fact that T_j read the value written by T_i and *precedence* edges that express the fact that T_j wrote an item that was previously read by T_i.

In brief, the SGT method works as follows. Each client maintains a copy of the serialization graph locally. The serialization graph at the server includes only the transactions *committed* at the server, while, in addition, the local copy at the client includes any active read-only transactions that were issued at this site. At each cycle, the server broadcasts any updates of the serialization graph. Upon receipt of the updates, the client integrates them into its local copy of the graph. A read operation at a client is executed only if it does not create a cycle in the local serialization graph. The serialization graph at the server is not necessarily used for concurrency control at the server, instead a more practical method, e.g., most probably two-phase locking, may be employed.

Implementation of the SGT Method

In the serialization graph testing (SGT) method, the server broadcasts at the beginning of each bcast the following control information:

- the *difference from the previous serialization graph*
  In particular, the server broadcasts for each transaction T_i that was committed during the previous cycle, a list of the transactions with which it conflicts, i.e., it is connected through a direct edge.

- an *augmented invalidation report*
  The report includes all data written during the previous bcycle along with an identification of the first transaction that wrote each of them during the bcycle.

In addition, the content of the broadcast is augmented so that the identification of the last transaction that wrote a data item is broadcasted along with the item.

Each client tunes in at the beginning of the broadcast to obtain the control information. Upon receipt of the graph, the client updates its local copy SG of the serialization graph to include any additional edges and nodes.

We describe next, an efficient method based on the assumption that histories are strict. A history is strict if no data may be read or overwritten until the transaction that previously wrote into it terminates. The method is applicable to other cases as well, but requires additional overhead. Let SG^i be the subgraph of SG that includes only the transactions that were committed during cycle i. An interesting property is that:

Claim 1 *There cannot be any incoming edges to transactions in SG^i from transactions committed in subsequent cycles $m > i$.*

This is true since all transactions in SG^i are committed prior to any transactions committed in subsequent cycles.

At the beginning of each bcycle $i + 1$, the client also adds precedence edges for all its active read-only transactions as follows. Let R be an active transaction and $RS^i(R)$ be the set of items that R has read so far. For each item x in the invalidation report such that $x \in RS^i(R)$, the client adds a precedence edge $R \rightarrow T_f$, where T_f is the first transaction that wrote x during bcycle i. Although R conflicts with all transactions that wrote x during bcycle i, it suffices to just add one edge to T_f since:

Claim 2 *Let $x \in RS^i(R)$ and SG_a be the serialization graph that includes edges $R \rightarrow T$ for each T that wrote x during bcycle i and SG_f the subgraph of SG that includes only one such edge $R \rightarrow T_f$, where T_f is the first transaction that wrote x during bcycle i. SG_a has a cycle if and only if SG_f has a cycle.*

Proof. In the Appendix.

When R reads an item y, a dependency edge $T_l \rightarrow R$ is added in the local serialization graph, where T_l is the last transaction that wrote y. The read operation is accepted, only if no cycle is formed. It can be shown using an argument similar to the one in the previous claim that it suffices to just add one edge $T_l \rightarrow R$ instead of adding edges $T' \rightarrow R$ from all transactions T' that wrote y.

Claim 3 *Let $y \in RS(R)$ and SG_a be the graph that includes edges $T \rightarrow R$ for each T that wrote y and SG_l be the subgraph that includes only an edge $T_l \rightarrow R$, where T_l is the last transaction that wrote y. SG_a has a cycle if and only if SG_l has a cycle.*

We will prove that the SGT method detects all cycles that include a read-only transaction R. We will use the following lemma:

Lemma 1 *Let o be the first broadcast cycle during which an item read by R gets overwritten.*

(a) During broadcast cycle m, the only type of cycles that can be formed that includes R are of the form $R \rightarrow T_{i_1} \rightarrow T_{i_2} \rightarrow \ldots T_{i_k} \rightarrow R$, where for any $T_{i_j} \in SG^l$, it holds $o \leq l < m$.

(b) The SGT algorithm detects all such cycles.

Proof. In the Appendix.

Figure 3 shows graphically the formation of such a cycle.

Theorem 3 *The SGT method produces correct read-only transactions.*

Proof. From Lemma 1, the SGT algorithm detects all cycles that involve R, thus from a direct application of the serialization theorem, R is serializable with all server transactions, thus R reads consistent data and is correct. □

Regarding the currency of read-only transactions, each read-only transaction R that performs its first read at c_0 reads values that correspond to a database state between the state at the beginning of broadcast cycle c_0 and the current database state.

