Scalable Consistency Protocols for Distributed Services *

Mustaque Ahamad
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

Rammohan Kordale
Scalable Information Systems
Silicon Graphics Inc.
Mountain View, CA 94087

August 1997

Abstract

A common way to address scalability requirements of distributed services is to employ server replication and client caching of objects that encapsulate the service state. The performance of such a system could depend very much on the protocol implemented by the system to maintain consistency among object copies. We explore scalable consistency protocols that never require synchronization and communication between all nodes that have copies of related objects. We achieve this by developing a novel approach called *local consistency* (LC). LC based protocols can provide increased flexibility and efficiency by allowing nodes control over how and when they become aware of updates to cached objects. We develop two protocols for implementing strong consistency using this approach and demonstrate that they scale better than a traditional invalidation based consistency protocol along the system load and geographic distribution dimensions of scale.

Keywords: Scalable Services, Distributed Objects, Replication, Caching, Consistency Protocols.

1 Introduction

As computing becomes more pervasive, local area networks are becoming larger and better interconnected. With this growth of internetworked computing, highly interactive applications will span many local networks. Examples of such applications come from the domain of computer supported cooperative work as well as virtual reality based distributed games, shopping and others. These applications need to be supported by distributed services that provide low latency access to dynamically changing shared state. The problem of scalable and consistent access is also important for dynamic content in the Web.

We are concerned with system support for building scalable distributed services that provide access to dynamically changing information. We explore scalability of such services along the two dimensions of *system load* and *geographic scale*. System load is the ability of the system to

*This work was supported in part by NSF grants CDA-9501637 and CCR-9619371.

withstand growth in the total number of potential clients. Geographic scale is the ability of the system to tolerate and reduce the effects of large latencies in wide-area distributed systems.

A common way to address scalability requirements of distributed services is to employ service replication and caching of service state at client nodes. We assume that services are encapsulated in distributed objects. Thus, scalable implementations of a service lead to the object state being distributed across multiple nodes. Server replication and client caching introduce the problem of consistency among multiple copies of an object. The focus of this paper is on the development and evaluation of scalable consistency protocols. The consistency problem is important because due to the interactive nature of applications, it cannot be assumed that access to shared service state is read-only.

Several approaches have been explored for developing scalable consistency protocols. In the context of distributed shared memory systems, synchronization information (e.g., lock operations) is exploited to reduce the cost of consistency maintenance [12, 21]. Because of the system scale and the varying degrees of coupling between application components, we do not assume that all accesses to service state are controlled via synchronization operations. For example, in a shared virtual environment, a node may want to read and render scenes from a recently cached copy without incurring the overhead of a lock operation. Another approach for improving scalability of consistency protocols is to provide only weak consistency. If shared state can be updated at multiple nodes, this could lead to conflicting updates which need to be resolved. We explore strong consistency protocols which ensure that all accesses to objects that implement related services can be serialized. Thus, all nodes agree on the order of updates to an object and conflicting updates are not possible.

Strong consistency can be formulated in a number of different ways. Sequential consistency [25] and serializability [10], used in shared memory systems and in databases, respectively, are two consistency models that require that execution of operations on shared objects should be serializable. However, these consistency conditions do not consider the time at which operations are executed. Thus, they do not require that an update to object x at one node must become visible at nodes that have copies of x in a certain amount of time. In interactive applications, timely dissemination of an update is required. Such timeliness is provided by strong consistency models such as atomic memory [30] and linearizability [18]. These consistency models require that serializations of operations respect the time at which the operations are executed. Clearly, protocols that implement such strong consistency are more expensive and do not permit any "lag" that may be acceptable in disseminating the effects of update operations. We explore a notion of strong consistency that guarantees serialization of all object accesses as in sequential consistency. However, it allows client nodes control over how quickly they want to become aware of updates to cached objects. We believe such flexible control over update dissemination is necessary for developing scalable protocols for strong consistency.

We develop a novel approach for building scalable consistency protocols. In particular, we develop a new mechanism called local consistency (referred to as LC). The basic idea of the LC mechanism can be summarized as follows. In traditional strong consistency protocols, nodes that update a shared object x take the responsibility of notifying nodes that have copies of x about the updates. When such nodes are widely distributed, update latencies can be very large. In LC based protocols, however, nodes that update shared state are not responsible for informing other nodes about the updates; instead, a node ensures that the object copies it accesses are consistent. In particular, only when a node needs to become aware of an external update (either by fetching a copy on demand or by accepting a copy that is sent to it), the node executes local consistency actions to invalidate local object copies that may be overwritten with respect to the newly arrived object. The advantages of this approach are numerous:

1. Access latencies can be lower in LC based protocols because they allow both read and write accesses to complete without requiring communication with all nodes that have copies of the accessed object.

2. Since object copies are lazily invalidated when a node executes local consistency actions and not when objects are written, the number of client requests to servers would be reduced thus decreasing server load. Furthermore, consistency related information can be batched and piggy-backed with object copies which decreases overall network load.

3. Nodes can fetch new copies of objects either on demand (as a result of an object fault) or new copies can be sent to nodes which can introduce them into their caches in a consistent manner. Thus, both push and pull style update dissemination is possible in LC based protocols.

