# The Design and Implementation of a High-Performance Storage Server

—

## Final Report

Stanley B. Zdonik
Brown University

1. ARPA ORDER NUMBER: 8220
2. BAA NUMBER: BAA 90-21
3. CONTRACT/GRANT NUMBER: N0014-91-J-4085
4. AGENT: ONR
5. CONTRACTOR/ORGANIZATION: Brown University
6. SUBCONTRACTORS: none
7. CO-PRINCIPAL INVESTIGATORS: none
8. START DATE: July 1, 1991
9. END DATE: June 30, 1996

DTIC QUALITY INSPECTED 1

19970312 018

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

# 1 Productivity measures

- Refereed papers submitted but not yet published: 0
- Refereed papers published: 6
- Unrefereed reports and articles: 0
- Books or parts thereof submitted but not yet published: 0
- Books or parts thereof published: 1
- Patents filed but not yet granted: 0
- Patents granted (include software copyrights): 0
- Invited presentations: 12
- Contributed presentations: 6
- Honors received: 0
- Prizes or awards received (Nobel, Japan, Turing, etc.): also include descriptions of the specific prizes. 0
- Promotions obtained: 1
- Graduate students supported: 2
- Post-docs supported: 0
- Minorities supported: 0

## 2 Summary of Results

This project investigated the design and iplementation of storage servers to account for costs imposed by current network constraints. The activity had two main focuses. The first was on the design of a storage server that was based on the use of asynchronous protocols and techniques where ever possible to cut down on the message traffic associated with synchrony. The second was on the use of push-based data broadcast to allieviate the problems of high network asymmetry.

In the first part of our study, our objective was to explore the use of asynchronous protocols in the construction of a distributed object storage manager. We have shown that we can achieve better performance than a system that uses highly synchronized techniques such as standard locking. We have also shown that such an approach can facilitate extensibility in the sense that we can decouple basic system components making it easier to add new access methods.

We have implemented a version of our storage server and have tested it under various workloads to experimentally verify our theory. We have shown that we can approach the transaction throughput of commercial object oriented database systems. Since our prototype is not highly tuned, we take this to be good news. Our experiments have also led us to a new cache coherency protocol called invalidate-on-abort that performs quite well.

In the second part of the effort, we have also developed a new technique that we call broadcast disks (BD) that can be used to deliver data to clients in asymmetric communication environments. We have demonstrated that under certain assumptions, this technique can significantly improve performance. We have also shown that this performance gain is most striking when a cost-based caching policy is used. We have designed such a policy as well as an implementable algorithm that approximates the ideal case. This algorithm has been shown to be very competitive.

We have also investigated the use of prefetching in this setting. Since the broadcast schedule is often fixed (or at least quite predictable) and the data is being transmitted anyway. Prefetching is very cheap in this environment. We hae further shown how the broadcast medium is actually quite robust in the face of updates. We have also studied the tradeoffs in invalidation and propagation for the broadcast case.

# 3 Detailed Project Overview

This effort has produced two main sets of results. The first results concern the design of high-performance object servers based on the use of asynchronous protocols to reduce the amount of client/server communication. The second area has to do with using push-based dissemination techniques in a client/server setting to overcome the problems of network asymmetry.

## 3.1 The Design of the BOSS Object Server

BOSS is constructed from loosely coupled entities that cooperate asynchronously through well defined interfaces. Avoiding synchronous operations helps reduce communication overhead. One reason synchronous messages are a problem is that they cannot be bundled. As soon as one is sent the sender must wait until it completes. If asynchronous messages are used, the communications layer can buffer them to produce one larger message. This actually increases the latency for each individual message, but reduces the aggregate cost. The algorithms already tolerate asynchrony, so this small increase in latency is not a problem because overall throughput increases.

BOSS addresses the problem of blocking due to failure with an original recovery algorithm that does not require any undos or redos, so it can recover quickly after a failure while requiring reduced output during commit. Writes are processed in the background or done lazily on demand.

