[11] D. Barbara, "Mobile Computing and Databases—A Survey," *IEEE Trans. Knowledge and Data Eng.*, vol. 11, no. 1, pp. 108–117 Jan./Feb. 1999.
[12] P.K. Chrysanthis, "Transaction Processing in Mobile Computing Environment," *Proc. IEEE Workshop Advances in Parallel and Distributed Systems,* pp. 77-83, 1993.
[13] G.D. Walborn and P.K. Chrysanthis, "Supporting Semantics-Based Transaction Processing in Mobile Database Applications," *Proc. Symp. Reliable Distributed Systems,* pp. 31-40, 1995.
[14] S. Mazumdar and P.K. Chrysanthis, "Achieving Consistency in Mobile Databases through Localization in PRO-MOTION," *Proc. Int'l Conf. and Workshop Database and Expert Systems Applications (DEXA),* pp. 82-89, 1999.
[15] S.K. Madria and B.K. Bhargava, "A Transaction Model for Mobile Computing," *Proc. Int'l Database Eng. and Application Symp.,* pp. 92-102, 1998.
[16] S.K. Madria and B.K. Bhargava, "On the Correctness of a Transaction Model for Mobile Computing," *Proc. Int'l Conf. and Workshop Database and Expert Systems Applications,* pp. 573-583, 1998.
[17] B. Yao, K.-F. Ssu, and W.K. Fuchs, "Message Logging in Mobile Computing," *Proc. Symp. Fault-Tolerant Computing,* pp. 294-301, 1999.
[18] D. Kuo, "Model and Verification of a Data Manager Based on Aries," *ACM Trans. Database Systems,* vol. 21, no. 4, pp. 427-479, Dec. 1997.
[19] C. Wallace, Y. Gurevich, and N. Soparkar, "A Formal Approach to Recovery in Transaction-Oriented Database Systems," *Springer J. Universal Computer Science,* vol. 3, no. 4, pp. 320-340, Apr. 1997.
[20] C. Pedregal-Martin and K. Ramamritham, "Guaranteeing Recoverability in Electronic Commerce," *Proc. Third Int'l Workshop Advanced Issues of E-Commerce and Web-Based Information Systems,* pp. 144-155, June 2001.
[21] C. Pedregal-Martin, "Transaction Recovery in Databases and Beyond," PhD thesis, Univ. of Massachusetts, Amherst, 2001.

Multiversion Data Broadcast

Evaggelia Pitoura, *Member*, *IEEE Computer Society*,
and Panos K. Chrysanthis, *Member*, *IEEE*

Abstract—Recently, broadcasting has attracted considerable attention as a means of disseminating information to large client populations in both wired and wireless settings. In this paper, we consider broadcasting multiple versions of data items to increase the concurrency of client transactions in the presence of updates. We introduce various techniques for organizing multiple versions on the broadcast channel. Performance results show that the overhead of supporting multiple versions can be kept low while providing a considerable increase in concurrency. Besides increasing the concurrency of client transactions, multiversion broadcast provides clients with the possibility of accessing multiple server states in a single broadcast cycle. Furthermore, multiversioning increases the tolerance of client transactions of disconnections from the broadcast channel.

Index Terms—Mobile computing, broadcast, transaction management, versioning, consistency.

1 Introduction

Although the concept of broadcast delivery is not new, recently, data dissemination by broadcast has attracted considerable attention due to the physical support for broadcast provided by an increasingly important class of networked environments such as by most wireless computing infrastructures, including cellular architectures and satellite networks [10]. The use of broadcast for disseminating information to large client populations is also motivated by the explosion of data intensive applications created by the dramatic improvements in global connectivity and the popularity of the Internet. In such a setting, the server repetitively broadcasts data to a number of clients without any specific data request. Clients monitor the broadcast channel and retrieve the data items that they may need as they appear on the broadcast channel. Applications typically involve a small number of servers and a much larger number of clients with similar interests, operating in read-only mode. Examples include stock trading, electronic commerce applications, such as auction and electronic tendering, and networks of sensors.

As broadcast-based systems continue to evolve, more and more sophisticated client applications will require reading current and consistent data, despite updates at the server. In most related research, updates are mainly treated in the context of caching at the client (e.g., [4], [2]). In this case, the focus is on cache coherency; there are no transactional semantics. Transactions and broadcast were first discussed in the Datacycle project [5], where special hardware was used to detect changes of values read by transactions and thus ensure consistency. Recent work involves the development of new correctness criteria for transactions in broadcast environments [12], as well as the deployment of the broadcast medium for transmitting concurrency control related information to clients so that part of transaction management can be undertaken by them [3]. In our previous work [8], we proposed and comparatively studied a suite of invalidation-based techniques to ensure the consistency of client read-only transactions.

