Supporting Read-Only Transactions in Wireless Broadcasting*

Evaggelia Pitoura
Department of Computer Science,
University of Ioannina,
GR 45110 Ioannina, Greece
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.

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 sometimes provided with a relatively 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 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., [5, 13]).

Although the concept of broadcast data delivery is not new, recently, it 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 that must be satisfied by transaction processing techniques in wireless mobile computing is identified. Then, we present two different approaches to the problem. One 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 the problem and in Section 3, we identify a set of properties according to which schemes for supporting consistent reads should be evaluated. A method 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. Finally, in Section 6, we conclude the paper with a comparison of the proposed techniques and a brief discussion of related work.

*In Proceedings of the DEXA98 International Workshop on Mobility in Databases and Distributed Systems, 1998. This and related reports are available from www.cs.uoi.gr/~ pitoura/pub.html

2 The Problem

The server periodically broadcasts all data items to a large client population. Each period of broadcast is called a broadcast *cycle*, while the content of the broadcast is called a *bcast*. Each client listens to the broadcast and fetches data as they arrive; clients cannot make any direct requests for data. 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. This is because the consistency of read-only transactions 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, the potential problem of overwhelming the server with a large number of client requests is avoided. In addition, not 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 Broadcast Organization

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. When the location of each data item in the broadcast remains fixed and clients have sufficient storage capacity, an index for the data of interest may be maintained locally at each client. Alternatively, the broadcast can be self-descriptive, in that, some form of directory information is broadcast along with data. In this case, the client 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 [8].

The smallest logical unit of a bcast is called a *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 is used by the client to determine the beginning of the next bcast when the size of the bcast 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 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 [4]. 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 each bcast 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, despite the fact that normal transaction processing takes place at the server during each broadcast cycle and some items may be updated while they are broadcast, the data values on air are the values that the items had at 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 bcasts, 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 bcast. 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 order in which transactions read data affects the response time of queries. Thus, a form of transaction optimization that orders requests for data based on the order by which they are broadcast can keep the transaction's span small.

Since client transactions read data from different cycles, there is no guarantee that the values 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 [11]. 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 [4]) 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

Type of Read-Only Transactions. Read-only transactions can be classified based on their consistency and currency requirements [7]. The *consistency requirements* specify the degree of consistency required by read-only transactions. Ensuring that the values read by a transaction correspond to 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 at the server are reflected by the data read by read-only transactions at the clients. 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.

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 each 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.

Tuning Time. 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. Since listening to the broadcast consumes energy, 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*. Schemes to support consistent reads must adhere to this requirement. To minimize tuning time, techniques have been proposed to provide index or hashing based access to broadcast data [8]. With respect to consistent read-only transactions and 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.

Tolerance to Disconnections. To save money and battery power, mobile clients may voluntary skip listening to a number of bcasts. Besides this voluntary form of disconnection, client disconnections are very common when broadcast data are delivered wirelessly. Wireless communications are in general less reliable and deliver less bandwidth than wireline communications. Thus, a desirable requirement from a broadcasting scheme is to allow clients to 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 of item at the beginning of the last x broadcast cycles are transmitted along with a version number that indicates the broadcast cycle to which each 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 R performs its first read operation. R is serialized after all transactions that committed prior to c_0 and before all transactions that committed after c_0. In terms of currency, the data items read by R correspond to the database state at the beginning of 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 a cycle. Thus, for some data items, the same value may be repeated on the broadcast. Since, the size of the bcast is fixed, each data item can be broadcast at the same position in every cycle. Thus, clients

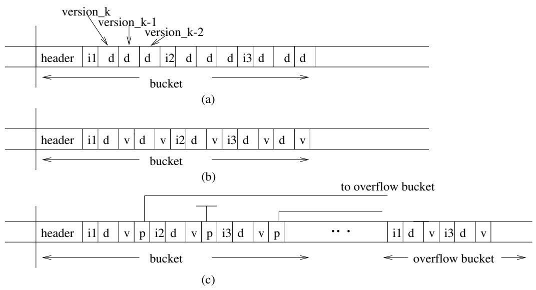

Figure 1: Multiversion broadcast with $S = 3$: (a) fixed-sized, (b) variable-sized, (c) variable-sized with overflow buckets

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 (Fig. 1(a)). During the i-th cycle ($1 \leq i \leq S$) of each read-only transaction, the client reads the data version at position $S + 1 - i$. There is no need to broadcast version numbers, since the position of the value in the bcast implies its version. Let D be the number of data items in the database, i be the size of the search key and d be the size of the remaining attributes. Then, the size of each broadcast is $D(i + Sd)$ and is the same for all cycles. There is no need to broadcast index information, since clients can use a locally stored directory. 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 bcast.

