# Scalable Information Sharing in Large Scale Distributed Systems [1]

Mustaque Ahamad[2] Sumeer Bhola Rammohan Kordale Francisco Torres-Rojas

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332 U.S.A

## 1 Introduction

Many application domains have already demonstrated that they can benefit greatly if efficient access can be provided to shared information across widely distributed users. We use the generic term *object* to describe units of shared information which could include files, web pages or language defined objects. Future applications will require object sharing modes richer than simple browsing. For example, a collaboration system that allows users distributed world-wide (e.g., managers of a multinational company) to interact with each other must manipulate objects that are both read and updated at multiple locations.

Several assumptions, which are natural in the context of large scale systems, can be made about scalable object sharing systems. First, such systems will consist of many server nodes that will act as the storehouses for shared objects. These servers will enable access to objects to a much larger number of client nodes. To avoid high latencies and communication costs, servers will store replicated copies of objects frequently accessed by clients in their vicinity. Furthermore, clients will cache objects to reduce access latency and frequency of communication with servers. Both replication at servers and caching at clients result in multiple copies of an object which introduces the problem of maintaining consistency among the copies.

Many levels of consistency are possible and the choice of a particular consistency level has implications on the programming as well as performance of a distributed application. Although consistency requirements across copies of a single object are easily seen, such requirements can arise between copies of different but related objects. For example, assume that user 1 writes a memo object $o_1$ to produce version $o_{1,1}$ and later generates its updated version $o_{1,2}$. User 2 reads $o_{1,2}$ and writes memo object $o_2$ in response. If user 3 reads memo $o_2$ and wants to read $o_1$ to understand it, it must be provided $o_{1,2}$ and not $o_{1,1}$ which could have been cached by it due to a previous read. Thus, consistency requirements exist across copies of $o_1$ and $o_2$ even when user 3 accesses them in a *read-only* mode.

We first identify some requirements that must be met by scalable object sharing schemes. We present arguments to demonstrate that existing techniques cannot easily be adapted to meet these requirements. This is followed by an outline of our approach.

[1] This work was supported in part by ARPA contract DABT-63-95-C-0125 and NSF grant CDA-9501637.

[2] Contact author – address: College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, email address: mustaq@cc.gatech.edu, phone: (404) 894-2593, fax: (404) 894-9846.

## 2 Scalable Object Sharing

We claim that the following requirements must be met by a scalable object sharing system.

*Replication and Caching:* A server stores objects that are of interest to clients in its vicinity. Since certain objects may be of interest to geographically distributed clients, such objects have to be replicated at widely distributed servers. Replication also enhances availability of objects. Clients must cache frequently used objects to reduce access latency and the frequency of communication with servers which can improve the scalability of servers.

*Consistency:* We claim that consistency requirements that arise due to replication and caching can be met in scalable object sharing systems only by providing more *flexible* consistency levels than the traditional single-copy consistency. Furthermore, the system needs to support multiple levels of consistency which can be changed according to resource availability (e.g., available communication bandwidth) and application needs. For example, in a collaborative environment, authors modifying shared documents may need stronger consistency than reviewers that only read and comment on the documents.

*Object Access and its Cost:* Neither the execution of object operations nor the request to fetch an object should require synchronization and communication among a large set of nodes. In particular, most requests should complete with a client communicating with one or a small number of servers. Furthermore, the communication and processing overheads experienced by a client should only depend on accesses requested by it.

## 3 Related Work

Our goal of flexible consistency levels is similar to systems that provide different types of message ordering as well as replicated data systems that implement different consistency levels. For example, atomic and causal ordering guarantees of multicast messages in the Isis system address consistency needs of different applications [3]. The various session guarantees of Bayou [13] also provides different levels of consistency. Our work is primarily focussed on efficient and scalable implementations of object sharing when both server replication and client caching are employed. Such implementations should allow consistency levels associated with objects to be changed dynamically and an application should be able to share objects at different levels of consistency.