- *E. Pitoura is with the Department of Computer Science, University of Ioannina, GR 45110 Ioannina, Greece. E-mail: pitoura@cs.uoi.gr.*
- *P.K. Chrysanthis is with the Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260. E-mail: panos@cs.pitt.edu.*

Manuscript received 15 July 2001; revised 15 May 2002; accepted 4 June 2002.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 116692.

In this paper, we propose broadcasting multiple versions of items to increase the concurrency of client transactions. Multiversion schemes have been successfully used to speed up processing of online read-only transactions in traditional database systems (e.g., [6]). Here, we explore their applicability in broadcast databases, as well as the new issues, protocols, and overheads that arise. Versions are combined with invalidation reports to inform clients of updates and thus ensure the currency of their reads. We assume that updates are performed at the server and disseminated from there. The currency and consistency of the values read by clients is preserved without requiring that clients contact the server. We introduce protocols for interleaving current and previous versions and for determining the frequency of broadcasting older versions. Multiversion broadcast was first introduced in [9].

Besides increasing the concurrency of client transactions, multiversion broadcast provides clients with the possibility of accessing multiple server states. For example, such a functionality is essential to support applications that require access to data sequences and have limited local memory to store the previous versions, as is the case with data streams.

Performance results show that the overhead of maintaining older versions can be kept low, while providing a considerable increase in concurrency. For instance, when about 10 percent of the broadcast items are updated per broadcast, maintaining two versions per data item increases the number of consistent read-only transactions that successfully complete their operation from below 40 percent (when only one—the most current—version is maintained) to above 80 percent. The increase of the broadcast size is below 20 percent of the original broadcast size. For less update-intensive environments, the overhead is considerably smaller. Furthermore, in the case of client disconnecting from the broadcast channel, broadcasting two versions per data item results in reducing the number of transactions that are aborted by a factor ranging from 25 percent to 90 percent, depending on the frequency and the duration of the disconnections.

The remainder of this paper is organized as follows: Section 2 introduces the problem and presents two basic approaches for maintaining consistency. Section 3 describes the multiversioning scheme, while Section 4 proposes various protocols for interleaving versions on the broadcast channel. Section 5 presents our performance model and experimental results. Section 6 concludes the paper.

2 Broadcast and Updates

In our model, a data 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. This way, data can be accessed concurrently by any number of clients without the performance degradation that would result if the server were to submit data to individual clients. However, access to data is strictly sequential since clients need to wait for the data of interest to appear on the channel. The smallest logical unit of a broadcast is called a *bucket*. Buckets are the analog to blocks for disks. Data items correspond to database records (tuples). Users access data by specifying the value of one attribute of the record, the search key. Each bucket may contain several data items.

We assume that all updates are performed at the server and disseminated from there. Let us consider what value for an item x is placed on the broadcast at some time t. There are two reasonable choices: 1) *Immediate Updates:* The value that is placed on the broadcast channel at time t for an item x is the most recent value of x (that is, the value of x produced by all transactions that committed at the server by t). 2) *Periodic Updates:* Updates at the server are not reflected on the broadcast content immediately, but at each bcycle. In particular, the value of item x that the server places on the broadcast at time t is the value of x produced by all transactions committed at the server by the beginning of the current bcycle. Note that this may not be the value of x at the server at time t if, between the beginning of the current bcycle and t, x has been updated at the server. Without loss of generality, in this paper, we assume periodic updates.

2.1 Updates and Consistency

We assume that the server broadcasts items from a database. A database consists of a finite set D of data items. A *database state* is typically defined as a mapping of every data item to a value of its domain. Thus, a database's state, denoted DS, is a set of ordered pairs of data items in D and their values. In a database, data are related by a number of 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.

A client transaction may read data items from different bcycles. We define the *span* of a client transaction R, $span(R)$, to be the maximum number of different bcycles from which R reads data. We define the *readset* of a transaction R, denoted $Read\_Set(R)$, to be the set of items it reads. In particular, $Read\_Set(R)$ is a set of ordered pairs of data items and their values that R 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 [11]. We make no assumptions about transaction management at the server. Since the set of values broadcast during a single bcycle correspond to the same database state, this set is a subset of a consistent database state. Thus, if, for some transaction R, $span(R) = 1$, R is correct. However, since, in general, client transactions read data values from different bcycles, there is no guarantee that the values they read are consistent.

2.2 Invalidation Methods

To ensure the correctness of read-only transactions, we invalidate, e.g., abort, transactions that read data values that correspond to different database states. To determine whether all values in the readset correspond to a single database state, we consider two basic approaches: 1) attaching control information with each data item; we call this method *versioning*, and 2) broadcasting control information periodically; we call this method *invalidation report*.

