Detecting Mutual Consistency of Shared Objects*

Mustaque Ahamad
Francisco Jose Torres-Rojas
Rammohan Kordale
Jasjit Singh

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

Shawn Smith

Department of Computer Science
Rice University
Houston, TX 77005, USA

Abstract

In systems that support disconnected operations, shared objects must be cached at client sites to ensure that the objects are accessible while the client is disconnected. It is desirable that objects cached at a client site are mutually consistent with respect to the level of consistency provided by the system. We present a new technique for detecting mutual consistency of objects and show how it can be used to support causal consistency, which is appropriate in systems that support disconnected operations.

1 Introduction

In distributed systems, objects stored in shared repositories need to be cached at sites where they are frequently accessed to combat high latency and to reduce communication costs. Such caching is essential in systems that support disconnected operation because a client cannot access objects from shared servers during the periods when it is disconnected. The Coda file system [8] is an example of a system where client caching is used both to improve performance and also to support disconnected operations.

Client caching results in multiple copies of an object which leads to the problem of mutual consistency among the copies. Mutual consistency requirements also arise between copies of different objects. For example, an author writing a paper can create two files: file f_1 includes results of an experiment and the discussion of these results is contained in file f_2. Assume that the author first updates f_1 to include new results and then modifies the discussion in file f_2. Clearly, we want to detect and avoid situations where a client site has the new copy of f_2 but an old copy of f_1. Intuitively, the cached copies of f_1 and f_2 are not mutually consistent because f_2 contains discussion of results that are not reflected in the old copy of f_1 that is cached at the client. In fact, if the files are read from and written to a central server, it would not be possible for a client to access the new version of f_2 and the old version of f_1.

The mutual consistency problem has been addressed for copies of a single object. For example, Coda makes use of *version vectors* that were first introduced by Parker et. al. for maintaining file consistency in Locus [11]. Coda detects concurrent updates to a file that can lead to inconsistency. It either repairs the effects of mutual inconsistency between file copies or notifies the file owner. The consistency problem for a single object is relatively simple because the object should have a unique value at any given time and concurrent updates lead to multiple values. For a group of objects, additional consistency requirements need to be defined to determine if their copies are mutually consistent.

We consider a system where a set of distributed clients access objects from a logically shared object space. These objects are stored at one or more server nodes that provide permanent storage for them. A client node caches objects that it has accessed recently. Since objects can be cached and updated, a key issue is the order in which clients see the updates made to shared objects. Strong consistency, such as serializability which requires that the results of all accesses are equivalent to their execution in some serial order, cannot be provided in systems where clients are temporarily disconnected. This is because communication between nodes is required before an access can be completed in cases where objects are cached at two or

*This work was supported in part by National Science Foundation grant CCR-9106627 and ARPA contract N00174-93-K-0105. To appear in Proceedings of Workshop on Mobile Computing Systems and Applications, December 1994.

more nodes. Weak consistency guarantees are appropriate when completion of an access to a cached object should not block for communication to complete. We define *causal consistency* (CC), which is a weakly consistent shared data model. CC is based on causal orderings [10] between accesses to shared objects.

We claim that CC is appropriate in systems that support disconnected operation. CC allows both read and write operations to complete with a cached copy. In strong consistency, a write can return only after all copies of the object being written are updated or invalidated[1], which requires communication to complete before the write is done. Thus, strong consistency cannot be provided when there are disconnected operations. CC can not only be provided in such an environment but it also captures causal relationships among operations that access the objects. The latter allows it to meet the consistency requirements of many applications where asynchronous entities interact by accessing shared objects (e.g., computer supported cooperative work (CSCW) applications).

Simple techniques such as version vectors used in Coda cannot be used to detect mutual consistency of different objects when these objects are accessed independently [5]. In this paper, we present a new technique for detecting mutual consistency of cached copies of a set of objects. This technique also allows us to detect concurrent updates to the copies of a single object. Our technique makes use of vector clocks [6] which capture causal orderings between events in a distributed system. In particular, we define the notion of a *lifetime* for a value of an object. The lifetime of a certain value of object f is the duration defined by two logical times that are read from vector clocks: t_f^c is the vector time when this value was created and t_f^v is a vector time up to which this value of the object is known to be *valid*. For causal consistency, copies of two objects f_1 and f_2 are mutually consistent if they store values that have overlapping lifetimes. On the other hand, if the values stored in two copies of the same object have overlapping lifetimes, they are written concurrently and it may be necessary to reconcile these copies.