Memory management is critical to the performance of a database system. All of the caches and data stores in BOSS are accessed via a consistent memory management framework. This framework simplifies the construction of new storage classes because the effort optimizing on storage class can easily be shared by another. The framework provides the basic functionality to implement intelligent prefetching and clustering. This topic is discussed further in

BOSS introduces an original recovery algorithm called No-Redo Write Ahead Logging (NR-WAL). NR-WAL does not require the current state of an object to be stored stably, so it does not need to be recovered immediately after a crash. In addition, the algorithm provides better performance characteristics during normal operation because it requires fewer synchronous writes to stable storage.

The stable state of an object is composed of a **base version** and a sequence of operations called a **history**. During normal operation the base version can fall behind the current state. The object is quickly brought up-to-date, by applying the appropriate operations from the history. The core of NR-WAL algorithm is a set of data structures that determine when to apply an operation to an object.

BOSS uses a timestamp based, optimistic concurrency control algorithm. A transaction collects its operations in an *intentions list.* During the transaction the operations are incrementally written to the log. A record in the log is addressed by it's *Log Sequence Number* (LSN). As records are added to the log their LSN's will increase monotonically. Since the LSN of an operation is a monotonically increasing value, it can be used as a timestamp for the purpose of concurrency control. A copy of an object contains it's version number. In this implementation the LSN of the last applied operation which denotes both the time of the last operation and it's location in the log. In order to check for correctness, every operation in the intentions list contains the version of the object it should be applied to. The version "points" directly to the previous operation on the object.

The *Version Table* contains the most recent version for every object. It is the Version Table that indicates which operations need to be applied. At commit, the transaction manager checks the operations in the intentions list against the Version Table. There is no conflict if the transaction read the most recent version of every object. When an object is read by a client, the base version is checked against the entry in the Version Table. The object is up-to-date if they are the same. Otherwise, the Version Table "points" to the operations that need to be applied.

The first and second phases of 2-phase commit modify the tables to indicate their current state. It is worth noting that updates do not lock the entire Version Table or Transaction Table at once. During the prepare phase the intentions list is forced to the log along with a prepare record. A small record is forced to

the log to commit or abort the transaction. Committing an object requires no updates to the base versions of objects. These will be brought up-to-date as needed or eventually by a background process.

Applying an operation to a copy of an object brings that copy up-to-date, but it does not actually modify the base version. It only modifies the copy in stable memory. The base version is finally modified when the buffer manager at the server needs more space. The buffer manager keeps track of which objects have been modified and sends them to stable storage before it discards a volital copy.

Both the history of operations and the base version of an object are stored stably, so they are not affected by a system failure. The Version Table and the Transaction Table must be updated frequently. Without some sort of fast non-volital memory it is not practical to store them stably. Recovery is instantaneous if there is a small amount of non-volital memory to hold these tables.

The intention, prepare, and commit records in the log contain the information necessary to recover the tables after a crash. The tables are checkpointed periodically to reduce recovery time and the compact the log. Several optimizations keep the checkpoints small. There does not need to be an entry for an object who's base version is up-to-date. Likewise, a transaction that is known to have committed or aborted at all sites does not require an entry. When a checkpoint is complete its location is written to a well known location in stable storage.

## 3.2 Broadcast Disks

Having described the notion of data dissemination and outlined the different styles of data delivery that can be employed in a DBIS, we now focus on one particular approach that we have developed, called Broadcast Disks. Using Broadcast Disks, a broadcast program containing all the data items to be disseminated is determined and this program is transmitted in a periodic manner. As stated above, the repetitive nature of the broadcast allows the medium to be used as a type of storage device; clients that need to access a data item from the broadcast can obtain the item at any point in the broadcast schedule that the item is transmitted.

The Broadcast Disk model differs from previous data broadcasting work, however, in that it integrates two main components: 1) a novel "multi-level" structuring mechanism that allows data items to be broadcast non-uniformly, so that bandwidth can be allocated to data items according to the importance of those items; and 2) client-side storage management algorithms for data caching and prefetching that are tailored to the multi-disk broadcast. As is discussed below, these client-side policies differ in significant ways from those that would be used for the other styles of data delivery (e.g., pull-based, or aperiodic push-based).