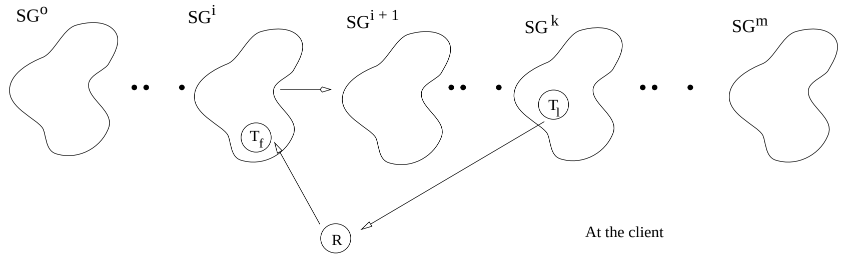


Figure 3: At bcycle m, read-only transaction R reads x from T_l committed during bcycle k ($o \leq k < m$). Transaction T_f committed during bcycle i ($i \leq o$) overwrote an item previously read by R.

Space Efficiency

Instead of keeping a complete copy of the serialization graph locally at each client, by Lemma 1, it suffices to keep for each read-only transaction R only the subgraphs SG^k with $k \geq c_o$, where c_o is the bcycle when the first item read by R was invalidated, i.e., overwritten. Thus, if no items are updated, there is no space or processing overhead at the client. Furthermore, at most S subgraphs are maintained, where S is the maximum transaction span of the queries at the client. By Lemma 1, we may also keep only the outgoing edges from R; there is no need to store the incoming edges to R.

However, the volume of the control information that is broadcasted is considerable. Let tid be the size of a transaction identifier, N the maximum number of transactions committed during a broadcast cycle, and c the maximum number of operations per transaction at the server. We assume that transaction identifiers are unique within each broadcast cycle, thus it suffices to allocate $log(N)$ bits per transaction identifier when the bcycle is known. To distinguish between transactions at different bcycles, a version number is broadcasted indicating the bcycle at which the transaction was committed; the size of such version number is $log(S)$ bits, since only the last S bcycles are relevant. The size of the broadcast data is $\lceil \frac{D(d+k+log(N))}{b} \rceil$, while the size of the invalidation report is $\lceil \frac{u(k+log(N))}{b} \rceil$. Since, there are at most c operations per transaction, each transaction participates in at most c conflicts with other transactions. Thus, the difference from the previous graph has at most $N\,c$ edges. The total size of the difference is: $\lceil \frac{cN(log(N)+(log(S)+log(N)))}{b} \rceil$, assuming that we broadcast pair of conflicting transactions where the first transaction in the pair is a newly committed transaction, and the second one any previously committed transaction with which it conflicts. If we broadcast the control information at the end of the previous bcast, then the offset of each item from the beginning of each bcast remains fixed and a locally stored directory can be used.

4 Caching

To reduce latency in answering queries, clients can cache items of interest locally. Caching reduces not only the latency but also the span of transactions, since transactions find data of interest in their local cache and thus need to access the broadcast channel for a smaller number of cycles. When the items broadcasted are updated, the value of cached items become stale. We assume that each page, i.e., the unit of caching, corresponds to a bucket, i.e., the unit of broadcast.

There are various approaches to communicating updates to the client. The two basic techniques are invalidation and propagation. For invalidation, the server sends out messages to inform the client of which pages are modified. The client removes those pages from its cache. For propagation, the servers sends the updated values. The client replaces its old copy with the new one.

Invalidation combined with a form of autoprefetching has been shown to perform well in broadcast delivery [2]. At the beginning of each broadcast cycle (or at other pre-defined points), the server broadcasts an invalidation report, which is a list of the pages that have been updated. This report is used to invalidate those pages in cache that appear in the invalidation report. These pages remain in cache to be autoprefetched later. In particular, when the new value of an invalidated page appears in the broadcast, the client fetches the new value and replaces the old one. Thus, a page in cache either has a current value (the one in the current broadcast) or is marked for autoprefetching.

The cache invalidation report is similar to the invalidation report used in our query processing schemes. However, the two reports differ in granularity. The cache invalidation report includes the pages (or buckets) that have been updated, whereas the query-processing invalidation report includes the data items that have been updated.

4.1 Extending Caching to Support Read-only Transactions

The proposed approaches can be readily extended to accommodate caching at the client. We assume that an appropriate technique, such as invalidation coupled with autoprefetching, is used to maintain cache consistency. For the invalidation-only scheme, each read first checks whether the item is in cache. If the item is found in cache and the page is not invalidated, the item is read from the cache. Otherwise, the item is read from the broadcast. A simple enhancement to the above scheme is to extend the cache so that along with each value it also includes the bcycle during which the value was inserted in the cache. Let R be a query and u the first bcycle at which an item $x \in RS(R)$ is invalidated. Instead of aborting R, R is marked abort and continues operation as long as old enough values for all future reads can be found in cache. In particular, R continues its read operations as long as the items it wants to read exist in the cache and have versions $c < u$. We call this method invalidation-only with versioned cache.

Theorem 4 *The invalidation-only with versioned cache method produces correct read-only transactions.*

Proof. Let R be a committed query and u be the first bcycle at which an item read by R was invalidated. Let DS^{u-1} be the database state broadcasted at bcycle $u-1$. Then, the values read by R correspond to the database state DS^{u-1}. This holds because all the values read by R till bcycle u correspond to DS^{u-1}, then a value is read only if the version in cache is $c < u$. This value is the value the item had at $u-1$ otherwise it should have been invalidated and a new version should have been autoprefetched. □

To support the multiversion broadcast method, the cache must also include the version number for each item. Similarly for the SGT method, the cache must be extended to include for each item the last transaction that wrote it; information that is broadcasted anyway. Each time an item is read from the cache, the same test for cycles as when the item is read from the broadcast is executed.