With the versioning method, a timestamp or version number is broadcast along with the value of each data item. This version number corresponds to the bcycle at the beginning of which the item had the corresponding value. Let v_0 be the bcycle during which a transaction performs its first read. For each subsequent read, we test whether the item read has version number v such that $v \leq v_0$. If the item has a larger version number, the transaction is aborted. Clearly [7]:

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

With the invalidation report method, we broadcast an invalidation report at the beginning of each bcast. The invalidation report includes a list with the data items that have been updated since the previous invalidation report was broadcast. In addition, at each client, a set $RS(R)$ is maintained for each active transaction R that includes all data items that R has read so far. The client tunes in at the beginning of the bcast to read the invalidation reports. A transaction R is aborted if an item $x \in RS(R)$ appears in the invalidation report, i.e., if x is updated. (A possible optimization is to just mark R as *invalid* if one of its $x \in RS(R)$ appears in an invalidation report and abort R only if it tries to read another data item.) It can be shown [8] that:

Theorem 2. *The invalidation report method produces correct read-only transactions.*

In terms of currency, with the versioning method, transaction R reads values that correspond to the database state at the beginning of the bcycle at which R performs its first read operation. With the invalidation report method, R reads values that correspond to the database state at the beginning of the bcycle at which it commits.

3 Multiversion Broadcasting

In the current broadcast schemes, only the last committed value for each data item is broadcast. Instead, in our proposed multiversion scheme, the server maintains and broadcasts multiple versions for each data item. Versions correspond to different values at the beginning of each bcycle and version numbers to the corresponding bcycle.

Let v_0 be the bcycle at which R performs its first read operation. During v_0, R reads the most current versions, that is, the versions with the largest version numbers. In subsequent bcycles, for each data item in its readset, R must read the version with the largest version number v_c smaller than or equal to v_0. If such a version exists in the broadcast, R proceeds, else R is aborted. We call this scheme *multiversioning*. It can be shown [7] that:

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

If invalidation reports are available, we get the following variation of the multiversion method that we call the *multiversioning with invalidation reports* method. Initially, R reads the most current version of each item. Let v_i be the bcycle at which R is invalidated for the first time, i.e., a value that R has read is updated. After v_i, R attempts to read the version with the largest version number v_c such that $v_c < v_i$. If such a version exists in the broadcast, R proceeds, else R is aborted. It can be shown [7] that:

Theorem 4. *The multiversioning with invalidation reports method produces correct read-only transactions.*

In terms of currency, in the multiversion method, R reads values that correspond to the database state at the beginning of the bcycle at which R performs its first read operation (as in the versioning method). In the multiversioning with invalidation reports method, R reads the values that correspond to the database state at the beginning of the bcycle of its first invalidation v_i. Clearly, multiversioning with invalidation reports provides better currency than simple multiversioning, but at the cost of broadcasting invalidation reports.

3.1 Caching

To reduce latency in answering queries, clients may cache items of interest locally. Caching reduces the span of transactions since transactions find data of interest in their local cache and thus need to access the broadcast channel less frequently. We assume that each page, i.e., the unit of caching, corresponds to a bucket, i.e., the unit of broadcast.

In the presence of updates, items in the cache may become stale. There are various approaches to communicating updates to the client caches. Invalidation combined with a form of autoprefetching was shown to perform well in broadcast delivery [2]. In this approach, 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 the cache that appear in the invalidation report. The invalidated pages remain in the cache to be autoprefetched later. In particular, at the next appearance of the invalidated page on the broadcast, the client fetches its new value and replaces the old one. Without loss of generality, we assume this kind of cache updates in this paper. To support multiversioning, items in the cache have version numbers. For reading items from the cache, we perform the same tests regarding their version numbers as when reading items from the broadcast. To ensure that items in the cache are current, the propagation of cache invalidation reports must be at least as frequent as the propagation of invalidation reports for data items. This way, a cached page is either current (i.e., corresponds to the value at the current bcycle) or is marked for autoprefetch.

3.2 Disconnections

In many settings, for example, in the case of clients carrying portable devices and thus seeking to reduce battery power consumption, it is desirable that the clients do not to monitor the broadcast continuously. Further, access to the broadcast may be monetarily expensive and, thus, minimizing access to the broadcast is sought for. Finally, when data are delivered wirelessly, client disconnections are very common. 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. For the reasons above, clients may be forced to miss a number of broadcast cycles.