We claim that existing techniques which can be used to build replicated object systems do not meet the requirements enumerated in Section 2. We consider two main approaches that have been extensively investigated. The first one advocates the use of ordered group communication and is exemplified by systems such as Horus, Transis and Isis (see Communications of the ACM, April, 1996, for a survey of group communication and details of these systems). The second one develops protocols for propagating updates to replicated objects which do not rely on ordered group communication. Examples of such systems include Bayou, Ficus [5] and several others (e.g., lazy replication [9]).

Group communication, where nodes having copies of related objects define a group, can be used to send updates to group members. The ordering implemented by the communication system (e.g., causal broadcast) can ensure that nodes access the objects in a consistent manner. There are several problems with such an approach. First, differences in consistency requirements at nodes that store copies of an object cannot easily be handled. For example, group members close to a node where the object is updated may want to receive the update quickly whereas distant members may want to receive it less frequently. One can use separate groups but either ordering benefits are lost or more complex ordering protocols that work across groups have to be employed. The second problem arises due to the fact that sending updates is a *push-based* approach which works well only when the consumers of the new values are well known. Since the set of objects in which a client node is interested can be dynamic, in a push-based protocol a client may be forced to receive and process messages containing updates to objects in its cache even when it no longer needs to access those objects. Finally, ordered group communication requires that messages be received in a given order even when a client node can use a more recent message. As an example, if an object is updated twice in an ordered fashion, it is not incorrect to receive the second update before the first one; if the first update arrives later, the system can detect that it is older and discard it.

Update propagation protocols such as those used in Bayou and Ficus do not rely on ordered group communication and deal with the consistency problem at the object level. However, since these are also primarily *push-based* schemes, they work well only when the set of nodes that maintain object copies is relatively stable. In particular, addition and deletion of new nodes to the set of nodes that maintain copies of related objects is not straightforward. Such changes could be common when clients cache new objects and uncache existing ones as their working sets change. Both Bayou and Ficus use update propagation only among servers and do not address client caching[3]. We feel that for clients, which have dynamic access patterns, a *pull-based* approach where the client is able to dynamically request and cache objects of interest to it is more appropriate. Rover [6] does allow client caching but each object must have a single home server. Furthermore, Rover tries to serialize update accesses to an object and allows applications to handle the cases where conflicting accesses do occur to an object. Although applications can provide different levels of consistency, in our approach, the system provides several well defined consistency levels. The Coda [12] file system makes use of both server replication and client caching. Similar to Rover, when concurrent updates updates are performed to a file, either the system repairs the effects of such updates or application specific consistency can be ensured via user intervention. The isolation-only-transactions in Coda only provide serializability based consistency levels and do not explore weaker consistency levels.

## 4 Our Approach

We address the scalable information sharing problem at the object level. Our system is based on replication at server nodes and caching at client nodes as shown in Figure 1. Each node stores both private objects (P) as well as shared objects (S). We define the *view* of a node at a given time $t$ by values of the objects that are stored by it at $t$. Let the *system view* be defined as the values of all the objects in the system as seen by an external observer who can observe update operations instantaneously. As objects are updated, the system view changes. The system's execution can be characterized by a sequence of views[4]. Let this sequence be $v_0$, $v_1$, $v_2$,...,$v_n$.

Ideally, updates made to an object should propagate instantaneously to all nodes that have

[3]Although a server instance can be run at a client node in Bayou, the assumption that a server stores copies of all related objects can make it difficult to cache objects at resource poor clients.

[4]For simplicity sake, we assume that system views are ordered linearly. In consistency levels based on causality, system views are only partially ordered and we do explore such consistency levels.

copies of that object. In such a system, the view of a node will be consistent with the latest values of the objects. In other words, the view of each node will be contained in the current system view. Unfortunately, protocols that ensure that each node view is contained in the current system view can be very expensive and may require global synchronization prior to accessing shared objects. Such strong synchronization is not required even in implementations of strong consistency.