4. LC based protocols guarantee serialization of all object accesses and yet allow each client to control how quickly updates should become visible to it. Thus, the currency of cached copies can be controlled based on resource availability (e.g., communication bandwidth) and application needs.

To evaluate the LC based consistency protocols, we built an object caching framework on top of a system that follows the CORBA distributed object model. The preliminary performance results support our hypotheses. We compared an LC based strong consistency protocol, referred to as $\mathrm{SC}_{\mathrm{lc}}$, with an invalidation based protocol called $\mathrm{SC}_{\mathrm{inv}}$. $\mathrm{SC}_{\mathrm{lc}}$ carries out consistency actions only when new object copies are pulled on demand because of object or access faults. On the other hand, $\mathrm{SC}_{\mathrm{inv}}$ synchronously invalidates all cached copies of an object when the object is written[1]. We show that with this version of $\mathrm{SC}_{\mathrm{lc}}$, which exploits the LC approach by allowing some read operations to complete with older object copies, better scalability is achieved along both dimensions of scale. By driving the system with a synthetic workload that captures read and write sharing patterns in large scale systems, we demonstrate that: (1) the average response time for object accesses in $\mathrm{SC}_{\mathrm{inv}}$ is 39% slower than in $\mathrm{SC}_{\mathrm{lc}}$ in the presence of widely distributed users, (2) $\mathrm{SC}_{\mathrm{inv}}$ imposes 15% more load on the server per client node than $\mathrm{SC}_{\mathrm{lc}}$, and (3) $\mathrm{SC}_{\mathrm{inv}}$ entailed up to 51% more messages per client node than $\mathrm{SC}_{\mathrm{lc}}$.

The rest of the paper is organized as follows. Section 2 motivates the LC approach. In Sections 3–6, we describe the two LC based implementations of strong consistency. The prototype and its evaluation are the topics of Sections 7–9. We discuss related work in Section 10 and conclude the paper in Section 11.

[1] Since we do not assume that locks are always acquired before accessing an object, invalidation messages cannot be sent to only those nodes that will request the lock.

2 LC Approach

In the system model assumed here, shared objects are stored at server nodes. Client nodes can cache object copies and access them locally. Objects can be related and consistency requirements can be defined across related objects. For example, if x and y are two related document objects (e.g., x contains results of an experiment and y contains a discussion of these results) and a user first updates x and then y, it is desirable that another user that accesses the updated copy of y will also see the new copy of x. Since nodes have copies of such objects, they are interested in updates to objects accessed by them. We define a system *global view* and *node views* to capture the values of objects that can be accessed by nodes.

2.1 Global and Node Views

Consider an imaginary external observer that sees each update to an object instantaneously. A global view is defined by the values of objects that are seen by such an external observer. As objects are updated, the observer will see a progression of *global views*. Since a node caches a subset of the objects, a *node view* is defined by the values cached at the node. The ideal case in which each node learns of an update instantaneously can be summarized as:

$Node's\ View \subset Current\ Global\ View$

To achieve this ideal case, a node that wants to update an object must synchronize with all nodes that have a copy of the object. Wide area latencies can make such synchronization costly when nodes are widely distributed. Furthermore, nodes having a copy of the updated object may not be interested in its new value (e.g., the object is not in their working set any more) .

In the LC approach, when the global view changes because of an update to an object, the updating node is not made responsible to advance others' views to the current global view. Thus, node views can lag the current global view. However, when a node view advances, the node must ensure that its new view does belong to some global view that is more recent. The following intuition shows why this is necessary. Assume the system in question is a centralized one in which all accesses are completed by sending requests to the centralized server. When a client's update request is serviced by the server, the global view changes. A node's view is defined by the values returned by its read requests. Suppose the current global view is G_i and node P_i wants to update the value of a shared object x from v to v'. Assume that the global view will change to G_{i+1} after P_i's update is completed. Thus, the value of x in global views G_i and G_{i+1} is v and v' respectively. It is interesting to observe the result of another node's, say P_j's, read request for x. The result will depend on whether P_j's request reaches and is processed by the server before or after P_i's request. In other words, even if a global clock existed and P_i's request was made "before" P_j's request, P_j need not see the newer value of x because of factors such as network delays. However, once P_j sees v' for x, subsequent reads of x by P_j cannot return older values (such as v). The LC approach is based on the above observation and can be defined in terms of node and global views. A node's view is *locally consistent* if the following hold:

1. $Node's\ View_i \subset Global\ View_j$, and

2. If $Node's\ View_i \subset Global\ View_j$ and $Node's\ View_{i+1} \subset Global\ View_k$, then

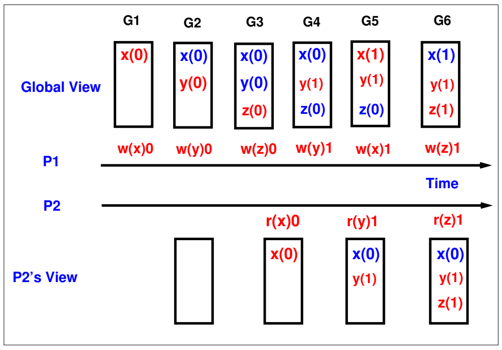

Figure 1: Global and Local Views

$Global\ View_k$ is a more recent global view than $Global\ View_j$.