In this paper, we present preliminary results of our work on mutual consistency requirements for shared objects that are appropriate in systems that support disconnected operation. We define mutual consistency based on causality, and causal consistency in Section 2. Section 3 outlines an implementation that ensures mutual consistency among cached objects. We discuss related work in Section 4 and the paper is concluded in Section 5.

[1] In many replicated data management protocols, only a majority of the copies can be written but that does not allow reads to be executed locally.

2 Causal Consistency

Causal consistency (CC) is defined based on causality that orders an operation o_1 before another operation o_2 if o_1 can potentially have an effect on the execution of o_2. Assume that we want to provide CC for a set of objects which we denote by $\mathcal{S}$. We distinguish object operations as *read* and *write* based on if they just read the object or the object state is also updated. A causal order can be defined between operations that access the objects in $\mathcal{S}$ as is done in message passing systems [10]. In particular, we say that for two operations o_1 and o_2, which are executed on objects in $\mathcal{S}$, o_1 causally precedes o_2 if one of the following holds:

1. o_1 and o_2 are executed by the same process and o_1 is executed before o_2.
2. o_2 reads an object value that was written by o_1.
3. There is a sequence of operation $o_{i,1},\ o_{i,2}, ..., o_{i,m}$ such that $o_{i,1} = o_1$, $o_{i,m} = o_2$, and for all $1 \leq j < m$, $o_{i,j}$ and $o_{i,j+1}$ are related by either (1) or (2).

If o_1 and o_2 do not causally precede each other, these operations are called *concurrent*. Clearly, concurrent operations cannot effect each other.

CC can now be defined as follows. For two objects f_1 and f_2 in $\mathcal{S}$, assume that their values cached at a site were written by operations o_1 and o_2, respectively (when the value is produced by a sequence of write operations, consider the last operation that wrote the value). The cached copies of f_1 and f_2 are mutually consistent[2] if one of the following holds.

1. o_1 and o_2 are concurrent and hence neither causally precedes the other.
2. The two operations are causally ordered. Without loss of generality, assume o_1 precedes o_2. There cannot be another operation o' such that o_1 precedes o', o' precedes o_2, and o' writes a value to object f_1 that is not present in the cached copy of f_1.

A set of objects are mutually consistent if each pair of them is mutually consistent. A system provides CC if for each client, the cached object copies at the client are mutually consistent at all times. In other

[2] This definition for mutual consistency is developed for CC. For other consistency conditions, mutual consistency will be defined differently according to their requirements.

words, clients only access objects that are mutually consistent with respect to CC. Since we define CC for objects cached at a site at a given time and not objects that may have been previously used at the site, it is the responsibility of an application that it caches all related objects before one is accessed. This is natural if the client is going to be disconnected because objects cannot be fetched on demand (see additional discussion in Section 3.2).

To illustrate CC, consider an example where there are two file objects: f_1 and f_2. Site 1 first writes f_1, producing its copy f_1^0. It again writes this file which results in a new copy which we denote by f_1^1. Site 2 now reads f_1^1 and writes a second file f_2. Let the copy of f_2 produced by site 2 be f_2^0. If site 3 caches f_1^0 and f_2^0, copies of the two files are not mutually consistent at site 3. This is because f_1 has been written again and this write operation causally follows the operation that produced f_1^0, and precedes the operation that wrote f_2 to produce f_2^0. On the other hand, if site 3 caches f_1^1 and f_2^0, the two copies are mutually consistent because no intervening write to f_1 exists.

CC is weaker than one-copy serializability (1SR) which ensures that the results of executing a set of accesses to shared objects are equivalent to their execution in some serial order with a single copy of each object. This can be easily seen from an example. Again, consider two sites that both cache mutually consistent copies of objects f_1 and f_2. Also, assume that the values of these objects are v_0^0 and v_1^0 respectively. Site 1 now writes object f_2 and then reads object f_1. Similarly, site 2 writes object f_1 and then reads f_2. CC allows the reads to return the old values, v_0^0 and v_1^0. This is permitted because the values of the objects remain mutually consistent at the sites even after the writes. On the other hand, 1SR cannot allow the two reads to return the old values because no serial order of the write and read operations is possible in which both reads can return the old values. It can be easily shown that 1SR is strictly stronger than CC. This follows from the fact that the equivalent serial order that demonstrates 1SR also respects the causal order between the operations. Actually, systematic weakening of consistency requirements is what makes CC appropriate for systems with disconnected operations because it can allow both read and write operations to complete with cached copies of shared objects. Both reads and writes cannot be completed at a disconnected site when serializability has to be ensured.