4.2 Multiversion Caching

The client cache can be used to provide an alternative storage medium for older versions of data items for those active transactions that may need to read them for concurrency control purposes. We call this approach *multiversion caching (MC)*. In multiversion caching, each entry in the cache has a version associated with it that corresponds to the bcycle when the version was created. When an item is updated at the server, its cache entry is not updated, instead a new entry is inserted in the cache for the new version. Thus, for a data item, there may be multiple entries with different versions.

Multiversion caching can be used in conjunction with all previous schemes to increase concurrency. We present such a multiversion caching method that combines invalidation-only reports with versions. We assume that the cache replacement policy is such that: for each data item, the versions cached are the most recent ones. Let R be a read-only transaction, and c_u the first bcycle during which an item read by R was updated for the first time. Similar to the multiversion broadcast method, in subsequent bcycles, R reads the largest version v of an item such that $v < c_u$. If such a version is found in cache, then it is read from the cache, otherwise the transaction is aborted.

Theorem 5 *The multiversion cache method produces correct read-only transactions.*

Proof. Let R be a read-only transaction, c_u the first bcycle during which an item read by R was updated for the first time and DS^{s_u} the database state broadcasted at bcycle c_u. We will show that the values read by R correspond to the database state $DS^{s_u - 1}$ which is consistent and thus R is correct. The items read before c_u were not updated prior to c_u thus their values correspond to the database state DS^{s_u-1}. In subsequent bcyles, R reads the largest version v, such that $v < c_u$. This value is the most recent value produced before cycle c_u, that is the value that the item had at DS^{s_u-1}. $\square$.

With multiversion caching, the effective cache size is decreased, since part of the cache is used to maintain old versions of items. However, for long-running transactions that read old versions, there may be some speed-up, since older versions may be found in cache. Whereas, S (the number of older versions broadcasted) in the multiversion broadcast, is a property of the server, in multiversion caching, S (the number of versions kept in cache) is a characteristic of each client. Transactions at different clients may have varying spans. In this case, it is the client's responsibility to adjust the space in cache allocated to older versions, based on the size of its cache, the requirements and types of its read-only transactions, or other local parameters. The increase in the broadcast size is that of the invalidation-only method plus the additional space needed to broadcast version numbers.

There are various approaches to cache replacement in a multiversion cache. One is to consider the different versions of an item as different items and replace the page with the overall highest probability of being accessed, without taking into account versions. The one we adopt is dividing the cache space into two parts: one that maintains current versions and one that maintains older ones. In this case, different cache replacement policies can be used for each part of the cache. This approach provides also for adaptability, since the percentage of cache allocated to older versions can be adjusted dynamically.

5 Performance Evaluation

In this section, we comparatively evaluate the techniques proposed with respect to various parameters.

5.1 The Performance Model

Our performance model is similar to the one presented in [1]. The server periodically broadcasts a set of data items in the range of 1 to $BroadcastSize$. We assume for simplicity a flat broadcast organization in which the server broadcasts cyclicly the set of items.

The client accesses items from the range 1 to *ReadRange*, which is a subset of the items broadcasted (*ReadRange* $\leq$ $BroadcastSize$). Within this range, the access probabilities follow a Zipf distribution. The Zipf distribution with a parameter *theta* is often used to model non-uniform access. It produces access patterns that become increasingly skewed as *theta* increases. The client waits $ThinkTime$ units and then makes the next read request.

Updates at the server are generated following a Zipf distribution similar to the read access distribution at the client. The write distribution is across the range 1 to *UpdateRange*. We use a parameter called $Offset$ to model disagreement between the client access pattern and the server update pattern. When the offset is zero, the overlap between the two distributions is the greatest, that is the client's hottest pages are also the most frequently updated. An offset of k shifts the update distribution k items making them of less interest to the client. We assume that during each bcycle, N transactions are committed at the server. All server transactions have the same number of update and read operations, where read operations are four times more frequent than updates. Read operations at the server are in the range 1 to $BroadcastSize$, follow a Zipf distribution, and have zero offset with the update set at the server.

The client maintains a local cache that can hold up to $CacheSize$ pages. The cache replacement policy is LRU: when the cache is full, the least recently used page is replaced. When pages are updated, the corresponding cache entries are invalidated and subsequently autoprefetched. Table 4 summarizes the parameters that describe the operation at the server and the client. Values in parenthesis are the default.