In general, versioning frees transactions from the requirement of reading invalidation reports. When there are no version numbers associated with data items, a transaction cannot tolerate missing any invalidation reports since there is no other way for it to determine whether an item has been updated. Furthermore, with multiversioning, 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 the bcast. In general, a transaction R with $span(R) = s_R$ can tolerate missing up to $k - s_R$ bcycles in any broadcast with k versions. The tolerance of the multiversion scheme of intermittent connectivity also depends on the rate of updates, i.e., the creation of new versions. For example, if the value of an item does not change during m, $m > k$, bcycles, this value will be available to read-only transactions for more intervals. Note that the multiversion with invalidation reports scheme behaves similar to the invalidation scheme with regard to disconnections. However, it can be made to adopt to disconnections by switching to pure multiversion in anticipation of disconnections.

4 Multiversion Broadcast Organization

To reduce the latency of client transactions, it has been proposed [1] that, instead of broadcasting each item once during a bcast, the frequency of broadcasting an item is determined based on the probability of it being accessed by the clients. Such a schema is called *broadcast disk* organization.

To further describe the broadcast disk organization, we will use an example; for a complete definition of the organization, refer to [1]. In a broadcast disk organization, the items of the broadcast are divided in ranges of similar access probabilities. Each of these ranges is placed on a separate disk. In the example of Fig. 1, buckets of the first disk, $Disk_1$, are broadcast three times as often as those in the second disk, $Disk_2$. To achieve these relative frequencies, the disks are split into smaller equal sized units called chunks; the number of chunks per disk is inversely proportional to the relative frequencies of the disks. In the example, the number of chunks is one (chunk 1) and three (chunks 2a, 2b, and 2c) for $Disk_1$ and $Disk_2$, respectively. Each bcast is generated by broadcasting one chunk from each disk and

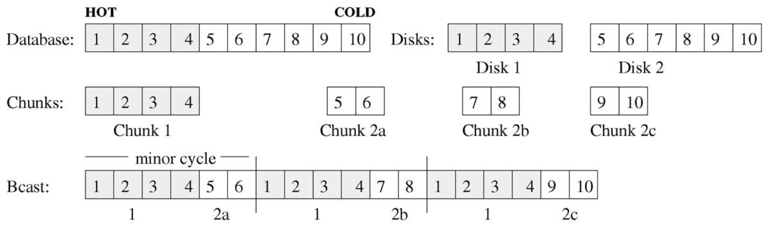


Fig. 1. Broadcast disks.

cycling through all the chunks sequentially over all disks. A *minor cycle* is a subbcycle that consists of one chunk from each disk. In the example of Fig. 1, there are three minor cycles.

We consider three multiversion broadcast disk organization schemes that address two interrelated problems: where in the bcast to place the old versions and how to determine what the optimal frequency of transmitting old versions is.

4.1 Clustering

With the clustering approach, all versions of each item are broadcast successively (Fig. 2b). Thus, older versions of hot items (chunk 1 in Fig. 1 and Fig. 2a) are placed along with the current values of hot items on fast disks, while versions of cold data (chunks 2a, 2b, and 2c in Fig. 1 and Fig. 2a) are placed on slow disks. Consequently, clustering works well when each transaction may access any version of an item with equal probability. The size of each disk, and, thus, the size of its chunks, is increased to accommodate old versions. The number of chunks per disk, however, remains fixed. The overall increase in the size of the bcast depends on how the hot data items are related to the items that are frequently updated. The increase is the largest when the hot items are the most frequently updated ones since their versions are broadcast more frequently during each bcycle.

4.2 Overflow Bucket Pool

With the overflow approach, older versions of items are broadcast at the end of each bcycle. In particular, one or more additional minor cycles at the end of each broadcast is allocated to old versions (Fig. 2c). While, in the clustering approach, the overhead in latency due to the increase in the broadcast size is equally divided among all transactions, in the overflow approach, long-running read-only transactions that read old versions are penalized since they have to wait for the end of the bcast to read such versions. However, transactions that are satisfied with current versions do not suffer from such an increase in latency.

A drawback of this approach is that, by introducing an additional minor cycle, the relative speed of each disk is affected. Another problem is that the space allocated to old versions is fixed; it is a multiple of the size of a minor cycle. To avoid this restriction, older versions can be placed on the slowest disk. In this case, the size of the slowest disk and the size of its chunks are increased to accommodate old versions. Old versions are placed on those chunks of the disk that are broadcast last. Again, the increase of the size of the slowest disk affects the relative speed of the disks.

4.3 Old Versions on New Disk

With the new disk approach, a new disk is created to hold any old versions. The relative frequency of the disks with the current versions is maintained by simply multiplying their frequency by a positive number m so that the slow disk that carries the old versions is m times slower than the disks with the current versions. Take, for instance, the broadcast of Fig. 2. A new disk, $Disk_3$, with six chunks is created for the old versions (Fig. 2d). Current items are broadcast twice ($m = 2$) as frequently as old versions. The relative frequency of the two disks is maintained; items of $Disk_1$ are broadcast three times as frequently as items of $Disk_2$. The resulting bcast is twice the size of the original bcast plus the extra space for the old versions. The new disk approach is easily adaptive. Old versions can be placed on faster disks (by selecting a small m) when there are many long-running transactions and on slower disks (by selecting a large m) when most transactions need current values.