In other words, a node's view does not necessarily have to be a subset of the current global view. However, it should belong to some global view. Also, the currency of a node's view should monotonically advance. The decoupling of a node's view from the instantaneous global view is the key for providing scalable as well as consistent access to shared objects.

Figure 1 shows a simple example to illustrate consistent node views. Node P_1 writes into shared objects x, y, and z, resulting in the progression of the shown global views. Assume P_1 has completed the first four writes when P_2 starts with an empty cache and reads x, y, and z. P_2 first caches in $x(0)$. The notation $x(0)$ means that the value of the copy of object x is 0. Clearly, P_2's view is locally consistent; its view belongs to a global view (one of G_1 to G_4). Suppose P_2 then reads $y(1)$. Adding this object copy to its cache still keeps all the cached copies locally consistent; its view belongs to G_4 and it is at least as recent a global view to which P_2's previous view belonged. Now, suppose that P_1 writes the new values into x and z (5th and 6th writes). Although P_1 updated its copy of x, P_2 need not be informed of it since object copies in its view are still locally consistent. If P_2 requests a copy of z and reads $z(1)$, adding $z(1)$ to P_2's cache and taking no further action would leave P_2's cache in the state shown in Figure 1. This view of P_2 is not locally consistent since it does not belong to any global view. To maintain consistency, P_2's copy of x can be invalidated when the copy of z is added to its cache. The resulting view, which contains $y(1)$ and $z(1)$, belongs to global view G_6 which is more recent than P_2's last view which was G_4. As this example suggests, a cache can be kept consistent by locally invalidating overwritten object copies when the node's view is advanced.

Consistency protocols that synchronously invalidate cached copies on writes ensure that local views of nodes are synchronized with the current global view. LC based protocols, on the other hand, allow local views to lag the current global view. The degree of lag can be controlled depending on resource availability and application needs. For example, when communication bandwidth is available, a client can periodically request updates or a server can push new object values to caching nodes. The client nodes will receive such values and consistently introduce them into their caches by performing local consistency actions. Thus, LC based protocols provide enhanced flexibility as they allow clients to access consistent object copies with various levels of currency of the cached information.

3 Strong Consistency

We use the LC approach to develop two scalable protocols for implementing strong consistency (SC). Our protocols ensure that accesses completed by all nodes can be serialized in an order that is consistent with the order of accesses at each node. The protocols focus on ensuring this safety requirement (e.g., they avoid conflicts which make such serializations impossible) and they can be augmented with additional mechanisms to meet timeliness or liveness requirements. We focus on the mechanisms that are used by the protocols to ensure that all operations can be serialized. Since we will compare these protocols with the invalidation protocol $\mathrm{SC_{inv}}$, we first briefly describe this protocol. The features of the LC based protocols are also outlined before the actual protocols are presented.

3.1 Invalidation Protocol ($\mathrm{SC_{inv}}$)

In an invalidation protocol, clients cache object copies that they access. When a client P_i writes an object x, it needs to cache a copy of x in exclusive mode. If x's copy has not already been cached, P_i experiences a write-miss. On the other hand, if a copy has been cached in a non-exclusive mode, P_i experiences a write-fault. In either case, P_i communicates with the server which returns its copy of x if it is the only copy in the system. If other copies exist, the server invalidates all such copies, collects acknowledgments from the respective nodes and returns exclusive access to x to client P_i. We refer to the client that caches an object copy in exclusive mode as its *owner*. When P_i attempts to read an object copy and experiences a read-miss, it communicates with the server. If no client caches the object copy in exclusive mode, the server just returns its copy to the client. Otherwise, the server downgrades the owner's copy to read-only mode and provides P_i the latest copy from the owner. The scalability of this protocol is limited because it requires communication with all nodes having object copies when a write request is received by the server. The two LC based protocols presented next do not require such communication.

3.2 LC based Protocols

Similar to $\mathrm{SC_{inv}}$, LC based protocols also assume a single writer for an object at a time but there are important differences between the invalidation protocol and those based on LC.

1. In LC based protocols, when a server transfers write access to a new node, existing readers are not sent invalidation messages. In fact, readers of an object can coexist with a writer by reading values from old but consistent node views. Thus, not only invalidation messages are avoided, a node does not need to receive and process messages each time an object cached by it is updated at another node.

2. LC based protocols order all accesses to related objects by advancing node views in a consistent manner. At the time a new object copy is added to a node cache, the node performs local consistency actions to ensure that currently stored copies of shared objects are valid in the new node view. Such consistency actions require no communication because they only invalidate local copies that are potentially overwritten in the new view.

3. A client node must communicate with an object's server when it either does not have the object in its cache or it wants to update the object but has only read permission. Such communication with the server can advance the client's view. In addition, client views can be advanced with periodic communication that can be initiated by the server or the client. This periodic communication ensures that the lag in client views and the global view is bounded. Thus, while guaranteeing that object accesses can be serialized, LC based protocols offer the flexibility to control the currency of cached object copies based on resource availability and application needs.