5.2 Comparison of the Methods

5.2.1 Performance Results

Concurrency. Updates at the server may invalidate data values read by read-only transactions and cause them to be aborted and reissued anew. Besides the multiversion broadcast scheme, with which, all read-only transactions are accepted, the other schemes accept only a percentage of the read-only transactions. To estimate this percentage, first we varied the number of read operations per query (Figure 5(left)). Whereas the SGT method with caching outperforms all other schemes, the invalidation-only scheme with versioned cache

Server Parameters		Client Parameters	
D (BroadcastSize)	1000	ReadRange (range of client reads)	250
UpdateRange	500	theta (zipf distribution parameter)	0.95
theta (zipf distribution parameter)	0.95	Think Time (time between client reads in broadcast units)	2
Offset (update and client-read access deviation)	0 - 250 (100)	Number of reads per quey	5 - 50 (10)
ServerReadRange	1000	S (transaction span)	varies
N (number of server transactions)	10		
Offset (update and server-read access deviation)	0	**Cache**	
u (Number of updates at the server)	50 - 500 (50)	CacheSize	125
c (Number of operations per server trans)	(u + 4 * u) / N	Cache replacement policy	LRU
k (size of the key field)	1 unit	Cache invalidation	invalidation + autoprefetch
d (size of the other fields)	5 * k		
b (bucket size)	d units		

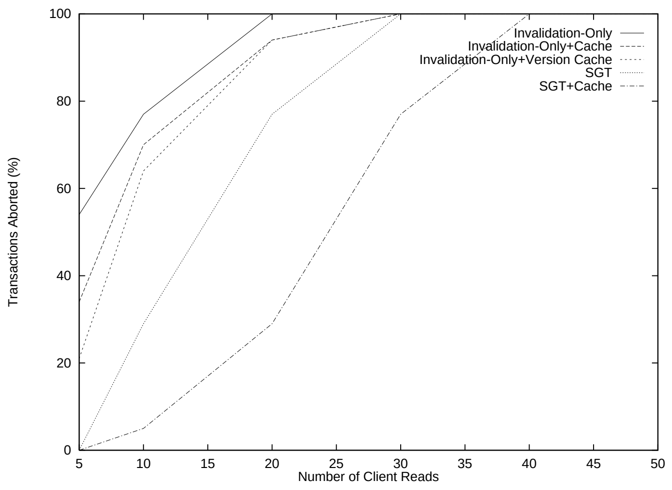
Figure 4: Performance Model Parameters

seems to offer an attractive alternative for queries with less than 30 reads, thus avoiding the considerable overhead of the SGT method. Caching reduces the number of transactions aborted since it reduces their span and thus the probability of invalidation. Then, we considered the overlap between the client read and the server update pattern (Figure 5(right). As expected, when the overlap is the maximum, that is the client's hot data are those that are most frequently updated, all schemes have the highest abort rates. When the overlap is small (less than 50%), the SGT methods accept all transactions.

Finally, we considered the number of updates (Figure 6). In this case, the invalidation-only scheme with versioned cache outperforms all other schemes for a large number of updates (over 1/4 of the $BroadcastSize$). This is because the possibility of cycles in the serialization graph increases with the number of operations at the server. In general, the SGT methods are less attractive than the invalidation-only methods when there is a lot of activity in the server. Thus, while for a small number of operations at the server the SGT methods more than double the number of queries that are accepted, when the number of operations at the server increases, the increase of the accepted transaction decreases to 10% (Figure 6).

Broadcast Size. The increase of the size of the broadcast is an important measure of the efficiency of the proposed schemes, since transmitting it consumes bandwidth. Furthermore, the volume of the broadcast data affects the response time of client transactions. Since access to data is sequential, the larger the volume of the broadcast, the longer the clients need to wait until the data of interest appear on the channel. Figure 7 shows the increase of the broadcast size as a function of the maximum transactions' span and the number of updates using the formulas developed in the previous sections.

Latency. We quantify latency, the mean duration of read-only transactions, as the mean number of bcycles per transaction. None of the methods, but multiversion broadcasting, adds to the latency (besides the need to read control information at each bcycle). For implementing multiversion broadcasting, we use the overflow-bucket approach. This approach imposes an increase in latency, since a number of read-only transactions have to wait for old



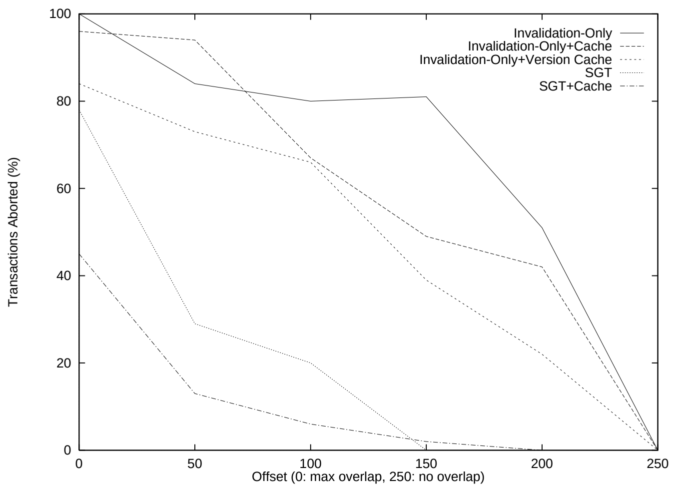


Figure 5: Abort rate: (left) with the number of operations per query (right) with the offset (deviation between the client-read and the server-write pattern)