### 3.2.1 Server-Side Broadcast Scheduling

The multi-level approach to broadcast scheduling used in Broadcast Disks allows data items to be placed on sub-broadcasts of varying size and periodicity. An arbitrarily fine-grained memory hierarchy can be created by constructing a multi-level broadcast such that the highest level is relatively small and repeats relatively frequently, while subsequent levels are progressively larger and repeat progressively less frequently. Such a broadcast program is analogous to a traditional memory hierarchy ranging from small, fast memories (e.g., cache) to larger, slower memories (e.g., disks or near-line storage). By placing data items on faster levels, the access time for higher-priority data can be reduced at the expense of increasing the latency for lower-priority items. For instance, in a highway traffic information system, information on accidents or potential delays can be placed on faster levels, while less time critical information (such as local point-of-interest listings) can be placed on slower levels. In this way, newly arriving motorists can receive data about incidents more quickly than they could if all information was broadcast in a uniform (i.e., flat) manner.

Figure 1 shows a simple example broadcast of three pages arranged on two disks. The page labeled A is on a disk that is spinning twice as fast as the two pages that are labeled B1 and B2. The pages B1 and B2 are said to be on a slower spinning disk that is twice the size of the fast disk. The desirable characteristics for a multi-level broadcast program have been outlined in [Acha95a]. Briefly, a good broadcast program is

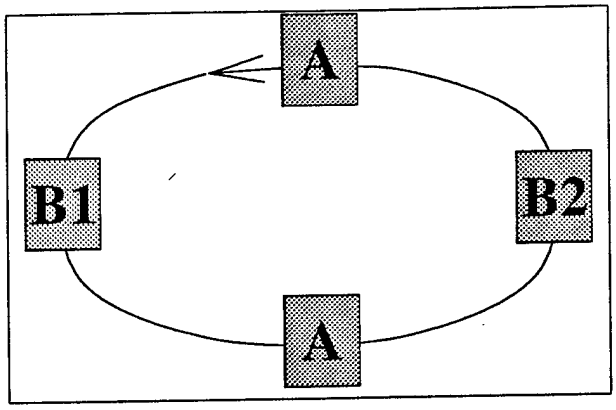


Figure 1: Multidisk with 3 distinct pages on 2 disks

periodic, has fixed (or nearly fixed) inter-arrival times for repeated occurrences of a data item, and allocates bandwidth to items in accordance with (but not in direct proportion to) their access probabilities. Multiple levels are superimposed on a single broadcast channel by interleaving the data items of the various levels in a manner that results in the desired relative frequencies. For example, the multidisk shown in Figure 1 would be generated by repeatedly broadcasting the pattern :"A, B1, A, B2, ...". The algorithm used by the server to generate the broadcast program requires the following inputs: the number of levels , the relative frequency of each level and assignments of data items to the levels on which they are to be broadcast. The algorithm is described in detail in [Acha95a].

### 3.2.2 Profiles, Feedback, and Client-side Storage Management

In order to decide how to structure a broadcast schedule, the server must use its best knowledge and synthesis of the needs of the clients that require disseminated data. There are two potential pitfalls with this approach. First, the server is likely to have inaccurate data about the needs of the clients, as these needs can change quickly and are often hard even for the clients themselves to predict. Second, even given perfect knowledge of the access profiles of the clients, the server must *synthesize* an "average" profile on which to base its broadcast schedule; this average profile is unlikely to be optimal for any *individual* client. A key to solving this problem is the use of client storage resources for caching and prefetching data items. By intelligent management of local storage, a client can (to a large extent) isolate itself from many of the mismatches between its local priorities and the global priorities determined at the server.

We have developed implementable caching and prefetching algorithms to address this issue and have compared them to ideal policies [Acha95a, Acha96a]. The policies take into account both the local access probability (using heuristics such as LRU or usage frequency) and the expected latency of a data item; items that reside on slower levels have higher average latencies. The caching policies [Acha95a] favor slow-level data items, allowing clients to use their local storage resources to retain pages which are important locally, but are not sufficiently popular across all clients to warrant placement on a faster level. Broadcast Disks presents an excellent opportunity for prefetching because objects continually flow past the receivers. We have also developed client *prefetching* algorithms [Acha96a]. These algorithms are more dynamic than the caching policies because they take into consideration the time-to-arrival for items at a given instant. The prefetching policies are able to retain items in a client's cache during those portions of the broadcast cycle where their time-to-arrival is at its highest, which results in a significant performance improvement.