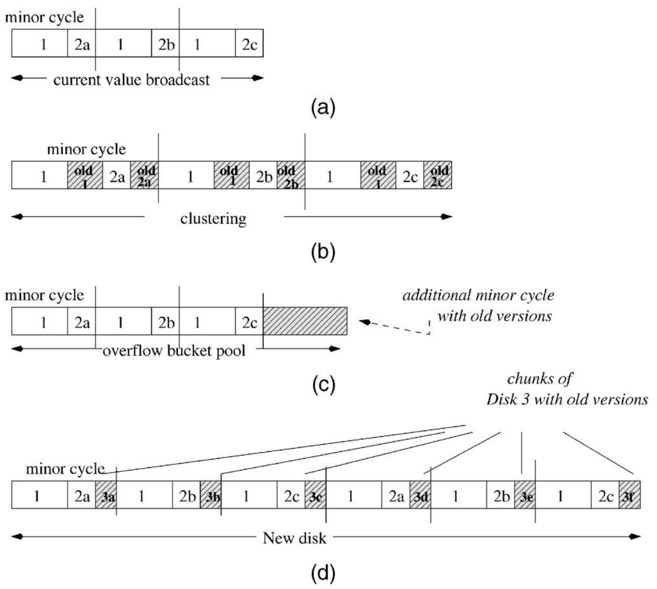


Fig. 2. Multiversion broadcast organization: (a) single version broadcast; (b) multiversion broadcast using clustering; (c) multiversion broadcast with an overflow bucket pull; (d) multiversion broadcast with a new disk.

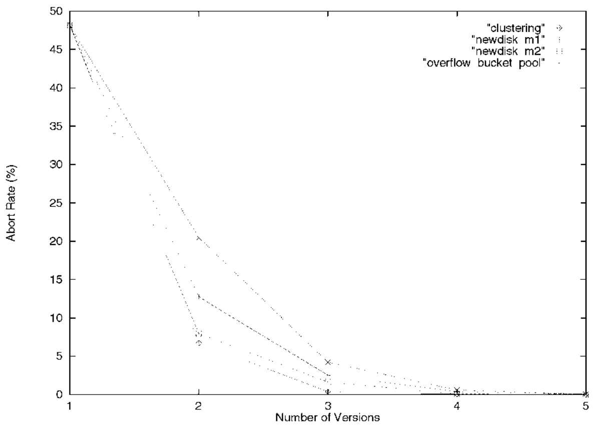
5 Performance Evaluation

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 $NoItems$. We assume a broadcast disk organization with three disks and relative frequencies 5, 3, and 1. The unit of time is set to the time it takes to broadcast a single item. The client accesses items from the range 1 to $ReadRange$, which is a subset of the items broadcast ($ReadRange \leq NoItems$). Within this range, the access probabilities follow a Zipf distribution. The Zipf distribution with a parameter $theta$ is often used to model nonuniform access. It produces access patterns that become increasingly skewed as $theta$ increases. The client waits $ThinkTime$ units and then makes the next read request.

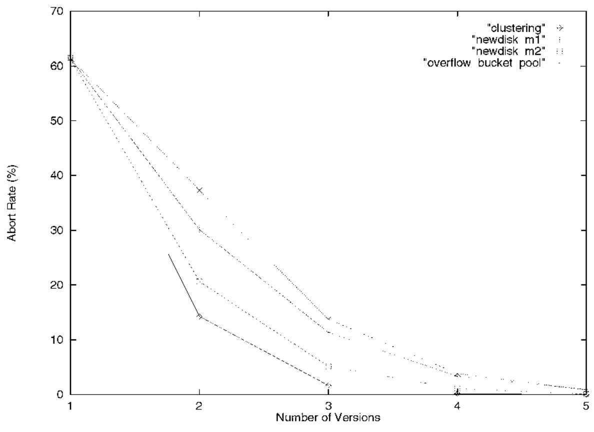
Updates at the server are generated following a Zipf distribution similar to the read access distribution at the client. The update distribution is across the range 1 to $UpdateRange$. An update is generated at the server every $UpdateTime$ units. In the following, for clarity, we express the update rate as a percentage of the items that are updated during each broadcast cycle. This estimation is based on the duration of a single version broadcast. We assume

TABLE 1
Performance Model Parameters

Server Parameters		**Client Parameters**	
No of Items Broadcast	1000	ReadRange (range of client reads)	500
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	1 - 40 (10)
UpdateTime (time between consequtive updates in broadcast units)	varies	S (transaction span)	varies
Broadcast Disk Parameters		**Cache**	
No of disks	3	CacheSize	125
Relative frequency: Disk1, Disk2, Disk3	5, 3, 1	Cache replacement policy	LRU
No of items per range (disk) Range1, Range2, Range3	75, 175, 750	Cache invalidation	invalidation + autoprefetch



(a)



(b)

Fig. 3. Abort rate: (a) 5 percent of the database items are updated per bcast; (b) 10 percent of the database items are updated per bcast.

