SUPPORTING READ-ONLY TRANSACTIONS IN WIRELESS BROADCASTING

E PITOURA

13-98

Preprint no. 13-98/1998

Department of Computer Science
University of Ioannina
451 10 Ioannina, Greece

Supporting Read-Only Transactions in Wireless Broadcasting*

Evaggelia Pitoura
Department of Computer Science
University of Ioannina
GR 45110 Ioannina, Greece
Phone : + 30 (651) 97 311
Fax : + 30 (651) 48 131
email: pitoura@cs.uoi.gr

Abstract

Wireless communications support a new form of data delivery in which servers broadcast data to a number of clients that listen to the broadcast channel and retrieve data of interest as they arrive on the channel. In this paper, we address the problem of ensuring the consistency and currency of read-only transactions when the values of broadcast data change. We identify a set of criteria that methods for ensuring consistency in wireless mobile computing must satisfy. We then present a number of such methods and evaluate the degree at which they fulfill the criteria set. Consistency is ensured without contacting the server.

Keywords: broadcast, transaction management, mobile wireless computing

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 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. In push-based data delivery, the server repetitively broadcasts data to a client population without a specific request. Clients

*University of Ioannina, Computer Science Department, Technical Report No: 98-013

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, 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, 24]). Many commercial products have been developed that provide wireless dissemination of Internet-available information. For instance, the AirMedia's Live Internet broadcast network [2] wirelessly broadcasts customized news and information to subscribers equipped with a receiver antenna connected to their personal computer. Similarly, Hughes Network Systems' DirectPC [22] 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 contacted in the area of Teletext and Videotex systems [4, 23]. Previous work also includes the Datacycle project [9] at Bellcore and the Boston Community Information System (BCIS) [12]. 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 broadcast 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 mobile computing because of the physical support for broadcast in both satellite and cellular networks. Broadcast delivery in mobile wireless computing poses a number of difficulties. Mobile clients are resource-poor in comparison to stationary servers. Energy conservation is a major concern. The communication environment is asymmetric, in that there is typically more communication capacity from servers to clients than in the opposite direction.

In this paper, we address the problem of preserving the consistency of client's read-only transactions when the values of data broadcast are updated at the server. A set of desired properties is identified that must be satisfied by transaction processing techniques in wireless mobile computing. Then, we present two different approaches to the problem. One

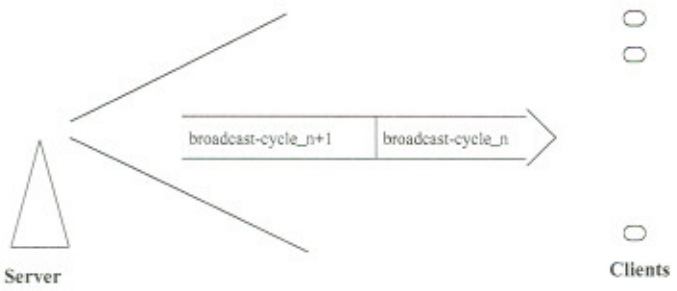

Figure 1: Broadcast-based data delivery

approach is based on broadcasting multiple versions of data items. The other approach uses a conflict serialization graph in conjunction with invalidation reports to ensure serializable executions. We evaluate both methods based on the criteria set. In all the methods presented, consistency is ensured without contacting the server.

The remainder of this paper is organized as follows. In Section 2, we introduce th problem of supporting consistent read-only transactions in the presence of updates. In Section 3, we identify a set of properties according to which schemes for supporting consistency should be evaluated. A method for supporting consistent reads that is based on broadcasting multiple versions of data items is presented in Section 4, while a method that utilizes a serializability graph testing technique is introduced in Section 5. In Section 6, we discuss related work. Finally, in Section 7, we conclude the paper with a comparison of the proposed techniques.

2 Read-Only Transactions and Broadcast Delivery

The server periodically broadcasts all data items to a large client population. Each period of broadcast is called a broadcast cycle. Each client listens to the broadcast and fetches data as they arrive; clients cannot make any direct requests for data (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. Clients access data from the broadcast in a read-only manner. Any updates are applied at the server and disseminated from there. Providing transaction support tailored to read-only transactions is important for

many reasons. First, a large number 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. First, consistency is 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.