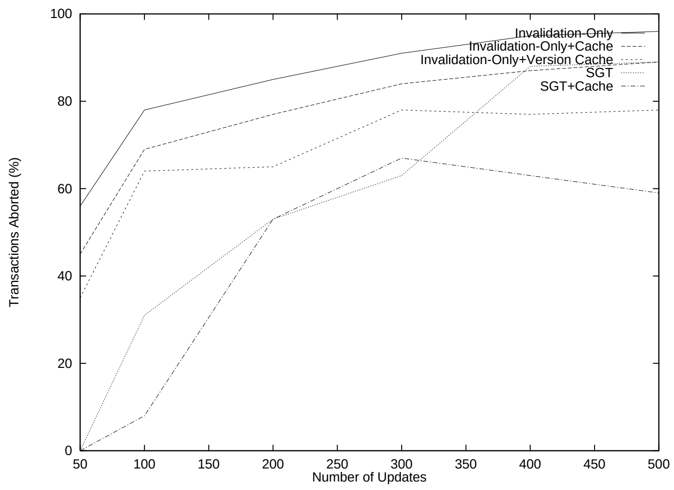


Figure 6: Abort rate with the number of updates

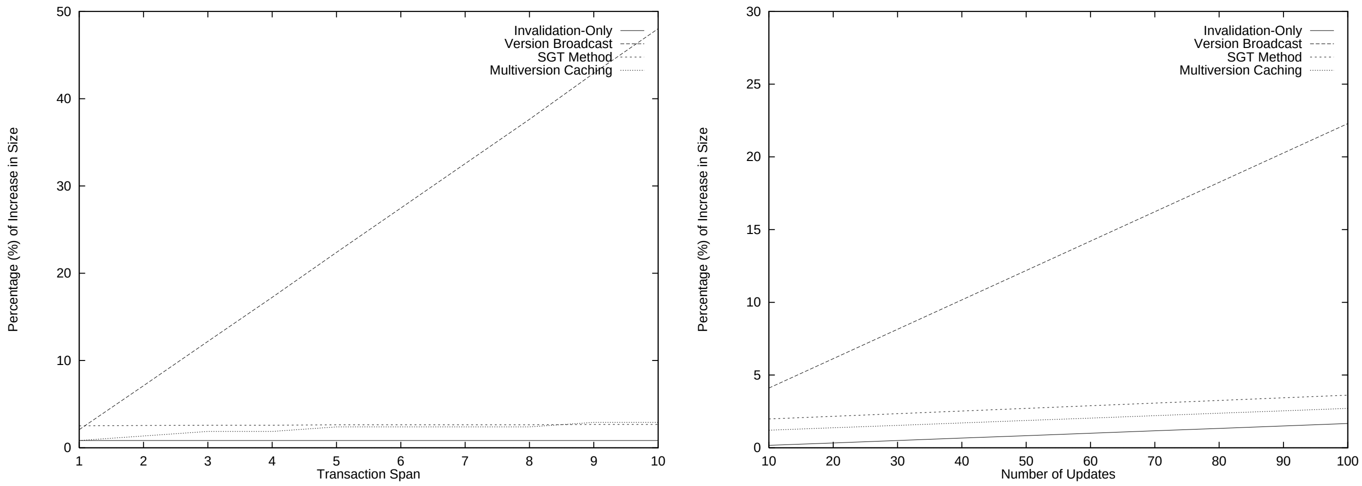


Figure 7: Increase in the size of the broadcast: (left) with the transaction span (for $U = 50$ updates per bcycle) (right) with the number of updates (for $span = 3$)

versions to appear at the end of the bcast. Figure 8(left) shows latency with the number of operations per read-only transaction. The deviation from the expected value (e.g., one half of the read operations for the invalidation-only scheme) is due to the fact that we estimate the latency of the accepted, i.e., non-aborted, transactions only. Figure 8(right) shows the latency of the multiversion broadcasting with the offset. The smaller the overlap between the server-update and the client-read pattern, the smaller the increase in latency.

5.2.2 Other Issues

Currency and Consistency Read-only transactions can be classified based on their currency requirements [12]. *Currency requirements* specify what update transactions are reflected by the data read by read-only transactions, The *consistency requirements* specify the degree of consistency required. Ensuring that the values of a transaction form a consistent database state is a form of *weak* consistency. A *stronger* requirement is that each read-only transaction is serializable along with all update transactions. In multiversion broadcast, this state corresponds to that at the beginning of the read-only transaction, while in the invalidation-only approach, the state corresponds to the current database state. In the serializability method, the state is one in between these two states, in particular a state produced by a serializable execution of a subset of transactions committed during the execution of the read-only transaction. Finally, in multiversion caching, the state seen by read-only transactions is the one when an item read by the transaction is invalidated for the first time. Thus, the invalidation-only method provides the most current view of the database, the serialization and the multiversion caching method a less current one, and the multiversion method the oldest one.