CC is closely related to causal memory which was developed by us earlier [1], but there are differences in the way the two are formulated. For example, causal memory defines a value for a shared location at a certain point in a process's execution based on all preceding memory operations whereas CC defines mutual consistency requirements among the copies of a set of objects that exist in a cache at a certain time. When a client's cache only stores a subset of the shared objects, CC could allow executions that are not permitted by causal memory.

3 Implementing Causal Consistency

We consider the implementation of CC in a system where servers store shared objects which are accessed at client sites. Clients complete their access to shared objects by caching them locally. Copies of objects cached at a client can be freely accessed by the processes executing at the client. An object copy that is updated at a client can be sent back to the server which can supply it to other clients. Thus, there is no direct communication between client sites. A client is forced to communicate with a server only when it wants to access an object that is not locally cached. Objects may be replicated at several servers for higher availability but we do not consider that in the simple implementation that follows. Furthermore, in this implementation, we assume that point-to-point messages are delivered reliably and in first-in-first-out (FIFO) order.

Since we want a client to be able to access cached objects without communicating with other sites, it is necessary that objects cached at the client be mutually consistent. To ensure mutual consistency, we need to detect that when two cached copies are produced by causally ordered operations, there is no intervening write that could make the copies inconsistent according to CC. In other words, we need to maintain causality information about the operations that produced cached copies. Timestamps derived from vector clocks [6] can be used to precisely capture the causal relationships between operations in a distributed environment. When an object is written, the copy can be assigned a vector timestamp to capture the logical time at which the object copy was generated. However, with this scheme, cached copies of two objects will be guaranteed to be mutually consistent only if they have timestamps that are not comparable (neither timestamp is smaller than the other). If the timestamps are ordered, one cannot determine if the two copies are mutually consistent or not. As a result, it has to be assumed that they are potentially inconsistent. In this case, either one of the copy has to be invalidated or a validation step needs to be executed to ensure that the two copies are indeed consistent. Since the timestamp associated with a copy does not change even when the copy is validated, it may be necessary to validate the copy again in the future when it is cached again. Thus, only using vector timestamps that are associated with objects when their copies are written does not provide an efficient means of detecting mutual consistency of long lived object copies.

3.1 Object Lifetimes and Mutual Consistency

We provide CC for objects cached at a site by associating a pair of vector timestamps with each copy of a shared object. These timestamps, called *creation time* and *validation time* define a lifetime for a particular value of an object. The timestamps are assigned from vector clocks which are used to capture the causal orderings between events in distributed systems [6]. Let $\mathcal{C}$ be the set of client sites where objects in set $\mathcal{S}$ are accessed. The vector clock, maintained at each site in $\mathcal{C}$, is a vector of integers, one for each site in $\mathcal{C}$. We denote the clock at site i by VT_i. When an object is created or updated at site i, VT_i is incremented by adding 1 to its i^{th} component. VT_i can also be advanced with respect to a timestamp t by assigning to VT_i the component-wise maximum of t and the current value of VT_i. We can also compare vector timestamps. t_1 is less than t_2 if each component of t_1 is less or equal to the corresponding component of t_2 and at least one component of t_1 is strictly less than the corresponding component of t_2. Two timestamps are not comparable when neither is less than the other. In this case, we call the two timestamps concurrent because they correspond to events that are not causally ordered. Since servers provide permanent storage for objects, they also maintain creation and validation times for objects stored at them. However, there are no components in the vector clocks for servers because we assume that objects are read and updated only at the clients.