4.2 Broadcast with Variable Periodicity

Another approach is to add a new value to the broadcast, only for the data items that were updated during the previous broadcast cycle (Fig 1(b)). In this case, a version number needs to be broadcast 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. At the client, during the first broadcast cycle c_0, a transaction R reads the most up-to-date value for each data item, that is, the version with the largest version number. In later cycles, R 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. Since the location of each data item in the broadcast is not fixed, clients can no longer 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.

To keep the position of each data item in the bcast 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 bcast in *overflow* buckets (Fig. 1(c)). Thus, the server needs not recompute and broadcast an index. Instead, the client uses its locally stored directory to locate the first appearance of the data item in the broadcast. Then, if it needs an older version, it uses the pointer to locate older versions of the item in the overflow buckets. 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 distance from the beginning of the overflow bucket to the end of the bcast, 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 R with $span(R) = s_R$ can tolerate missing up to $S - s_R$ broadcast cycles. If the items read are not updated 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 beginning of some broadcast cycle. However, it suffices for transactions to read values that correspond to any consistent database state not necessarily one at the beginning 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 [4]. 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, 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.

5.1 Conflict Serializability

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 transaction. A transaction is *alive*, if it has performed some operation but it has not yet been committed. In particular, the content of the broadcast is augmented with the following control information:

- An *invalidation report* that 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, for each transaction T_i that was committed during the previous cycle, the server broadcasts a list of transactions with which it conflicts, i.e., it is connected through a direct edge.

The server maintains this control information and broadcasts it at the beginning of each bcast. Each client tunes in at the beginning of the bcast 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

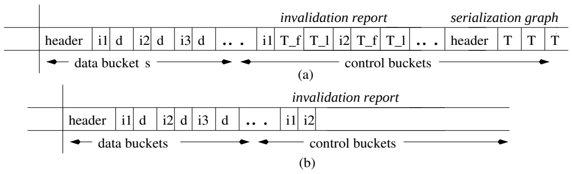

Figure 2: Invalidation broadcast to: (a) ensure conflict serializability (b) just invalidate obsolete reads

first wrote x. Upon receipt of the invalidation report of x, if $x \in RS(R)$, the client 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, if no cycle is formed. 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 proven by an application of the serialization theorem that read-only 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 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 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 $c\,N$ 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 bcast remains fixed and a local directory can be used (Fig. 2(a)).

This method does not tolerate any disconnections from the server. If a client misses a broadcast cycle, it can no longer guarantee serializability. Thus, any alive read-only transactions must be reissued anew. An enhancement 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 listened to. Another approach to tolerate disconnections is to periodically broadcast summary information, such as the complete 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 bcast is just equal to $u\,d$ (Fig. 2(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 minimal overhead. However, all active transactions that have read values that were updated in subsequent cycles are aborted.

6 Conclusions and Related Work

We have presented a variety of methods that guarantee correctness of read-only transactions by ensuring that values read by each transaction 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, it corresponds to the current database state. In the serializability method, the corresponding consistent state is one in between. In terms of concurrency, in multiversion broadcast, any read-only transaction is accepted as long as the corresponding versions exist 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 that it has 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 transactions committed at the server.

Recently, there has been considerable interest in broadcast delivery (for a review, see for example Chapter 4 of [10] and [6]). 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 [3], [1], and [9]. 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 [12], where 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 employed in the certification-report method [2]. Read-only transactions in this 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.

References

[1] S. Acharya, et al. Disseminating Updates on Broadcast Disks. In *Proceedings of the 22nd VLDB Conference*, 1996.

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

[3] D. Barbará and T. Imielinski. Sleepers and Workaholics: Caching Strategies in Mobile Environments. In *Proceedings of the ACM SIGMOD Conference*, 1994.

[4] P. A. Bernstein, et al. *Concurrency Control and Recovery in Database Systems*. Addisson-Wesley, 1987.

[5] A. Bestavros and C. Cunha. Server-initiated Document Dissemination for the WWW. *IEEE Data Engineering Bulletin*, 19(3), 1996.

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

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

[8] T. Imielinski, et al. Data on Air: Organization and Access. *IEEE TKDE*, 9(3):353–372, 1997.

[9] J. Jing, et al. Bit-Sequences: An Adaptive Cache Invalidation Method in Mobile Client/Server Environments. *ACM/Baltzer Mobile Networks and Applications*, 2(2), 1997.

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

[11] R. Rastogi, et al. On Correctness of Non-serializable Executions. In *Proceedings of the ACM PODS Conference*, 1993.

[12] J. Shanmugasundaram, et al. Transaction Processing in Broadcast Disk Environments. In S. Jajodia and L. Kerschberg, editors, *Advanced Transaction Models and Architectures*. Kluwer, 1997.

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