Disconnections. The techniques presented also differ on whether they require active clients to monitor the broadcast continuously. Raising the continuous monitoring requirement is desirable in various settings. For example, in the case of mobile clients, their operation relies

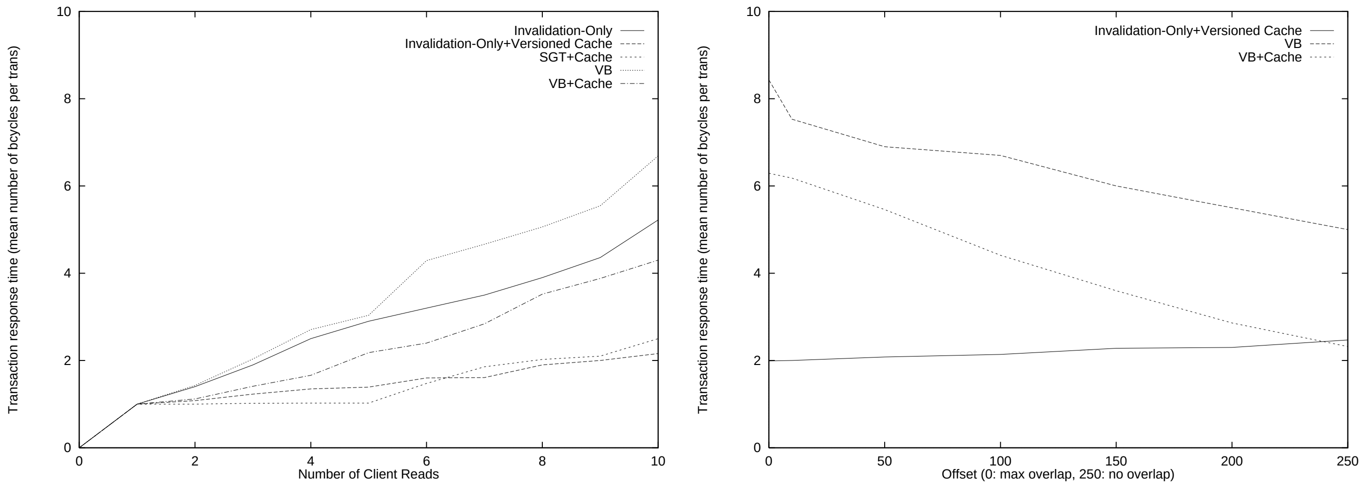


Figure 8: Duration of a transaction: (left) with the number of reads per read-only transaction (right) with the offset

on the finite power provided by batteries, and since listening to the broadcast consumes energy, selective tuning is required. Besides, access to the broadcast may be monetarily expensive, and thus minimizing access to the broadcast is sought for. Finally, client disconnections are very common when the data broadcasted are delivered wirelessly. Wireless communications face many obstacles because the surrounding environment interacts heavily with the signal, thus in general wireless communications are less reliable and deliver less bandwidth than wireline communications. In such cases, clients may be forced to miss a number of broadcast cycles.

In the invalidation-only scheme, a client has to tune-in at each and every cycle to read the invalidation report. Otherwise, it cannot ensure the correctness of any active read-only transaction. On the other hand, in multiversion broadcast, client transactions can refrain from listening to the broadcast for a number of cycles and resume execution later as long as the required versions are still on air. In general, a transaction R with $span(R) = s_R$ can tolerate missing up to $S - s_R$ broadcast cycles in any S-multiversion broadcast. The tolerance of the multiversion scheme to intermittent connectivity depends also on the rate of updates, i.e., the creation of new versions. For example, if the value of an item does not change during k, $k > S$, cycles, this value will be available to any read-only transactions for more than S cycles. The SGT method does not tolerate any client disconnections. If a client misses a broadcast cycle, it cannot anymore guarantee serializability. Thus, any active read transactions must be reissued anew. An enhancement of the scheme to increase tolerance to disconnections would be to broadcast along with items version numbers. Then, a read operation could be accepted as long as its version number was smaller than the version of the last broadcast that the transaction has listen to. This guarantees that the client has all the information required for cycle detection. In all the schemes, periodic retransmission of control information would increase their tolerance to intermittent connectivity. For instance, an invalidation report of the items updated during the last w bcycles may be broadcasted to allow clients to resynchronize. Finally, version

	Invalidation-Only	Multiversion Broadcast	SGT Method	Multiversion Caching
Concurrency (percentage of transactions accepted)	Minimum	Maximum (all transactions accepted)	Moderate (depends on the trans activity at the server)	Moderate (depends on the cache size)
Processing Overhead	Small	Moderate	Considerable (includes maintaining SGs at both the server and the client)	Small
Size (increase of the broadcast volume)	depends on the update rate (1% for U = 50 updates and span = 3)	depends on the update rate and the span (12 % for U = 50 updates and span = 3)	depends on the activity at the server (2.5 % for N = 10 server trans and U = 50 updates)	depends on the update rate (1.8 % for U = 50 updates and span = 3)
Latency (number of bcycles)	Not affected	Increases for long transactions	Not affected	Not affectet
Currency (database state seen by the clients)	The state when the last read is performed	The state when the first read operation is performed	A state between the first and the last operation	The state when an item previoulsy read is overwritten for the first time
Tolerance to Disconnections	None	Some, depends on the individual transaction's span and the update rate	None, unless additional information is broadcasted	Some, depends on the update rate and the cache size

Table 1: Comparison of the Proposed Approaches

caching improves the tolerance to disconnections, since a read-only transaction can proceed as long as appropriate versions can be found in cache.