The operation of the system is as follows. A client can freely read and update objects cached at its site. If object f is updated at client i, VT_i is incremented and this clock value is assigned as the creation time of f. When a client sends its copy of f to the server, the current value of VT_i is assigned as the validation time of the object copy being sent to the server. Although we do not require that a client send updates to the server at a certain point in its execution, for simplicity, we do require that updates are sent to the server in the order of creation times. Thus, if two objects f and g cached at a client are updated in the order f and then g, we require that the copy of f be sent to the server before g is sent. This does not require that the two objects be replaced from the client cache in the same order. The client can replace g before f but it must send the current copy of f to the server before it replaces g.

The interesting part of the scheme is when there is a miss in a client cache for object f. The client site must communicate with the server to request a copy of the object. The server sends the copy it stores along with its creation and validation times. On receiving this message, the client advances its clock by doing an update with the creation time of the received copy. Furthermore, it checks if the existing object copies in the cache are consistent with the incoming copy of f. This is done by checking the following for each cached object g:

1. The creation times of f and g are not comparable. In other words, the creation time of f is not less than g and the same holds for g and f. Thus, the write operations that produce the copies of f and g are not causally related and the copies are mutually consistent.

2. If (1) does not hold, the lifetimes of f and g are compared. If they overlap, the two copies are mutually consistent. The lifetimes of f and g overlap if the component-wise maximum of the creation times of f and g is less than or equal to the component-wise minimum of their validation times. Mutual consistency of f and g follows in this case because the copy of f in the cache is valid at least up to the logical time when the copy of g was produced. Thus, there cannot be an intervening write between the operations that produced the cached copies of f and g.

3. If neither (1) nor (2) hold, the following is done. Since (1) does not hold, one of the copy has a creation time which is less than the creation time of the other. In this case, the older copy has to be validated. The client requests the server which communicates with clients that cache the object to determine if the copy is valid. If it is, the validation time for the object copy is advanced. Otherwise, a new copy is supplied to the client that suspected mutual inconsistency.

The third case in which validation is required needs more careful examination. There are many different ways in which such an invalidation can be done. We only sketch one possible way. We are validating the object copy which has an older creation time. Let this object be f and assume that its copy is not mutually consistent with a copy of object g. Furthermore, assume that t_f^c and t_g^c, respectively, are the creation times of the copies of f and g that are in the local cache. The validation step either guarantees that no other writes have taken place to f between t_f^c and t_g^c or it results in the caching of a new copy of f. In the first case, the server must determine that there is no write operation that generates a new value of f with a timestamp that is greater than t_f^c and smaller than t_g^c. If the server already has a copy of f that has a creation time that is greater than t_f^c, it can supply this new copy to the client which can check if it is mutually consistent with the copy of g. If not, then the server requests that all clients which cache copies of f return the creation times of the copies that they store. If t_f^c is the creation time of all copies, no writes are done to f between times t_f^c and t_g^c. In this case, the validation time of f at the client where the inconsistency was detected is advanced to t_g^c. This makes the lifetimes of the copies of f and g overlapping and as a result, the two objects become mutually consistent. If the server detects a write to f with creation time that is between t_f^c and t_g^c, the validation process must be executed again with this new copy. If cached copy of the object has a creation time that is concurrent with t_g^c, this copy can be supplied since it is mutually consistent with g.

The fact that clients access an object asynchronously further complicates the validation process. No write between times t_f^c and t_g^c when the validation is carried out does not imply that there will be no such write in the future unless clocks at all clients that cache f are advanced beyond t_g^c. If this is not done, the current clock value of a client could be between t_f^c and t_g^c and this client could write f after the validation completes. We avoid such anomalies by advancing a client's clock to the maximum of its current value and t_g^c. This may result in ordering operations that are really concurrent but this does not violate correctness.

The above validation process can also be used to advance the validation times of f at all clients. This could reduce the possibility that other clients also have to perform a validation when they cache a new object. Also, the frequency of the validation process can be reduced if validation times are advanced in the background. This can be done periodically by the server.

Validation of an object copy requires that a server communicate with other clients. It may not be possible to communicate with all such clients because they may be disconnected. In such a case, either g can be removed from the cache or it can be marked as potentially inconsistent. Users can be informed of such potential inconsistency. Also, the same object can be updated concurrently at multiple clients. The server can detect such updates because the creation times of the copies will not be comparable (similar to version vectors in Coda). Thus, users can also be informed of such concurrent updates to a single object which are allowed by CC.