For clarity of presentation, we assume that the data content of the broadcast remains the same, that is no items are deleted or added to the broadcast. However, the methods can be easily extended to handle such cases.

2.1 Organization of the Broadcast

Clients do not need to continuously listen to the broadcast. They tune-in to read specific items. To do so, 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 broadcast 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 in [13, 14, 15].

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 broadcast cycle as an offset from the beginning of the broadcast cycle as well as the offset to the beginning of the next broadcast cycle. The offset to the beginning of the next broadcast cycle can be used by the client to determine the beginning of the next broadcast cycle 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.

In order for the clients to tune in at the right time, there is a need for synchronization.

To take care of small discrepancies in distributed clocks, the client may tune in epsilon buckets in advance, where epsilon depends on the accuracy of the clock synchronization. Furthermore, it takes some time for the client to tune in and out of the broadcast. If this set-up time is not negligible compared to the time it takes to broadcast a bucket, it must be taken into consideration in accessing data by modifying the access protocol, for instance by tuning in ahead of time to account for the set-up time [15].

2.2 Read-Only Transactions and Updates

A database state is typically defined as a mapping of every data to a value of its domain. 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 state is consistent if the integrity constraints are satisfied [7]. While data items are being broadcast, transactions are executed at the server that may cause updates of data. We assume that the values of data items that are broadcast during a 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.

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 broadcast cycles, that is data 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 by which they are broadcast 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. Our correctness criterion for read-only transactions is that

each transaction reads consistent data. In particular, the values read by each read-only transaction must form a subset of a consistent database state [20]. 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 execution [7]) of a number of transactions produces a consistent database state. The goal of the methods presented in this paper is to ensure that the values read by each read-only transaction correspond to such a state.

3 Parameters of Concern

In this section, we identify some of the desired properties that processing techniques for read-only transactions in wireless broadcast must posses.

3.1 Type of Read-Only Transactions

One important characterization of a processing schema for read-only transactions is the type of read-only transactions it supports. Read-only transactions can be classified based on their consistency and currency requirements [11]. The *consistency requirements* specify the degree of consistency required by read-only transactions. Ensuring that the values of a transaction form a consistent state is a form of *weak* consistency. A *stronger* requirement is that each read-only transaction is serializable along with all update transactions. *Currency requirements* specify what update transactions are reflected by the data read by read-only transactions.

Updates at the server may invalidate data values read by read-only transactions and cause read-only transactions to be aborted and reissued. Another important characterization of transaction processing techniques is the degree of concurrency they provide, that is how many read-only transactions can proceed along with updates at the server.

3.2 Volume of Control Information

To guarantee correctness, additional control information must be broadcast along with data. Processing of control information is required at both the client and the server. The server must compute and broadcast this information during broadcast cycle. The client must read this information from the broadcast channel and interpret it appropriately.

The size of this control information is an important measure of the efficiency of a transaction processing scheme, since transmitting control information consumes bandwidth. Another requirement is minimizing the overhead of processing this information both at the server and at the clients. Finally, 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.

3.3 Tuning Time

Listening to the broadcast consumes energy. Energy consumption is a major concern in the case of portable mobile computers, since they most often rely for their operation on the finite energy provided by batteries. Even with advances in battery technology, this concern will not cease to exist. Thus an additional requirement posed in mobile systems is minimizing the amount of time that clients spent listening to the channel. This time is called *tuning time*.

To minimize tuning time, techniques have been proposed to provide index or hashing based access to broadcast data [15]. Schemes to support consistent reads must adhere to the requirement of minimizing tuning time. Issues include appropriate organization of control information, for example whether control information should precede or be interleaved with data, and support for indexing and hashing. When the location of each data item in the broadcast remains fixed, another approach is to maintain of an index for the data of interest locally at each client. This approach assumes that clients have enough storage capacity to maintain such copies.