6 Related Work

Recently, there has been considerable interest on broadcast delivery (for a review, see for example [11] and Chapter 4 of [17]). Updates have been mainly treated in the context of caching. In this case, clients maintain a local cache of the data of interest. Invalidating cache entries by broadcast is the focus of much current research, including [6], [2], [10], and [15]. Updates are considered in terms of local cache consistency; there are no transaction semantics.

A weaker alternative to serializability for transactions in broadcast systems is proposed in [19]. In this work, read only transactions have similar semantics with weak transactions in the conflict-serializability approach. However, the emphasis is on developing and formalizing a weaker serializability criterion rather than on protocols for enforcing them. Finally, broadcast in transaction management is also used in the certification-report method [5]. Read-only transactions in the certification-report method are similar to read-only transactions in the invalidation-only method. However, in the certification-report method, data delivery is on demand, the broadcast medium is mainly used by the server to broadcast concurrency control information to its clients.

7 Conclusions and Future Work

We have presented a set of processing techniques that provide support for consistent queries for broadcast push delivery in both wired and wireless settings with mobile or stationary clients. The techniques are scalable in that their performance is independent of the number of clients. We have compared the proposed techniques both quantitatively and through simulation and show their relative advantages.

There are various ways that the proposed techniques can be extended. First, instead of the invalidation reports being broadcasted at the beginning of each broadcast cycle, such reports can as well be broadcasted at other pre-specified intervals h, $h \leq T$, where T is the period of the broadcast. In this approach, the values broadcasted correspond to the values produced by all transactions committed by the beginning of the current interval, while the invalidation reports include all items updated during h.

Second, there can be variations of the proposed schemes with regards to granularity. For example, invalidation reports may include buckets instead of items. A bucket is considered updated if any of its items has been updated. Instead of maintaining for each transaction R the set of items it has read, the set of buckets is maintained. Then, a query is aborted if one of the buckets it has read is subsequently updated. This scheme may lead to aborting queries that normally should not have been aborted, since the invalidation of a bucket does not necessarily means that the specific item read has also been updated. However, only correct queries are accepted and the imposed overhead is limited. Similarly, in multiversion broadcast, versions of buckets, as opposed to items, may be kept. With this approach, a single pointer per bucket is maintained that points to older versions of the bucket. Such older versions of buckets include older versions of those items of the bucket that were updated during the S previous cycles.

Finally, the methods were presented for a flat broadcast organization, in which all items are broadcasted with the same frequency. One possible extension is to consider a broadcast-disk organization [1], where specific items are broadcasted more frequently than others, i.e., are placed on "faster disks". An interesting problem related to read-only transaction processing is determining the optimal frequency for transmitting invalidation reports or versions, e.g., placing older versions in slower disks.

References

[1] S. Acharya, R. Alonso, M. J. Franklin, and S. Zdonik. Broadcast Disks: Data Management for Asymmetric Communications Environments. In *Proceedings of the ACM SIGMOD Intl. Conference on Management of Data (SIGMOD 95)*, June 1995.

[2] S. Acharya, M. J. Franklin, and S. Zdonik. Disseminating Updates on Broadcast Disks. In *Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB 96)*, September 1996.

[3] AirMedia. AirMedia Live. www.airmedia.com.

[4] A. H. Ammar and J. W. Wong. The Design of Teletext Broadcast Cycles. *Performance Evaluation*, 5(4), 1985.

[5] D. Barbará. Certification Reports: Supporting Transactions in Wireless Systems. In *Proceedings of the IEEE International Conference on Distributed Computing Systems*, 1997.

[6] D. Barbará and T. Imielinski. Sleepers and Workaholics: Caching Strategies in Mobile Environments. In *Proceedings of the ACM SIGMOD Intl. Conference on Management of Data (SIGMOD 94)*, pages 1–12, 1994.

[7] P. A. Bernstein, V. Hadjilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addisson-Wesley, 1987.

[8] A. Bestavros and C. Cunha. Server-initiated Document Dissemination for the WWW. *IEEE Data Engineering Bulletin*, 19(3), September 196.

[9] T. Bowen, G. Gopal, G. Herman, T. Hickey, K. Lee, W. Mansfield, J. Raitz, and A. Weinrib. The Datacycle Architecture. *Commuinications of the ACM*, 35(12), December 1992.