In our approach, we do not require that node views be contained in the current system view. Thus, node views may lag the current system view subject to certain consistency constraints. The nodes in the system access the objects in a consistent manner if (1) each node view is contained in some system view $v_i$, and (2) as time progresses, a node's view is contained in system views that monotonically increase in currency; in other words, if a node's view changes with the corresponding system views changing from $v_j$ to $v_k$, then $v_k$ is a more recent system view than $v_j$. Thus, a more recent view at a node reflects the execution of additional operations that were not known in the older view. Furthermore, node views advance in a consistent manner because each node view must be contained in some system view. This approach allows us to develop consistency protocols where the results of update operations need not be propagated immediately, or at the same time to all copies of the updated object. Each node that has an old copy of the object can continue to access it because it lives in a view that is prior to the execution of the update operations. We show this in Figure 1 by the degree of shading at each node. The nodes in darker shades have views that are contained in more recent system views. The monotonicity implies that the shading at each node grows darker continually.

We assume a system model where clients cache objects of interest to them. Occasionally, a client will access an object that does not exist in its cache which will result in an object fault. Consider an object fault for object $O$ at client $i$. The object fault is serviced by requesting that a server node supply a copy of $O$. Although the server may supply a newer copy of $O$ than what had been accessed by the client previously, such a copy cannot simply be added to the client cache without performing actions that preserve the desired level of consistency. This is because the object copies existing in the client cache together with the new copy of $O$ may not form a node view that is contained in a more recent system view. A system view that has this new value of $O$ may also have newer values of some of the objects cached at $i$. Such overwritten objects need to be removed from the node cache when $O$ is added to it. We develop the notion of *mutual consistency* (MC) to determine what copies in the cache need to be invalidated because they cannot coexist with the new copy of $O$. Such invalidations are local at the client and do not require communication among nodes. Thus, the consistency maintenance cost does not depend on the number of nodes that may be storing copies of the invalidated objects.

Our work has explored an efficient implementation of the MC mechanism. We call MC a mechanism because it can be used in implementing a range of consistency levels. In what follows, we summarize the novel techniques that we have developed to implement scalable object sharing.

1. Our system allows an application to access shared objects at different levels of consistency. These levels include strong consistency, causal consistency [2] and causal coherency [11]. Strong consistency is similar to sequential consistency in shared memory systems and one-copy serializability in database systems. However, instead of defining consistency at the level of memory operations or transactions, we define it at the level of object operations. Causal coherency provides all the guarantees of causal consistency but also orders update operations

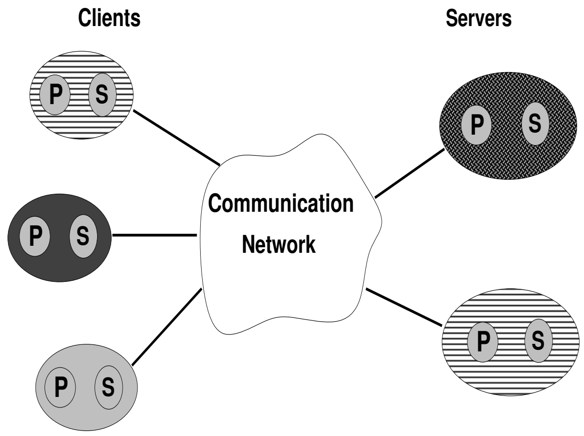


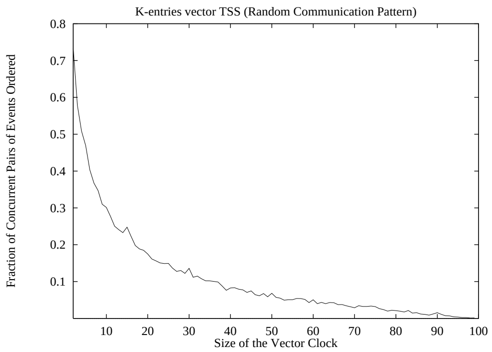

Figure 1: System Model

on a single object the same way in all node views. In contrast, causal consistency permits concurrent update operations to be ordered differently in different node views.

2. Our system supports the multiple consistency levels using the common MC mechanism. We implement MC by associating *lifetimes* with object copies which are read from logical clocks. A lifetime is based on two timestamps stored with each copy: *creation time* of the copy was assigned when it was last written and a *validation time* is the greatest known time when the object copy was found to be not overwritten (the validation time for immutable objects is set to a very high value). Clients can ensure that MC requirements are met for two object copies in their cache by checking that the validation time of the older copy is at least as much as the creation time of the newer copy. Thus, two object copies are mutually consistent and hence can coexist in a node cache if their lifetimes overlap. If lifetimes of two object copies do not overlap when the copies are mutually consistent, their lifetimes can be extended by communicating with nodes that know that more recent updates have not been made to the objects. For example, in strong consistency, a server node is informed when an update operation is performed. Validation time of an object can be advanced by communicating with such a server. More detailed descriptions of MC and its implementations can be found in [1, 7].