3.4 Tolerance to Disconnections

Listening to the broadcast consumes energy. In addition, access to the broadcast data may be monetarily expensive. Thus, mobile clients may voluntary skip listening for a number of broadcast cycles. Besides this voluntary form of disconnection, client disconnections are very common when broadcast data 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. Thus, a desirable requirement from a broadcasting scheme is to allow clients continue their operation after periods during which the clients miss listening to the broadcast signal.

4 Multiversion Broadcast

One way to support read-only transactions is for the server to maintain and broadcast multiple versions for each data item. Instead of broadcasting the last committed value for each data item, the values that the item had at the beginning of the last x broadcast cycles are transmitted along with a version number that indicates the broadcast cycle to which the version corresponds. The value of x is equal to S, the maximum transaction span among all read-only transactions.

The read-only transactions supported by this scheme are strongly correct. To see that, let c_0 be the cycle at which a transaction T performs its first read operation. T is serialized after all transactions that committed prior to c_0 and before all transactions that committed after broadcast cycle c_0. In terms of currency, the data items read by T correspond to the database state at the beginning of the broadcast cycle c_0.

There are a number of variations of this schema depending on how versions are broadcast. We consider two of them.

4.1 Broadcast with Fixed Periodicity

The last S values of each data item are always broadcast even when the data item is not updated during any of the S cycles. Thus, for some data items, the same value may be repeated on the broadcast. Since, the size of each broadcast cycle does not change, each data item can be broadcast at the same position in every broadcast cycle. Thus, clients can locally cache a copy of the directory.

For each data item, versions are transmitted in reverse chronological order, e.g., the most recent first. At each broadcast cycle, the server shifts the data values for each data item to the right and appends the new value at the front (Figure 2(a)). During its i-th read cycle ($1 \leq i \leq S$), the client reads the data version at position $S + 1 - i$ for each data item. There is no need to broadcast version numbers, since the position of the value on the

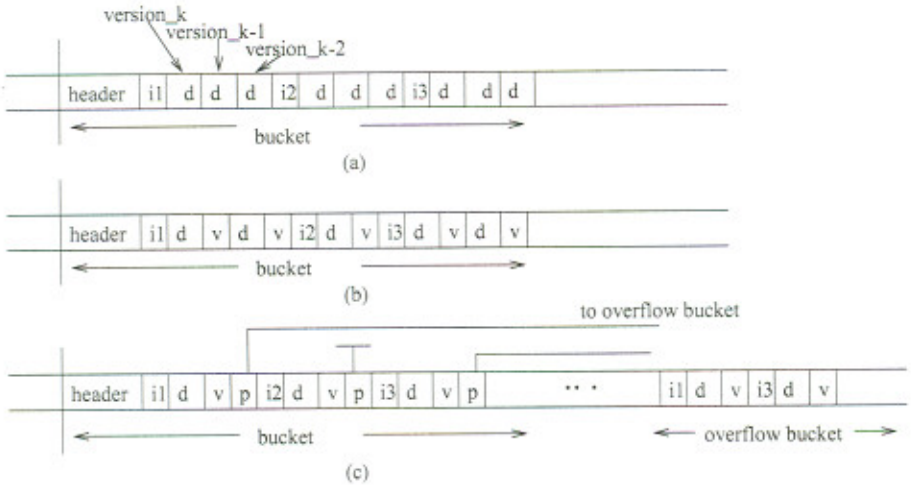

Figure 2: Multiversion broadcast with $S = 3$: (a) fixed-sized, (b) variable-sized, (c) variable-sized with overflow buckets

broadcast implies its version.

Let D be the number of data items in the database, i the size of the search key and d the size of the remaining attributes. Then the size of each broadcast is $D(i + Sd)$ and is fixed for all cycles. There is no need to broadcast index information. The client can use its locally stored directory to tune in when the data item of interest is broadcast. The access protocol remains the same as in the single version case, except that if not all versions of an item fit in one bucket, the client must read additional buckets. However, latency is increased due to the increase of the overall size of the broadcast data.