The novel aspect of our consistency protocols is how the local consistency actions are implemented. We will explore two implementations of such actions. The first one assumes that all objects across which consistency is desired are stored at a single server node. Although this protocol can precisely determine what object copies should be removed from a node cache when the node view advances, it cannot handle distributed servers. The second implementation addresses this problem and allows related objects to be stored at multiple server nodes.

4 Invalidation-Set Protocol

In this Section, we present a single-server protocol that follows the LC approach to implement strong consistency and we call it the *invalidation-set protocol.* This protocol is based on efficient sequential consistency protocols that have been developed for shared memory systems. For example, our protocol is similar to the one presented in [31] but we do not require that a client communicate with the server on each write operation. Other protocols that share a similar approach include the dynamic self-invalidation protocol [27] and the implementation of sequentially consistent memory described in [1].

The invalidation-set protocol maintains consistent node views by receiving information about what object copies stored at the node have been overwritten since it last communicated with the server. In the case of a single server, such information can be maintained efficiently by the server node. The protocol is shown in Figure 2. All write misses/faults require communication with the server. The server does not send messages to invalidate other copies in the system when such requests are received. Instead, information about the objects that need to be invalidated is maintained on a per-node basis, that we refer to as *invalSet* (for invalidation set). As shown in the *chngOwner* method in Figure 2, if a client node's (say P_i's) request for ownership of an object x results in the server determining that the copy on node P_j should be invalidated, x is added to *invalSet[j]* and the client's request returns. When P_i's request returns, *invalSet[i]* is sent along with the result of the request. Read misses are handled as in $\mathrm{SC_{inv}}$ except that the invalidation set corresponding to the requesting client is also returned as in the case of write misses/faults. When a client receives an invalidation-set from the server, it invalidates all cached copies of objects listed in the invalidation-set. Thus, object copies at a client are invalidated only when the client communicates with the server and not when the object is written.

A simple execution allowed by this protocol is shown in Figure 3. Client nodes P_i and P_j initialize x and y respectively. In steps 3 and 4, P_i and P_j cache y and x for reading,

```
ACTIONS AT CLIENT Pi:
  readMiss(x)
    < x, invalSet> = x.server.access(x, Pi)
    invalidate local copies of objects
      that are members of invalSet;
    x.access = read

  writeMiss(x)
    //same definition for writeFault(x) except
    // that a copy of x is not received
    < x, invalSet> = x.server.chngOwner(x, Pi);
    invalidate local copies of objects
      that are members of invalSet;
    x.access = read-write

  downgrade(x)
    x.access = read
    return(x);
```

```
ACTIONS AT SERVER:
  access(x, Pi)
    if (x.owner ≠ self)
      //communicate with current owner
      x = x.owner.downgrade(x);
      x.owner = self;
    returnSet = invalSet[i];
    invalSet[i] = ∅;
    return(< x, returnSet>);

  chngOwner(x, Pi)
    if (x.owner ≠ self)
      //communicate with current owner
      x = x.owner.downgrade(x);
    x.owner = Pi;
    for each client Pj caching x
      add x to invalSet[j];
    returnSet = invalSet[i];
    invalSet[i] = ∅;
    return(< x, returnSet>);
```

Figure 2: Invalidation-set protocol

Step	Node P_i	Node P_j	InvalSet[i]	InvalSet[j]	Remarks
1	w(x) 0		Empty	Empty	- P_i becomes x's owner
2		w(y) 0			- P_j becomes y's owner
3	r(y) 0				- P_j's copy downgraded to read only.
4		r(x) 0			- P_i's copy downgraded to read-only.
5	w(x) 1			Add x	- x added to invalSet[j]
6		w(y) 1	Add y	Remove x	- y added to invalSet[i].
					- P_j's copy of x is **invalidated**.
7	r(y) 0				- P_i reads old value of y.
8		r(x) 1			- P_j reads latest value of x

Figure 3: Invalidation-set protocol - a simple example.

thus downgrading the other node's copy to read-only mode. In step 5, P_i writes x. It has to communicate with the server which grants ownership of x to P_i while adding x to $invalSet[j]$. Note that P_j can continue to read the old copy of x until it needs to contact the server. In step 6, P_j contacts the server to write y. The server adds y to $invalSet[i]$ and returns $invalSet[j]$ to P_j. P_j locally invalidates x because it is included in $invalSet[j]$. In the last two steps, P_i reads the old copy of y while P_j reads 1 since its old copy of x was invalidated.

In the protocol described in Figure 2, we do not show an action for advancing the client views. Such an action can be executed either by the client or the server and results in the server sending the invalidation-set to the client in a message. The client invalidates local copies of the objects which are included in this message. Thus, a future access to an invalidated object will result in communication with the server which will advance the client view to a more recent one.

4.1 Correctness

The correctness proof of the invalidation-set protocol requires demonstration of a serialization of all object accesses when the protocol is used to maintain the consistency of node caches. Informally, a serialization α can be produced as follows. If all accesses are completed by communicating with the server, α is simply the order in which the central server executes the requested operations. Object caching permits clients to complete accesses without communicating with the server. In this case, the server only serializes the operations that require communication with the server. Assume that execution of operation o requires communication with the server. The set of local accesses that immediately follow o can be serialized after o in α. These accesses will be consistent because (1) all updates before these local operations become known to the client via the invalidation-set received when it last communicated with the server, and (2) writes to object copies cached with exclusive access become visible immediately. Such a serialization can be constructed even when one client reads values from an older view while the object is updated at another node. A more detailed proof can be found in [23].