3. A client node employs a *pull-based* approach to request an object copy from a server on demand (e.g., when it experiences an object fault). Before this object is added to the client cache, the client ensures that the objects cached at it are consistent with the newly fetched copy. This allows clients to define, enforce and change the level of consistency of objects accessed by them. Also, since the burden of consistency maintenance is not entirely on servers, it improves their scalability.

4. We have developed scalable algorithms to implement multiple consistency levels which include strong consistency, causal consistency and causal coherency [7]. These implementations, including the one for strong consistency, can complete most object accesses (read and write) with no communication or by communicating with a small number of nodes. In particular, communication with all clients that cache copies of an object is not needed.




(a) Compact Vector Clocks

| | |
|---|---|
| Null Sun RPC | 2.35 |
| Null Proxy Invocation | 2.7 |
| Creating a cacheable object | 3.8 |
| Caching an object | 8.0 |
| Method invocation on a cached object | 0.02 |
| Validating a cached object | 6.2 |

(b) Object Caching

Figure 2: System Performance

5. The efficiency and scalability of the consistency algorithms depends on the size of the logical clocks from which creation and validation timestamps are read. Although many of the ordered group communication protocols (e.g., the Isis CBCAST protocol) require timestamps in the size of the participants in communication, the MC technique can be implemented with more compact logical clocks. This is possible because such clocks, which may order some concurrent operations, only impact the performance but not the correctness of the consistency algorithms. We have developed scalable timestamping techniques using constant size vector clocks (traditional vector clocks have a component for each node in the system where an object is accessed). Figure 2(a) shows a simulation based study of a 100 node distributed system that makes use of vector clocks. We vary the number of components (shown on x-axis) and determine the fraction of concurrent operations (y-axis) which are ordered by the clock system. As can be seen, this fraction decreases rapidly which means that vector clocks with much smaller number of components compared to system size can capture most orderings between operations. For example, with vector clocks of size 20 in a 100 node system, unnecessary orderings occur for less than 15% of the concurrent event pairs. Two different implementations of compact clock systems and additional details of the performance studies are given in [14].

6. If a cached object is updated by a client, it is eventually sent to a server (based on cache replacement, consistency or timeliness requirements). The server can use a push-based protocol to propagate the new value to other servers. Parameters such as frequency and message hops for such propagation can be controlled to distinguish between different servers that maintain copies of the object. Although group communication can be exploited when available, ordered communication is not required.

7. We have implemented a scalable distributed object system on top of the Fresco framework which follows the CORBA object model [8]. We have enhanced Fresco by adding client caching and related consistency algorithms. Our preliminary results, summarized in Figure 2(b), show that even in a local area network with a small degree of access locality, the overhead of caching an object can be amortized over less than four method invocations.

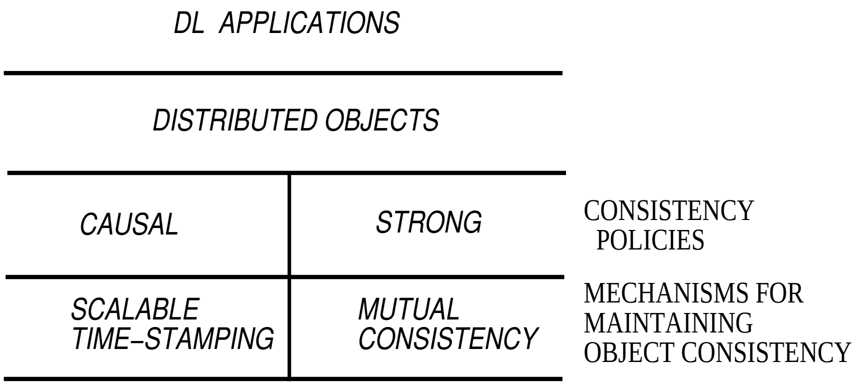



Figure 3: System Architecture

8. The distributed object system is being built to provide system support for the *Distributed Laboratories* (DL) project at Georgia Tech. DL will allow remote scientists to interact with each other as well as remotely executing programs. The distributed user interface and visualization systems will be based on the object system. Several applications that include an atmospheric modeling code as well as large scale distributed simulations will be developed in the context of DL to evaluate the performance benefits made possible by different consistency levels and the scalable implementations that are being developed for them.