4.2 Broadcast with Variable Periodicity

Another approach is to broadcast a new value for a data item only if the item was updated during the previous broadcast cycle. In this case, there is a need to broadcast a version number along with each data value. At each cycle, the server discards the $k - S$ version, where k is the current broadcast cycle. At least one value (the current one) is broadcast for each data item. A new value is added to the broadcast, only for the data items that were updated during the previous broadcast cycle (Figure 2(b)). At the client, during the first broadcast cycle, a transaction reads the most up-to-date value for each data item, that is, the version with the largest version number. Let c_0 be the broadcast number of the first broadcast cycle for T. In later cycles, T reads the version with the largest version number

c_n such that $c_n \leq c_0$.

Let v be the size of the version number. The size of the broadcast is $D(i+v+d)+(d+v)u(S-1)$, where u is the number of data items updated during the broadcast. To allocate less space for version numbers, instead of broadcasting the number of the broadcast cycle at which the data item was created, we can broadcast the difference between the current broadcast cycle and the cycle in which the value was created, i.e., how old the value is. For example, if the current broadcast cycle is cycle 30, and a version was created during cycle 27, we broadcast 3 as the version of the data value instead of 27. Since the location of each data item in the broadcast is not fixed, clients can not anymore utilize a locally cached directory to determine the position of items in the broadcast. Thus, prior to each cycle, the server must reconstruct an index structure and broadcast it along with data, thus further increasing the overall size of the broadcast. The client must first tune in to get index information. Again, there is an additional increase in latency and tuning time analog to the increase in the broadcast size.

To keep the position of each data item in the broadcast fixed, instead of broadcasting with each data item all its versions, we may broadcast just a single version: the most recent one. A pointer associated with each data item points to older versions that are broadcast in reverse chronological order at the end of the cycle in *overflow* buckets (Figure 2(c)). Thus, the server needs not recompute and broadcast an index in each broadcast cycle. Instead, the client uses its locally stored directory to locate the first appearance of the data item in the broadcast. After reading the item, if it needs an older version, it uses the pointer to locate older versions of the item in the overflow bucket. The size of data buckets is $D(i+d+v+P)$, where P is the size of the pointer, while the total size of the overflow buckets is $B = u(S-1)(d+v)+u\,i$. The pointer can be kept as the offset of the beginning of the overflow bucket from the end of the broadcast, and thus be analog to the number of overflow buckets, in particular $P = log(\frac{B}{b})$, where b is the size of a bucket in bytes.

Regarding disconnections, a transaction aborts, if all version numbers of all available data values for a data item are larger than c_0. In general, a transaction T with $span(T) = s_T$ can tolerate missing up to $S - s_T$ broadcast cycles. In addition, if a value does not change during the next k cycles, a transaction can tolerate to miss up to $k-1$ broadcast cycles in a row. Tolerance to disconnections can be improved if additional versions of data items are

broadcast.

5 Invalidation-Based Consistency

The multiversion method ensures that transactions read consistent values, i.e., values produced by a serializable execution, by enforcing transactions to read values that correspond to a state at the end of some broadcast cycle. However, it suffices for transactions to read values that correspond to any consistent database state not necessarily one at the end of some broadcast cycle. In other words, it suffices to ensure that the values read by a transaction are that produced by a serializable execution of a subset of the committed transactions. To this end, we use a conflict serialization graph.

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 acyclic 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, there can be two types of edges $T_i \rightarrow T_j$ from any transaction T_i to any transaction T_j in the serialization graph: *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.

Each client maintains a copy of the serialization graph locally. At each cycle, the server broadcasts any updates of the serialization graph. Upon receipt of the graph updates, the client integrates the updates into its local copy of the graph. The serialization graph at the server includes all transactions committed at the server. The local copy at the client in addition includes any alive read-only transactions. A transaction is *alive* if it has performed some operation but has not yet been committed.

5.1 Conflict Serializability

The content of the broadcast is augmented with the following control information:

- an *invalidation report*

  The report includes all data written during the previous broadcast cycle along with

an identifier of the transaction that first wrote each data item and an identifier of the transaction that last wrote each data item.

- 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 transactions with which it conflicts, i.e., it is connected through a direct edge plus a one-bit indication of the type of edge (e.g, dependency or precedence).