[10] A. Datta, A. Celik, J. Kim, D. VanderMeer, and V. Kumar. Adaptive Broadcast Protocols to Support Efficient and Energy Conserving Retrieval from Databases in Mobile Computing Environments. In *Proceedings of the 13th IEEE International Conference on Data Engineering*, April 1997.

[11] M. J. Franklin and S. B. Zdonik. A Framework for Scalbale Dissemination-Based Systems. In *Proceedings of the OOPSLA Conference*, pages 94–105, 1997.

[12] H. Garcia-Molina and G. Wiederhold. Read-Only Transactions in a Distributed Database. *ACM Transactions on Database Systems*, 7(2):209–234, June 1982.

[13] D. Gifford. Polychannel Systems for Mass Digital Communication. *Communications of the ACM*, 33(2), 1990.

[14] T. Imielinski, S. Viswanathan, and B. R. Badrinanth. Data on Air: Organization and Access. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):353–372, May/June 1997.

[15] J. Jing, A. K. Elmargarmid, S. Helal, and R. Alonso. Bit-Sequences: An Adaptive Cache Invalidation Method in Mobile Client/Server Environments. *ACM/Baltzer Mobile Networks and Applications*, 2(2), 1997.

[16] C. Mohan, H. Pirahesh, and R. Lorie. Efficient abd Flexible Methods for Transient Versioning to Avoid Locking by Read-Only Transactions. In *Proceedings of the ACM SIGMOD Conference*, 1992.

[17] E. Pitoura and G. Samaras. *Data Management for Mobile Computing*. Kluwer Academic Publishers, 1998.

[18] R. Rastogi, S. Mehrotra, Y. Breitbart, H. F. Korth, and A. Silberschatz. On Correctness of Non-serializable Executions. In *Proceedings of ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 97–108, 1993.

[19] J. Shanmugasundaram, A. Nithrakasyap, J. Padhye, R. Sivasankaran, M. Xiong, and K. Ramamritham. Transaction Processing in Broadcast Disk Environments. In S. Jajodia and L. Kerschberg, editors, *Advanced Transaction Models and Architectures*. Kluwer, 1997.

[20] Hughes Network Systems. DirectPC Homepage. www.direcpc.com, 1997.

[21] J. Wong. Broadcast Delivery. *Proceedings of the IEEE*, 76(12), December 1988.

[22] T. Yan and H. Garcia-Molina. SIFT – A Tool for Wide-area Information Dissemination. In *Proceedings of the 1995 USENIX Technical Conference*, 1995.

Appendix

Proof of Claim 2

($\Leftarrow$) If SG_f has a cycle then SG_a has a cycle since SG_f is a subgraph of SG_a.
($\Rightarrow$) Let SG_a have a cycle. Assume for the purposes of contradiction that SG_f is acyclic. Then, the cycle of SG_a must include an edge that does not belong to SG_f. This must be an edge $R \rightarrow T'$, where T' is a transaction other than T_f that wrote x. Since T' wrote x after T_f, there is an edge $T_f \rightarrow T'$. Thus, there is a path $R \rightarrow T_f \rightarrow T'$ in SG_f. By similar arguments, we can replace any edge in a cycle of the SG_a that does not exist in SG_f by a corresponding path. Thus, SG_f also has a cycle, a contradiction. $\square$

Proof of Lemma 1

(a) For R to be involved in a cycle, R must have both an incoming and an outgoing edge. An outgoing edge $R \rightarrow T_{i_1} \in SG^n$ is to a transaction T_{i_1} that overwrote an item read by R, thus $n \geq o$. From Claim 1, the edges among subgraphs of transactions committed at different broadcast cycles go from transactions committed at previous cycles to transactions committed at subsequent cycles, thus $l < m$. The outgoing edge $T_{i_k} \rightarrow R$ is from the last transaction T_{i_k} that wrote an item read by R.
(b) To prove that the SGT algorithm detects all such cycles, we must show that: (i) the local serialization graph at the client includes all such cycles and, (ii) the SGT algorithm detects them. (i) Since the server broadcasts all edges and nodes involving server transactions, it suffices to show that all incoming and outgoing edges to R are included in the local graph. From Claims 2 and 3, it suffices to include edges that involve T_f and T_l, as the SGT does. (ii) Since the graph at the server is acyclic, cycles may be created only when an incoming or outgoing edge to R is added. We claim that such cycles may be formed only when an incoming edge to R is added, that is only when an item is read. Assume for the purposes of contradiction that the addition of an outgoing, i.e., precedence, edge $R \rightarrow T_j$ can create a cycle. Such edges are added at the beginning of each broadcast cycle k, for each transaction $T_j \in SG^k$, where T_j is the first transaction that overwrote an item previously read by R. For a cycle to be formed, there must also be an edge $T_i \rightarrow R$, where from part (a), $T_i \in SG^q$ with $q \geq k$, that means that R read an item from the last transaction T_i that wrote this item during broadcast cycle q, which is impossible since no values produced during broadcast cycles $q \geq k$ have been read yet. $\square$