The overall architecture of the system is shown in Figure 3. The mechanisms for implementing various consistency levels include compact clocks and mutual consistency. These are used to implement a range of consistency policies. A consistency policy refers to a particular implementation of some consistency level. For example, we have developed two implementations of strong consistency, one based on the MC technique and another one that manages information to locally invalidate exactly those cached objects that have been overwritten. The object system we have developed allows a distributed object to be subclassed either from a proxy class which uses remote method invocations, or from a class corresponding to a consistency policy. The latter enables caching for the object and the system automatically provides the level of consistency that corresponds to the consistency policy. Finally, the distributed laboratories subsystems will make use of the distributed object framework defined by the consistency policies.

## 5 Concluding Remarks

We have developed a system that provides scalable object sharing by using server replication and aggressive client caching. It implements pull-based mechanisms that allow client nodes to cache objects on demand and enforce needed levels of consistency which can be changed depending on resource availability and application requirements. In contrast to this pull-based approach, servers can propagate updates to objects among themselves using a push-based approach. Such a hybrid scheme naturally accommodates dynamic object working sets at client nodes and allows differentiation among servers according the needs of clients serviced by them. The MC technique allows a node to detect what object copies in a node can exist together. It provides a common mechanism for implementing a range of consistency policies.

## References


[1] M. Ahamad, F. Torres-Rojas, R. Kordale, J. Singh and S. Smith. Detecting Mutual Consistency of Shared Objects. International Workshop on Mobile Systems and Applications, December 1994

[2] M. Ahamad, G. Neiger, P. Kohli, J. E. Burns, and P. W. Hutto. Causal memory: Definitions, implementation and programming. *Distributed Computing*, 1995.

[3] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM TOCS*, (9, 3):272–314, 1991.

[4] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *ACM Symposium on Principles of Distributed Computing*, 1987.

[5] R. G. Guy, J.S. Heidemann, W. Mak, T. W. Page, G. J. Popek, and D. Rothmeier. Implementation of the ficus replicated file system. In *USENIX Conference Proceeding*, June 1990.

[6] A. D. Joseph, A. F. deLespinasse, J. A. Tauber, D. K. Gifford and M. F. Kaashoek. Rover: A toolkit for mobile information access. Fifteenth ACM Symposium on Operating Systems Principles, December 1995.

[7] R. Kordale and M. Ahamad. A Scalable Technique for Implementing Multiple Consistency Levels for Distributed Objects. Proc. of International Conference on Distributed Computing Systems, May 1996.

[8] R. Kordale, M. Ahamad and M. Devarakonda. Object Caching in a CORBA Compliant System. Proc. of Second Conference on Object-Oriented Technologies and Systems. June 1996.

[9] R. Ladin, B. Liskov, L. Shira, and Ghemawat S. Providing high availability using lazy replication. *ACM TOCS*, 10(4), November 1992.

[10] Q. Lu and M. Satyanarayanan. Isolation-only transactions for mobile computing. *ACM Operating Systems Review*, 28(2), April 1994.

[11] M. Raynal, G. Thiakime and M. Ahamad. From serializable to causal transactions. Proc. of ACM Symposium on Principles of Distributed Computing (short version). May 1996.

[12] J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *ACM Symp. on Oper. Sys. Principles*, 1991.

[13] D. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. Welch. Session guarantees for weakly consistent replicated data. In *Proc. of International Conference on Parallel and Distributed Data Systems*, September 1994.

[14] F. Torres-Rojas and M. Ahamad. Plausible Clocks: Constant size logical clocks for distributed systems. To appear in Proc. of 10th Workshop on Distributed Algorithms (WDAG), Bologna, Italy. October, 1996.