The server maintains this control information and broadcasts it at the beginning of each broadcast cycle. Each client tunes in at the beginning of the broadcast to obtain the information. Upon receipt of the graph, the client updates its local copy of the serialization graph to include any additional edges and nodes. It also uses the invalidation report to add new precedence edges for all alive read transactions as follows. Let R be an alive transaction and $RS(R)$ be its readset, that is the set of data items it has read so far. For each item x in the invalidation report, let T_f be the transaction that first wrote x. The client upon receipt of the invalidation report of x, if $x \in RS(R)$, it adds a precedence edge from R to T_f.

When R reads an item y, the client adds a dependency edge from the last transaction T_l that wrote y to R. The read operation is accepted, only if no cycle is formed. In particular, a cycle is formed if there exists a transaction T that overwrote some item y in $RS(R)$ that precedes T_l in the graph, i.e., there is a path from T to T_l.

It can be proved by an application of the serialization theorem that read transactions are strongly correct since they are serializable along with the update transactions at the server. Regarding the currency of the 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.

Let *tid* be the size of a transaction identifier, c be the maximum number of transactions committed during a broadcast cycle, and N be 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 $log(c)$ bits per transaction identifier. When there is a need to distinguish between transactions at different cycles, we also broadcast a version number indicating the broadcast cycle at which the transaction was committed. Then, the

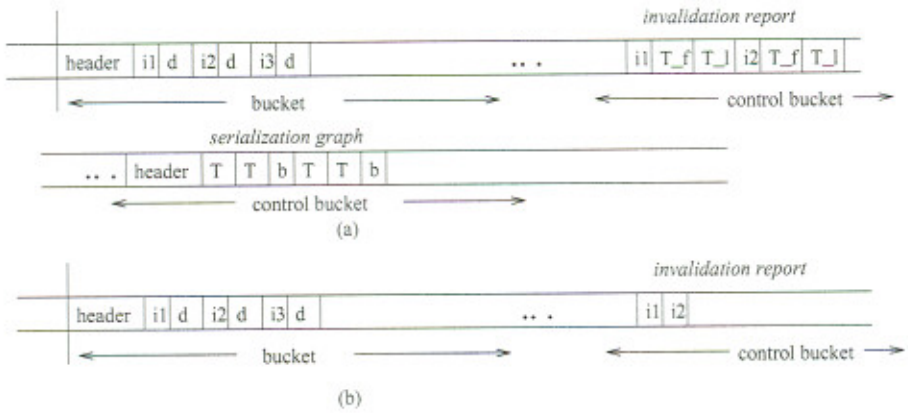

Figure 3: Invalidation broadcast to: (a) ensure conflict serializability (b) just invalidate obsolete reads

size of the invalidation report is: $u(i + 2log(c))$ Since, there are at most N operations per transaction, each transaction may participate in at most N conflicts with other transactions. Thus, the difference from the previous graph has at most cN edges. The total size of the difference is: $cN(2log(c)+2v)$, assuming that along with each transaction we also broadcast the broadcast cycle at which it was committed.

If we broadcast the control information at the end of the cycle, then the position of each item in the broadcast remains fixed and a local directory can be used (Figure 3(a)). Besides the increase in latency due to the increase of the broadcast size, there is an additional increase in latency and tuning time for getting and reading control information.

Besides strong correctness, weak correctness can also be enforced. The method for ensuring weak correctness is based on the assumption that the values written by a transaction depend solely on the data values it reads. Then, we can strengthen our requirement so that the path from T to T_f includes only dependency edges, since precedence edges do not affect the values written by a transaction. This reduces slightly the size of control information, because only dependency edges are relevant and need to be broadcast. However, the main benefit of weak correctness is increased concurrency, since additional read-only transactions may be accepted.

The method does not tolerate any disconnections from the server. If a client misses a broadcast cycle, it cannot anymore guarantee serializability. Thus, any alive read transactions must be reissued anew. An enhancement of the scheme to increase tolerance to

disconnections is to broadcast along with items version numbers. Then, a read operation is accepted if its version number is less than the version of the last broadcast that the transaction has listen to. Another approach to tolerate disconnections is to broadcast periodically summary information, such as the whole serialization graph.