Intelligent client cache management goes a long way towards relaxing the need for strict accuracy in user profile management. The feedback from user requests and status information delivered over a backchannel, however, can provide important information to help maintain the profile information at server. Algorithms for dynamically maintaining profiles and adjusting the broadcast accordingly are thus, important directions

in our current work.

### 3.2.3 Data Consistency

We have extended the mechanisms described above in order to incorporate data item updates and have studied their effect on performance [Acha96b]. As in the previous work, the design considerations divide into client-side and server-side issues. The server must decide how to modify its broadcast program in order to most effectively communicate updates to the client. The server can use both invalidation and propagation. Timely invalidation (i.e., notification of updates) is the minimal requirement in order to enable the preservation of data consistency. The server can also choose to propagate the new values of updated items in order to enhance performance. The client must perform appropriate actions to bring its cache contents back to a good state after an update. For example, the clients can prefetch items that have been invalidated from its cache. We have implemented this *automatic* prefetching and have integrated it with the regular (non-update) prefetching algorithms. Alternatively, a client can choose to ignore an invalidation if it determines that retaining a stale copy of a data item is preferable to having no copy of that item.

Two different consistency models have been studied — immediate and periodic (i.e., based on the period of the broadcast). Our studies have shown that under both models, for low to moderate update rates it is possible to approach the performance of the case in which no updates occur. In other words, the broadcast mechanism can be made quite robust in the face of updates. One reason for this is that when invalidated items are automatically prefetched by clients, the broadcast program itself can act as a propagation medium. Explicitly propagating an item, therefore, is useful only if clients are likely to need to access that item long before it would naturally appear in the broadcast; otherwise, the propagation wastes bandwidth.

# 4 Technology Transfer

- We have formed a partnership with Intel Corporation to investigate the use of the broadcast disk technology to be used as a part of their cable TV software offerrings. It could provide support for products like "CNN at Work" or set-top box management. This work is on-going.
- IBM has also expressed interest and is funding an IBM Fellowship at Brown in this area.
- We have recently been funded to apply some of our broadcast disk technology to the ARPA BADD program. This funding will allow us to transition these techniques to the BADD prototype with influence on the efforts of the BADD prime contractor.
- Brown Object Storage System (BOSS)
  A high-perfomance object server
  Requires: Sun Sparcstation, SunOS 4.1.3
  Contact: John Bazik, Brown University, 401/863-7624, jsb@cs.brown.edu

# 5 Publications

The following is a list of references for the major published works resulting from this effort.

# References


[Acha96b] S. Acharya, M. Franklin, S. Zdonik, "Disseminating Updates on Broadcast Disks", *International Conference on Very Large Databases (VLDB)*, Mumbai, India, September, 1996.

[Acha96a] S. Acharya, M. Franklin, S. Zdonik, "Prefetching from a Broadcast Disk" *12th International Conference on Data Engineering*, New Orleans, LA, February, 1996.

[Acha95a] S. Acharya, R. Alonso, M. Franklin, S. Zdonik, "Broadcast Disks: Data Management for Asymmetric Communications Environments", *Proc. ACM SIGMOD Conf.*, San Jose, CA, May, 1995.

[Acha95b] S. Acharya, M. Franklin, S. Zdonik, "Dissemination-based Data Delivery Using Broadcast Disks", *IEEE Personal Communications*, 2(6), December, 1995.

[Lang95] D. Langworthy, "On the Use of Asynchrony in Achieving Extensibility and High Performance in an Object Storage System", PhD Thesis, Brown University, Dept. of Computer Science, April, 1995.

[Lang94a] D. Langworthy and S. Zdonik, "Extensibility and Asynchrony in the Brown Object Storage System", in *Performance of Concurrency Control Mechanisms in Centralized Database Management Systems*, Prentice Hall, 1994, V. Kumar editor.

[Lang94b] D. Langworthy and S. Zdonik, "Storage Class Extensibility in the Brown Object Storage System", in *Proceedings of the Sixth International Workshop on Persistent Object Systems*, Tarscon, France, 1994.

[Zdon94] S. Zdonik, M. Franklin, R. Alonso, S. Acharya, "Are 'Disks in the Air' Just Pie in the Sky?", *IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, December, 1994.