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 $NoItems$, follow a Zipf distribution, and have zero offset with the update set at the server.

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 L shifts the update distribution L items, making them of less interest to the client. 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 1 summarizes the parameters that describe the operation at the server and the client. Values in parentheses are the default ones.

5.1 Experiment 1: Comparison of the Different Broadcast Organizations

We compare the three different multiversion organization schemes, namely, the clustering, overflow bucket pool, and new disk organizations. For the new disk organization, we set $m = 1$ and $m = 2$. We used multiversion with invalidation reports. In general, when about 5 percent of the database items are updated at the server at each bcycle, by using just one extra version ($k = 2$), all schemes reduce the abort rate from 47 percent (in the case in which only one version, the most current one, is available, i.e., invalidation) to around 6 to 20 percent, depending on the broadcast organization (Fig. 3a). For all organizations, the increase in the broadcast size is well below 20 percent (Fig. 4a). For an update rate at 10 percent, the abort rate is reduced from 60 percent to 15 percent (clustering) (Fig. 3b), while the increase in the broadcast size is below 25 percent (Fig. 4b).

Fig. 3 depicts the abort rate with the number of versions. With the overflow bucket pool approach, transactions have to wait for the end of the broadcast to locate old versions, thus their span increases, as does their probability of abort. Note that, for the new disk organization with $m = 2$, the size of the bcast is effectively double the size of the bcast of the other organizations. For this reason, new values and invalidation reports appear in the broadcast very late (at each new bcycle). Thus, we pay for the increase in concurrency by reading less current data.

Fig. 4 shows the increase in the broadcast size for a different number of versions. For the clustering approach, the increase depends on the offset. The increase is the maximum when the hot items are the most often updated ones ($Offset = 0$), while it is minimum when the frequently updated items are cold and, thus, their versions are placed on slow disks. In all other schemes, the offset does not affect the increase in the broadcast size. Note that, for different offsets, the relative behavior of all methods in terms of the abort rate remains the same. For the new disk approach with

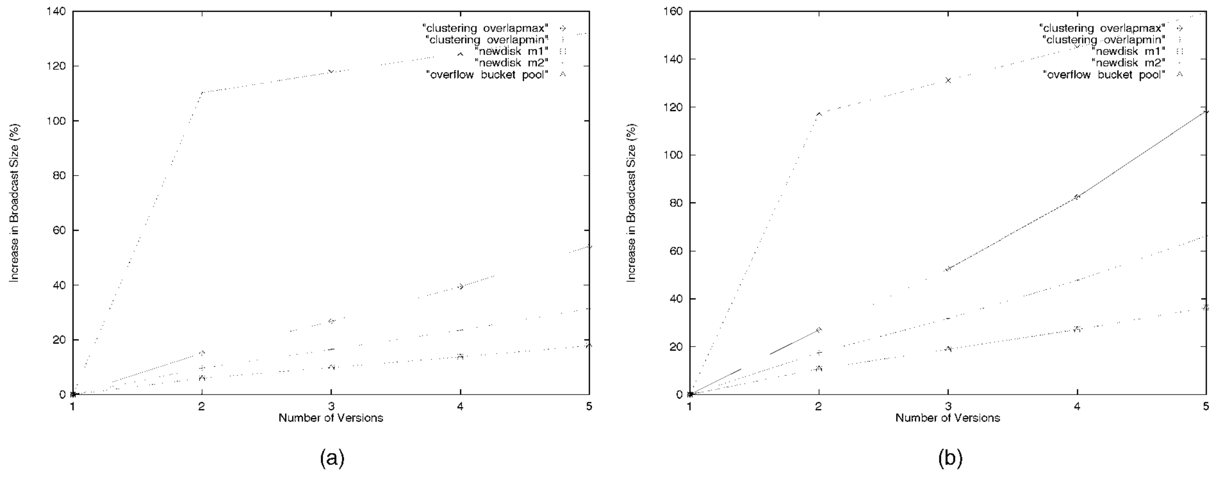


Fig. 4. Increase of the broadcast size: (a) 5 percent of the database items are updated per bcast; (b) 10 percent of the database items are updated per bcast.