5 Object Lifetime based Protocol

If consistency is required across related objects that are stored at several servers, an invalidation-set for a node can not be maintained at a single server without incurring additional communication overhead. We now develop a protocol that allows distributed servers. We assume that related objects are partitioned among servers and the server for an object remains fixed (object migration and server replication are not addressed in this paper). In an LC based protocol, when a node's view advances because of a newly cached object copy, it needs to know what existing object copies have been overwritten in the new node view. We take an approach in which meta-data associated with object copies is used to determine what existing object copies are still valid when a node moves to a newer view.

We denote the copy of object x that stores the value v by $x(v)$. The meta-data associated with $x(v)$ is called its *lifetime*. Intuitively, the lifetime of an object copy is the time interval during which the copy can be accessed without violating consistency. If a synchronized global clock C is available at nodes that access object x, the lifetime of $x(v)$ is defined by the time interval $[x(v).wt, x(v').wt)$. $x(v).wt$ is the time read from C when v is written into x and we call $x(v).wt$ the *write-time* of $x(v)$. At time $x(v').wt$ (this time is also read from C), v is overwritten by the value v' to generate a new copy $x(v')$. Thus, $x(v').wt$ is the write-time of $x(v')$. A lifetime can easily be associated with copies of an object when the object is updated periodically at known times[2]. However, in general, it is not possible to associate a lifetime with an object copy when it is written because the time of the next write to the object is not known. We address this problem by storing only the known lifetime of an object copy. This is done by associating a *valid-time*, $x(v).vt$, with copy $x(v)$. Thus, the known lifetime of $x(v)$ is the interval $[x(v).wt, x(v).vt]$. Initially, $x(v).vt$ starts out as the same as $x(v).wt$ but it is advanced as the system learns that $x(v)$ has not been overwritten.

If each object copy cached at a node has a lifetime associated with it, a straightforward

[2]The time-to-live (TTL) field included in web pages that store dynamic content is one example of how lifetime is used to establish validity of an object.

test can be developed to determine if a set of cached copies belong to a consistent global view. Two copies $x(v)$ and $y(v')$ belong to a consistent view if they have overlapping lifetimes. In other words, if $y(v')$ is written more recently than $x(v)$, $x(v).vt$ must be greater than $y(v').wt$ for the lifetimes of the two copies to overlap. In this case, v is the correct value of x when $y(v')$ is generated and hence the two copies, together, belong to a consistent global view. If the lifetimes of the two copies do not overlap, $x(v)$ could be locally invalidated when $y(v')$ is added to the node cache. Such a lifetime based protocol follows the LC approach because a node can carry out consistency actions locally without communicating with other nodes. Consistency is preserved because when cached copies have overlapping lifetimes, they do belong to common global views, including the global view in which the incoming object copy is generated. A set of copies belong to a consistent view if each pair of copies in the set has overlapping lifetimes.

We defined object lifetimes using a synchronized global clock. In large scale distributed systems, it is not reasonable to assume the existence of such a clock. We develop a consistency protocol that does not require synchronized clocks. Instead, logical clocks are used to record object lifetimes. In particular, we use vector clocks [32, 11]. Vector times have been used to order writes and to detect conflicting updates in replicated data systems such as Bayou [36] and Coda [22].

In our system, a vector clock is a vector of integers with a component for each *update node*[3], where an update node is defined as a node on which an object can be updated. We denote the clock at node P_i by VT_i. Vector times can be *incremented*, *updated* and *compared*. Node P_i *increments* VT_i by incrementing $VT_i[i]$. *update(VT, VT')* returns the component-wise maximum of the two vector times. A vector time, t_1, is less than another time, t_2, if each component of t_1 is less than 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 vector times are not comparable when neither is less than or equal to the other and such times are called *concurrent*.

The key to a correct implementation of strong consistency using object lifetimes read from logical clocks are a set of rules that specify how the clocks should be maintained. In particular, nodes must advance their clocks in such a way that node and global views can be ordered by logical times at which they are created. To achieve this, each node increments its component in the logical clock when its view advances. To order views resulting from the execution of operations at different nodes, we will develop a set of clock rules that use times associated with object copies to update node clocks. This will ensure that clock times in various views will correctly reflect the order between the views.

Object lifetimes may appear similar to time-to-live (TTL) fields (e.g., the Alex protocol [6]) or leases [15] which have been explored to reduce consistency related messages. However, there are many differences. We use logical time to define lifetimes and use them to ensure consistency across related but different objects. Also, a lease refers to a time interval in the future with a guarantee that a client will be notified of an update to a cached copy during the lease period. On the other hand, a lifetime refers to a time interval in the past in which the object is known

[3]Although a large number of nodes may access an object, we assume that most nodes access the object in read-only mode. If this is not true, the scalability of the vector clock could become an issue. We do not strictly need a component for each update node and more compact logical clocks can be used in place of vector clocks [38].

to be valid.