3.2 Discussion

CC is provided by the above implementation if all objects in a client cache are mutually consistent at any given time. Cached object copies remain mutually consistent when the client reads or updates locally cached copies. This is because no new orderings are created with other nodes and the client sees the changes made by it in the cache immediately. Thus, mutual consistency only needs to be checked when a client experiences a miss and brings in a new object from a server. The three step consistency detection algorithm that we present in the previous subsection is executed at this point to ensure that the new object copy is consistent with copies of the existing objects. Assuming f is the new object copy and g is an existing copy, mutual consistency follows in the first case because the copies have concurrent creation times and hence there cannot be a write to f that has a timestamp greater than t_f^c and less than t_g^c (a write to f that is ordered between the writes that produced the cached copies will have such a timestamp). In the second case, it follows from the fact that the older object copy (f which has a smaller creation timestamp) is valid until the creation time of the copy of g. In the last case, either a different copy of f is found and cached at the client or it is guaranteed that a more recent write to f does not exist.

The assumptions we make about the communication subsystem and the sending of updates to the server are necessary to ensure CC. They guarantee that if the client's copy reflects updates to an object up to some logical time, we know that all causally preceding updates are reflected in the copies stored at the server. This follows from the fact that clients communicate only via the server and updates are sent by clients to the server in creation timestamp order using FIFO channels.

We assumed that the mutual consistency of an incoming object be checked with all existing objects in the cache. This may be too conservative when the cache contains copies of unrelated objects. The system can provide a mechanism to specify what objects are related. In this case, when an object copy is received at a client, its mutual consistency need to be checked with only those cached copies that are related to it.

Our implementation also provides natural consistency guarantees that are not required by CC. For example, consider the extreme case where the client cache only holds a single object copy at any time[3]. Thus, a client replaces the object in its cache before it brings a new object. In this case, there are no mutual consistency requirements since the cache is empty when a new object is brought into it. However, because of the restrictions imposed on the order in which updates are sent from clients to a server and the fact that clients only get objects from a server, it can be easily shown that a client will never cache a copy that is causally overwritten by a copy that it cached earlier.

Many questions arise about how well our mutual consistency detection scheme works. First, the size of the vector timestamps which have a component for each client could be large when objects are shared between many clients. Also, an implementation has to make a choice between invalidating or validating locally cached objects that are not mutually consistent with an incoming object. We prefer invalidations because they can be done locally. However, which approach will provide better performance is a question that cannot be easily answered. Similarly, we are making an assumption that reads are far more frequent than writes. Thus, creation time will remain unchanged for long durations and advancing the validation time will allow us to avoid invalidations when object copies are mutually consistent.

We are currently building a system that will be used to answer such questions. This system is based on the Fresco toolkit [13]. Fresco provides an object wrapper around the Sun Remote Procedure Call facility but has no provisions for caching object copies. Our system allows processes executing at different nodes to map shared objects into their address spaces. Causal consistency for cached copies of these objects is ensured by the underlying system using the scheme that we discussed in this Section. In the near future, we expect to have experimental results that will allow us to evaluate the effectiveness of this scheme.

4 Related Work

We have argued that CC is the appropriate level of consistency for shared objects in systems that support disconnected operations. CC provides a weakly consistent system. Such systems have been investigated in many domains that range from distributed file systems to replicated databases. For example, the Sun Network File System (NFS), does not guarantee consistency when a file being accessed at one node is concurrently written at another node. Coda detects inconsistency for a single file but does not provide consistency across a group of files. This is because if a user changes file f_1 and then file f_2, another client that reads the new value of f_2 is not guaranteed to see the changes made to f_1. Although providing mutual consistency between different files was not a goal of Coda, the version vectors used by it cannot be easily extended if such consistency is desired [5].

Recently the Bayou system has proposed several weak orderings that are desirable in systems where shared data is replicated at servers [12]. Some of these orderings are weaker than CC (e.g., monotonic reads). CC is similar to a combination of writes follow reads and monotonic writes. However, it is not identical to the consistency in the Bayou system with these combined orders because CC does not imply eventual convergence. Version vectors are also used in the implementation of the the various orders but the entire shared database is treated as a single logical object. As a result, the values of the version vectors become highly conservative estimates of the ordering information. In our approach, this could lead to unnecessary invalidations of valid cached copies because updates to other objects will increment the vector timestamps which will make the cached copies appear to be potentially overwritten. Also, the Bayou system follows an update type scheme to propagate changes to servers. This is appropriate when servers maintain copies of all shared objects but its benefits are not clear when clients cache objects.