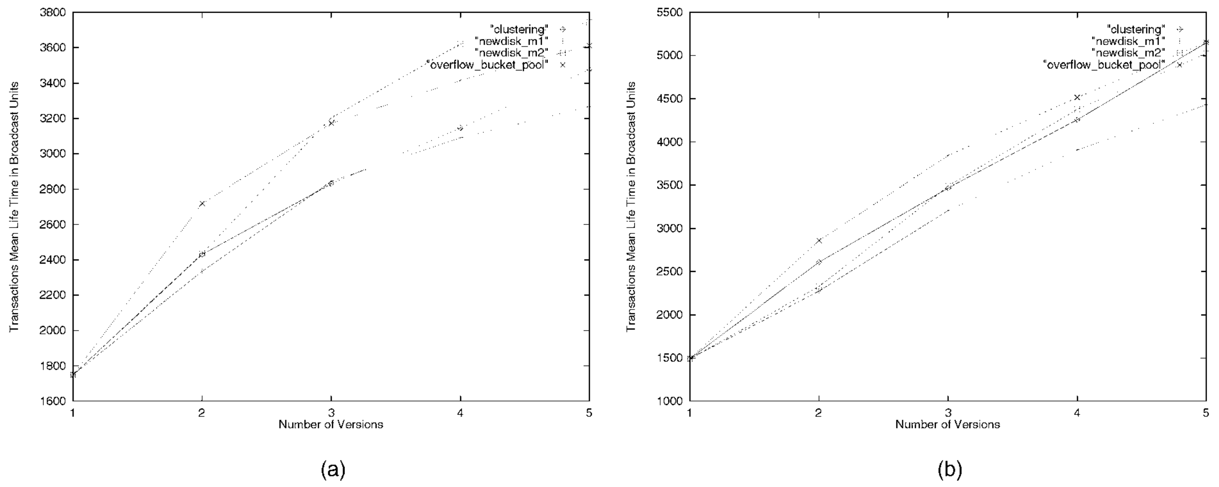


Fig. 5. Mean life time (in broadcast units). Only the response time of *nonaborted* transactions is reported. (a) 5 percent of the database items are updated per bcast; (b) 10 percent of the database items are updated per bcast.

$m = 2$, the size of the broadcast is doubled from the case of a single version. However, the current value of each item appears twice as often as in the single version case; thus, it is as if we had an additional bcycle.

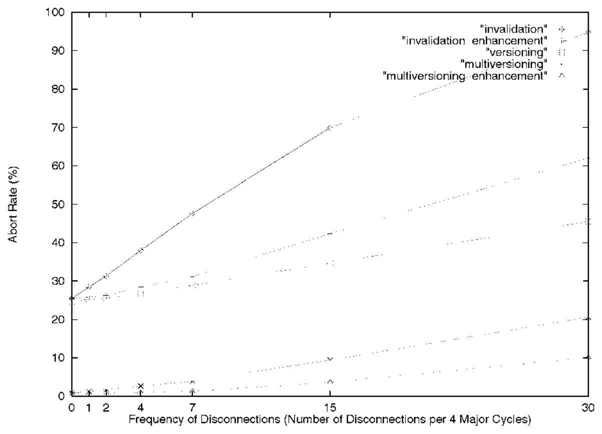
Fig. 5 shows the response time. Note that we report the response time only for the nonaborted transactions, thus the increase in the response time does not mean that transactions get slower, but that even slow transactions are not forced to abort. Although the increase in the broadcast size for the new disk organization with $m = 2$ is large, the mean life time of client transactions is very short (in some cases, even shorter than that for the clustering and the overflow pool organizations).

5.2 Experiment 2: Disconnections

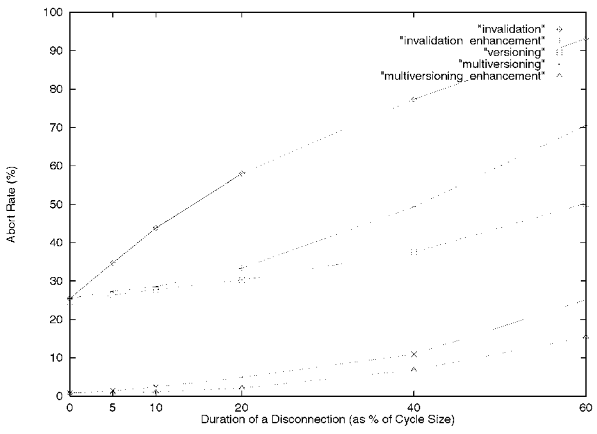
Disconnections are characterized by their duration, which we model using parameter DD, and their frequency, which we model using parameter DF. DD is equal to the number of consequent broadcast units that a disconnection lasts. DF is equal to the number of broadcast units during which a single disconnection of duration DD units occurs. When, exactly, inside DF the disconnection occurs is selected randomly. For example, DD equal to two broadcast units and DF equal to the duration of a minor cycle means that one disconnection lasting two broadcast units occurs in each minor cycle.