5.1 Clock Rules

Since we want to order views by logical clock times, clocks need to be advanced to reflect view changes. As in traditional vector clocks, views that result from the execution of operations at a given node can be ordered by incrementing the node clock after executing each operation. To order operations across nodes in message passing systems, vector times included with messages are used to advance node clocks. For example, when P_i sends a message to P_j, it includes a time read from its clock VT_i in the message. When P_j receives this message, the time included in the message is used to update VT_j. In object sharing systems, operations across nodes become ordered because of two reasons. First, if operation o_w writes v to x and the value v is read by operation o_r at another node, o_r must be ordered after o_w. Also, if v and v' are the values of x written by two consecutive write operations, a read operation that returns v must be ordered before $w(x)v'$. We can capture these orderings with a vector clock as follows. We will use the write-time, $x(v).wt$, to advance VT_i when P_i caches $x(v)$. Since o_r is executed after $x(v)$ is cached, this will ensure that the times at which o_w and o_r are executed will be properly ordered. To order read operations $r(x)v$ before the update that overwrites the value v, we associate a *read-time* with $x(v)$. The read-time of $x(v)$, denoted by $x(v).rt$, must reflect the times at which $x(v)$ has been read at various nodes (e.g., the times at which $x(v)$ has been read must be less than or equal to $x(v).rt$). If a read-time can be associated with an object copy (see Section 5.2), the needed ordering between a read that returns v and the write operation that overwrites v can be easily ensured. For example, if o'_w executed at P_k overwrites v, VT_k is updated with $x(v)$'s read-time before P_k executes o'_w. The clock rules, which capture how node clocks are advanced, are described below for each action generated by an object access.

1. Rule R1: R1 is used when a node P_i completes an access locally with a cached copy of an object. If the object copy is only read in completing the access, VT_i is not advanced because the node remains in the same view. If the access updates the object copy when the node has exclusive access to the object, VT_i is incremented because the node view advances as a result of the update.

2. Rule R2: R2 is used when an access at P_i results in a read-miss for an object x. P_i must communicate with x's server to fetch the object copy before the access can be completed. If the server knows that there is no writer in the system (e.g., all nodes caching x have it in read-only mode), the server sends its copy of x to P_i. Otherwise, the server communicates with the writer and returns the writer's copy to the client after downgrading its access mode. On receiving the copy $x(v)$ with the associated times, P_i advances VT_i as follows: $VT_i = update(x(v).wt, VT_i)$. This ensures that the view in which the read access is completed by P_i is ordered after the view in which $x(v)$ was produced.

3. Rule R3: If P_i has an object x cached in read-only mode and wants to update it, a write-fault is experienced. Similarly, if P_i wants to write x when it is not in the cache, a write-miss is experienced. In both cases, P_i must communicate with x's server. If another node has x cached in write mode, the server requests it to downgrade its access to read-only.

The current copy of the object, $x(v)$, is sent to P_i along with its write-time, read-time and valid-time. VT_i is first advanced as follows: $VT_i = update(x(v).wt, x(v).rt, VT_i)$. This action ensures that the new write being executed by P_i is ordered after the write that produced the overwritten copy $x(v)$. Furthermore, operations that read $x(v)$ are also ordered before the update done by P_i. Similar to writes to cached objects, VT_i is also incremented after the write is completed.

Rules R1–R3 ensure that times read from the vector clocks in node views are ordered consistently with the order among the views. For example, Rule 3 ensures that write times of a given object are totally ordered. Rules 2 and 3 together ensure that the time of an access that reads value v of object x is greater than the time at which write operation $w(x)v$ is executed. Similarly, if read-times for objects are maintained correctly, Rule 3 ensures that for two consecutive writes $w(x)v$ and $w(x)v'$, the times at which read operations $r(x)v$ complete are ordered before the time of $w(x)v'$. We next discuss how the various times are assigned to object copies.

5.2 Write, Read and Valid Times

5.2.1 Write-Times

If node P_i executes the operation $o_w = w(x)v$ to generate the object copy $x(v)$, $x(v).wt$, the write-time associated with $x(v)$, is simply the value of VT_i after o_w is completed. Since P_i must cache object x in exclusive mode by experiencing a write-miss or write-fault before P_i executes o_w, VT_i was updated with the write-time of x's value that is overwritten by o_w. This ensures that vector times corresponding to views in which new values of x are generated are totally ordered.

5.2.2 Read-Times

Read-times are used to update clocks so that a view in which operation $o_r = r(x)v$ executes is ordered before the view in which an update operation o_w overwrites v. Since the value v can be read at multiple nodes before it is overwritten, $x(v).rt$ must reflect all such read operations. One possible way to implement this is to have the server contact all clients that have read-only copies. The read-time returned to a node that experiences a write-miss or write-fault could be the component-wise maximum of the clock values obtained from all such clients. The invalidation based protocol $\mathrm{SC}_{\mathrm{inv}}$ does induce communication with all clients caching read-only copies when write-misses and write-faults are encountered. However, we avoid this by maintaining $x(v).rt$ at x's server node by using the following rules.

- When node P_i requests a read-copy of x from its server, it also informs the server of the value of its local clock, VT_i. The server updates the read-time maintained by it for x by setting $x(v).rt = update(x(v).rt, VT)$, where VT is the value of VT_i when P_i sends the request to the server.