Causality, from which CC is derived, has been investigated extensively. In our work on causal memory [1], we have developed implementations that are appropriate when data is shared between parallel applications [7]. These implementations either suffer from unnecessary invalidations or require that the size of the timestamp be linear in the number of shared objects which could lead to excessive communication and storage overhead. Since these implementations use the timestamp that is assigned when an object is written and this timestamp does not change until the object is written again, a client may invalidate the copy of an object several times as the vector times advance due to writes to other objects. The validation time proposed in this paper can be advanced as we validate the consistency of an object again and hence can reduce the frequency of unnecessary invalidations.

The CBCAST communication primitive of the ISIS system provides causally ordered message delivery [3]. It is possible to update the state of replicated servers using such a communication primitive. However, due

[3] In defining CC, we assumed that an application will cache objects needed by it before it accesses them.

to client caching, the set of clients that have copies of particular object could be dynamic which would require changes in group membership. Furthermore, since clients can read objects in an order that is different from the causal order in which the objects are written, we allow a client to access an object even when causally preceding updates have not been received at the node. Consistency is ensured due to local invalidations which disallow access to causally overwritten copies.

Weak consistency has also been proposed in other distributed data systems [2, 9]. Our approach differs from these systems because we rely on detection and invalidation (or validation) of potentially causally overwritten object copies instead of update propagation. We feel that our approach is better in system where the set of clients that cache an object is dynamic and any particular client accesses only a small fraction of the shared object space.

5 Conclusions

We have introduced causal consistency for shared objects which captures natural consistency requirements for shared objects in many environments. It is specially desirable in cooperative environments but can also be used in distributed file systems to provide enhanced consistency. We feel that CC is appropriate for systems with disconnected operations because it allows clients to locally complete read and write operations on cached objects. The overlapping lifetimes based technique we present for detecting mutual consistency of objects could provide better performance than earlier techniques which either work only for a single object or make use of only the creation time of object copies. When updates are few compared to reads, advancing the validation time for a copy could permit many reads to complete without requiring any communication with other sites even when objects are not concurrently updated.

We are currently building a system that will allow us to quantify the performance improvements made possible by the lifetime based algorithm compared to other schemes that can be used to implement CC. We also want to develop techniques that can be used by applications to specify the set of objects for which CC should be provided (e.g., sessions in Bayou). We plan to explore different techniques for validating objects and will consider server replication. Another important problem is the investigation of how the size of the vector timestamps can be limited which is necessary for making the algorithms applicable in large systems.

References

[1] Mustaque Ahamad, James E. Burns, Phillip W. Hutto, and Gil Neiger. Causal memory. In *Proceedings of the 5th International Workshop on Distributed Algorithms*, October 1991.

[2] R. Alonso, D. Barbara and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM Transactions on Database Systems*, September 1990.

[3] Kenneth Birman, Andre Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, (9, 3):272–314, 1991.

[4] David Cheriton and Dale Skeen. Understanding the limitations of causally and totally ordered communication. In *Proceedings of ACM Symposium on Operating Systems Principles*, December 1993.

[5] S. Davidson, H. Garcia-Molina and D. Skeen. Consistency in a partitioned network: a survey. *ACM Computing Surveys*, September 1985.

[6] Colin Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the Eleventh Australian Computer Science Conference*, University of Queensland, Australia, 1988.

[7] Ranjit John and Mustaque Ahamad. Evaluating causal memory for data-race-free programs. Georgia Institute of Technology Technical Report, August 1994.

[8] James Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, October 1991.

[9] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, November 1992.

[10] Leslie Lamport. Time, clocks and the ordering of events. *Communications of the ACM*, 21(7):558–565, July 1978.

[11] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual consistency in distributed systems. *IEEE Transactions on Software Engineering*, May 1983.

[12] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of IEEE Symposium on Parallel and Distributed Information Systems*, 1994.

[13] Fresco Sample Implementation Refernce Manual, X Consortium Working Group Draft, Version 0.7, April 1994.