In this experiment, we also consider an enhanced version of the protocols: each invalidation report is broadcast twice during each bcycle. Fig. 6 depicts the behavior of invalidation report, versioning, multiversioning (clustering with $k = 2$), and their enhanced versions. We kept the update rate at a low value (around 2 percent of the database items are updated at each bcycle) so that the abort rate is mainly due to disconnections rather than to updates. Fig. 6a shows the performance of the methods for varying disconnection frequencies. Each disconnection has a fixed duration of one minor cycle (around 7 percent of the broadcast cycle). Fig. 6b shows the behavior for disconnections having varying durations and occurring once per bcycle.

Invalidation methods behave poorly in the case of disconnections. This is because, in the case in which an invalidation report is missed, a transaction must abort since it cannot make any assumptions about which items have been updated. On the contrary, versioning and multiversioning are rather tolerant to disconnections. Short disconnections, even if they occur very often (four times per bcycle), result in aborting only 10 percent of the transactions (Fig. 6a). Similarly, even when half of the broadcast content is missed at each bcycle, still around 90 percent of the transactions complete successfully (Fig. 6b). The enhancement (broadcasting the invalidation report twice) improves the performance of both the invalidation and multiversioning.



(a)



(b)

Fig. 6. Disconnections: (a) each disconnection lasts one minor cycle (around 7 percent of the bcycle); (b) one disconnection per bcycle.

6 Conclusions

Data dissemination by broadcast is an important mode for data delivery in data intensive applications. This paper makes three important contributions. First, it proposes the use of multiversions to increase concurrency as well as to support data sequences reflecting multiple server states. Toward this, it suggests and evaluates three different multiversion data broadcast organizations, of which clustering offers the best balance between currency and consistency. Second, the paper introduces two different multiversions schemes with or without invalidation reports. Third, the paper discusses and demonstrates that these schemes work well with autoprefetch caching. Furthermore, by maintaining multiple versions, the tolerance of client transactions of disconnections is increased, as confirmed by our experiments, which is particularly important in the case of mobile data access. The proposed multiversion schemes are scalable in that their performance is independent of the number of clients.

Acknowledgments

The work of Evaggelia Pitoura was supported in part by the European Union through grant IST-2001-32645. The work of Panos K. Chrysanthis was supported in part by the US National Science Foundation under grants IRI-9502091 and IIS-9812532.

References

[1] S. Acharya, R. Alonso, M.J. Franklin, and S. Zdonik, "Broadcast Disks: Data Management for Asymmetric Communications Environments," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 199-210, 1995.

[2] S. Acharya, M.J. Franklin, and S. Zdonik, "Disseminating Updates on Broadcast Disks," *Proc. 22nd Int'l Conf. Very Large Data Bases,* pp. 354-365, 1996.

[3] D. Barbará, "Certification Reports: Supporting Transactions in Wireless Systems," *Proc. IEEE Int'l Conf. Distributed Computing Systems,* 1997.

[4] D. Barbará and T. Imielinski, "Sleepers and Workaholics: Caching Strategies in Mobile Environments," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 1-12, 1994.

[5] T. Bowen, G. Gopal, G. Herman, T. Hickey, K. Lee, W. Mansfield, J. Raitz, and A. Weinrib, "The Datacycle Architecture," *Comm. ACM,* vol. 35, no. 12, pp. 71-81, 1992.

[6] C. Mohan, H. Pirahesh, and R. Lorie, "Efficient and Flexible Methods for Transient Versioning to Avoid Locking by Read-Only Transactions," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 124-133, 1992.

[7] E. Pitoura and P.K. Chrysanthis, "Multiversion Broadcast," extended version, Technical Report, TR-2002-11, Computer Science Dept, Univ. of Ioannina, Apr. 2002.

[8] E. Pitoura and P.K. Chrysanthis, "Scalable Processing of Read-Only Transactions in Broadcast Push," *Proc. 19th IEEE Int'l Conf. Distributed Computing Systems,* 1999.

[9] E. Pitoura and P.K. Chrysanthis, "Exploiting Versions for Handling Updates in Broadcast Disks," *Proc. 25th Int'l Conf. Very Large Data Bases,* pp. 114-125, 1999.

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

[11] R. Rastogi, S. Mehrotra, Y. Breitbart, H.F. Korth, and A. Silberschatz, "On Correctness of Non-Serializable Executions," *Proc. ACM Symp. Principles of Database Systems,* pp. 97-108, 1993.

[12] J. Shanmugasundaram, A. Nithrakashyap, R. Sivasankaran, and K. Ramamritham, "Efficient Concurrency Control for Broadcast Environments," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* 1999.