- If x is currently cached by some node P_j in exclusive mode when a read-miss is experienced by P_i, to service the read-miss, x's server communicates with P_j. P_j downgrades its copy of x to read-only mode and returns the current value of VT_j along with $x(v)$. The server updates $x(v).rt$ with received clock value because it is as if P_j just acquired a read-only copy of x.

The server returns $x(v).rt$ as the read-time with $x(v)$ to P_i which experienced the write-miss or write-fault. Clearly, a write $o'_w = x(v')$ may not be ordered after all reads of the copy $x(v)$ that it overwrites. This is because $x(v)$ can be read at other nodes after o'_w is executed since readers are allowed to coexist with a writer. However, such reads are completed in views with times that are less than or concurrent with the time of the view that results from the execution of o'_w. This is because any operation, performed on node P_j, which follows o'_w (in other words, the operation is aware of o'_w), will result in the invalidation of $x(v)$ at P_j. Thus, the view resulting from the execution of o_w is either after or concurrent with the views in which the older copy $x(v)$ is read. Therefore, it is possible to order o'_w after all reads of $x(v)$ in a serialization.

5.2.3 Valid-Times

The local consistency check, performed when an object copy $x(v)$ is cached, invalidates all cached object copies whose valid-times are less than the write-time of x. This ensures that older copies that have non-overlapping lifetimes with the incoming copy are not accessed without communicating with the server because these copies could be overwritten in the view in which $x(v)$ is generated. Clearly, even when an object copy is not overwritten, it can be invalidated unnecessarily if a node has not advanced the valid-time of the copy. Advancing valid-times as much as possible reduces unnecessary invalidations. In our implementation, valid-times of object copies are updated on three occasions:

- The valid-times of locally cached object copies can be advanced to the node's clock value since the object copies are known to be valid in the node's present view. Also, arbitrarily high valid-times can be set for object copies that will not be written again (e.g., write-once copies).
- When a server requests a node for an object copy, the node updates the copy's valid-time with its clock value before returning the copy (for the same reason as above).
- In the protocol we develop, the server always returns a valid copy. Consequently, the server updates the valid-time of an object copy with its clock before returning it to a requesting client node.

Furthermore, to aggressively advance valid-times, a server node also maintains a vector clock. Each time a client contacts a server (cache misses, write faults and object copy flush), it provides the server with its local clock value. The server updates its clock using the vector time received from the client to advance its view to reflect the operations that have been executed at the client. If the client requested a copy of object x in exclusive mode, it will first increment

P1				P2			
wt	vt	VT1	operation	wt	vt	VT2	operation
[10]	[10]	[10]	w(x) 0	[01]	[01]	[01]	w(y) 0
[01]	[01]	[11]	r(y) 0				
				[12]	[12]	[12]	w(y) 1
				[10]	[12]	[12]	r(x) 0
[22]	[22]	[22]	w(x)1				

Figure 4: A sample execution scenario

its clock and then send it to the server (with the request for object ownership) to ensure that the server's clock reflects the update operation executed at the client. The server makes use of its clock to advance valid times of object copies before it sends them to requesting nodes.

5.3 The Protocol

The consistency protocol based on the object lifetime technique is shown in Figure 5. We simplify the notation by using x to refer to the copy $x(v)$ when the value stored in the object copy is not relevant to the discussion. Thus, we will use $x.wt$, $x.rt$, and $x.vt$ to denote the write-time, read-time and valid-time stored with a copy of object x.

Since a read or write access is completed locally with a cached copy, we omit the description of these operations and focus on how a client requests and adds an object copy to its cache in the desired mode. Thus, the handling of read-miss, write-miss and write-fault events are shown at the client side. When a client experiences an access miss or fault for object x, it requests a copy of x from its server. On receiving a copy of x with the various times, the client compares the incoming copy's lifetime with the lifetime of every related object copy that is currently in its cache. For example, if x and y are related, first $y.vt$ is updated with the node's clock. After this update, if $y.vt \not\geq x.wt$, y is locally invalidated. The local invalidations ensure that a node's cache contains consistent copies of objects. The node clocks are advanced according to rules R1–R3 and various times are associated with the object copy.

Although a read-miss results in communication that is similar to what is done in the invalidation protocol, write-misses and write-faults are handled very differently in the lifetime based protocol. On receiving a write miss/fault request from a client for object x, if an owner exists, the server requests the owner for a copy of x. The owner's status is downgraded to that of a read-only client; its copy is not invalidated. Other read-only copies of x are not invalidated and no messages are sent to these clients. At this point, the server has the most recent copy known in the system which is returned to the requesting client. Thus, multiple readers can coexist with a single writer.