5.2 Invalidation-Only Broadcast

A simpler approach is to broadcast only an invalidation report that includes all data items that were updated during the previous broadcast cycle. In this case, the increase in the size of the broadcast is just equal to $u\,d$ (Figure 3(b)). Then, 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 the method supports strongly correct transactions, since the values of all data read by each transaction R correspond to the current database state. This is the case, since all items read were not updated during any of the subsequent cycles. This method poses minimum overhead. It is adequate when only a few data items are updated and/or the readset of transactions is small.

5.3 Bounded-Inconsistency

Another approach to increase concurrency and reduce the overhead of transmitting and processing control information is to provide read-only transactions that can tolerate imported inconsistency. One formal characterization of inconsistency is provided by *epsilon-serializability* (ESR) [19, 18]. In epsilon-serializability, each read-only transaction has an import-limit that specifies the maximum amount of inconsistency that it can accept. ESR associates an amount of inconsistency with each inconsistent state, defined by its distance from a consistent state. It has meaning for any state that processes a distance function.

Let R be a read-only transaction and c_0 be the cycle at which R starts. The import limit for R can be quantified on a per data item basis. Let $x \in RS(S)$, then the inconsistency associated with x can be defined as the distance of the current value of x and the value of x at c_0 say x_0. R can tolerate reading x_0, and thus import inconsistency equal to this distance, if the distance is within the specified import limit. The inconsistency imported by R depends on the number of concurrent updates, i.e., the number of server transactions that commit while the read-only transaction R is in progress. One way to support this form

of imported inconsistency is to extend the validation report for a data item x to include the number of transactions that have updated x during the previous broadcast cycle.

There are other ways to quantify import inconsistency [3]. For example, for a data item x that takes numerical values, instead of transmitting an invalidation report each time it is updated, we may transmit an invalidation report only when the difference of its new value from its old one falls outside a specified range of values.

6 Related Work

Recently, there has been considerable interest on broadcast delivery (for a review, see for example Chapter 4 of [17] and [10]). 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], [1], and [16]. 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 [21]. 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

We have presented a variety of methods that provide support for consistent read-only transactions. The methods guarantee correctness of read-only transactions by ensuring that transactions read values that correspond to a consistent database state. 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 the transactions committed so far. Thus, the invalidation-only method provides the most current view of the database, the serialization method a less current one, and the multiversion method the oldest one.

In terms of concurrency, in multiversion broadcast, all read-only transactions are accepted as long as the corresponding version exists in the broadcast. However, this method increases considerably the size of the broadcast and accordingly latency. In the invalidation-only method, a read-only transaction is accepted only if its previous read operations are still valid, that is the values read have not been updated. This method has the least overhead among the ones proposed but also provides the least concurrency. Finally, the conflict-serializability method accepts the read-only transactions that do not conflict with the committed at the server transactions.

References

[1] 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.

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

[3] R. Alonso, D. Barbara, and H. Garcia-Molina. Data Caching Issues in an Information Retrieval System. *ACM Transactions on Database Systems*, 15(3):359–384, September 1990.

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

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

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

[13] T. Imielinski, S. Viswanathan, and B. R. Badrinanth. Energy Efficient Indexing on Air. In *Proceedings of the ACM SIGMOD Intl. Conference on Management of Data (SIGMOD 94)*, pages 25–36, 1994.

[14] T. Imielinski, S. Viswanathan, and B. R. Badrinanth. Power Efficient Filtering of Data on Air. In *Proceedings of the the 4th International Conference on Extending Database Technology*, March 1994.

[15] 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.

[16] 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.

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

[18] C. Pu and A. Leff. Replica Control in Distributed Systems: An Asynchronous Approach. In *Proceedings of the ACM SIGMOD*, pages 377–386, 1991.

[19] K. Ramamritham and C. Pu. A Formal Characterization of Epsilon Serializability. *IEEE TRansactions on Knowledge and Data Engineering*, 7(6):997–1007, 1995.

[20] 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.

[21] 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.

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

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

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