Consider the execution scenario shown in Figure 4 that provides more intuition into the workings of the protocol. Objects x and y are initialized to zero in the first step on nodes P_1, and P_2 respectively. The three quantities to the left of the individual operations in the Figure are respectively the write-time(wt), valid-time (vt) of the object copy, and the local clock value when the operation is executed. P_1 then caches a read-only copy of y thus downgrading P_2's copy of y. P_2 then wishes to update its copy of y. Therefore, it communicates with the server to obtain ownership of y. Following the rules, $y.wt$ = *increment(update(*$y.wt$, VT_2, $y.rt$ *))* =

```
ACTIONS AT CLIENT NODE Pi:
readMiss(x)
  x = x.server.access(x, read, VTi, Pi)
  x.access = read-only;
  cacheIn(x);  VTi = update(VTi, x.wt)

writeMiss(x)
    //Same definition for writeFault
  x =x.server.access(x,
        write, increment(VTi), Pi)
  x.access = read-write;
  x.wt = update(x.wt, x.rt, VTi))
  cacheIn(x);  VTi = x.wt

clntAccess(x)
  x.access = read-only
  x.vt = update(x.vt, VTi)
  return(x, VTi)

cacheIn(x)
  for every object g in the cache
  g.vt = update(g.vt, VTi)
  if ((g.vt ≯ x.wt) and g is not locally owned)
    invalidate g
```

```
ACTIONS AT x's SERVER:
access(x, mode, VT, Pi)
  VTserver = update(VTserver, VT )
  if (x.owner ≠ self)
    < x, VT' >= x.owner.clntAccess(x)
    VTserver = update(VTserver, VT')
    x.rt = update(x.rt, VT')
  x.vt = update(x.vt, VTserver)
  if (mode = read)
    x.rt = update(x.rt, VT)
    x.owner = self
    return(x)
  else
    x.owner = Pi
    return(x)
```

Figure 5: Lifetime based protocol

increment(update([01],[11],[10])) = [12] ($y.rt$ is VT_1 when y was read by P_1). No copies need to be invalidated locally since y is the only copy at node P_2. Also, VT_2 is advanced to *[12]*. Suppose now node P_2 caches a read-only copy of x and then P_1 wants to update x. It assigns the write-time using the same rules and hence $x.wt =$ [*22*]. In addition, the local copy of y is invalidated because $y.vt < x.wt$. Thus, while node P_2 continues to read the older value of x, a future read of y on node P_1 will result in a cache miss and read a value more recent than 0.

The protocol described in Figure 5 does not include actions that advance client views periodically. Similar to the invalidation-set protocol, such actions can be added to this protocol. A server can send object lifetimes recorded at it to the clients according to timeliness requirements. If the lifetime received from the server has a higher valid-time, the client advances its valid-time for the copy. If the received lifetime has a higher write-time than the write-time of the cached copy, the copy is locally invalidated. The latter can allow nodes to advance their views by requesting more recent copies of the invalidated objects.

5.4 Correctness

Similar to the invalidation-set protocol, the correctness of the lifetime based protocol also requires the demonstration of a serialization of all object accesses when this protocol is used to maintain consistency of object copies. The basic idea of producing such a serilaization is to order accesses according to the logical times at which they are executed. The clock rules, which update the clocks based on various times associated with object copies, ensure that the logical times at which operations are executed are ordered in a consistent manner. Another way in which correctness can be demonstrated is by using an argument that is inspired by work in scalable or weakly ordered memory systems [2, 12, 35]. These systems assume two types of operations: synchronization or labeled operations and ordinary operations. Synchronization operations order the ordinary operations so the resulting executions are ones that can also be obtained in a strongly consistent system. We do not assume that user programs always have "sufficient synchronization" or are properly labeled, since all accesses may not be executed after acquiring locks. For example, reads of slightly stale data are acceptable in many applications and such reads can be completed with cached copies without acquiring a lock. In the absence of explicit synchronization, orderings induced between operations by the clock rules in our approach result in implicit synchronization which allows us to demonstrate that all accesses can be serialized. A proof that demonstrates correctness can be found in [23].

6 A Hybrid Protocol

The object lifetime based protocol can sometimes be more conservative than required. For example, assume that in Figure 4, the second write of P_2 is to another related object z instead of $w(y)1$. In this case, P_1 will invalidate the copy of y when it executes $w(x)1$ even when the value of y at it is not overwritten. This is because the write to z will advance the clocks so that $y.vt < x.wt$ at P_1 when it executes $w(x)1$. Thus, the protocol can invalidate object copies unnecessarily. On the other hand, the invalidation-set protocol does not suffer from unnecessary invalidations but it only works correctly in the single server case. We next outline a hybrid protocol that combines the two protocols and call it $\mathrm{SC_{lc}}$.

When a client experiences a write-miss or a write-fault on object x, it contacts x's server as before. Let us denote this server as S_x. S_x maintains both the invalidation-set and write, read and valid times for each object copy. Both the invalidation set and the times are included in messages that are received by client nodes. On receiving this information, locally cached objects included in the invalidation-set are first invalidated. These objects are managed by S_x and since all invalidating writes to such object copies need to contact S_x, the invalidation set information for these objects will be maintained correctly. For objects that are not managed by S_x, consistency is maintained using the object lifetime based protocol with the times received with the copy of x. Thus, a combination of the two protocols correctly maintains the consistency of cached object copies. We implement $\mathrm{SC_{lc}}$ in the prototype system that is discussed next.

7 Prototype Implementation

We have implemented an object caching system to evaluate the protocols described in the previous sections. Our system supports IDL/C++ based distributed objects that can either be invoked via the remote method invocation (RMI) paradigm or their code and state can be cached locally. The system is built on top of the Fresco toolkit which supports RMI based distributed objects. To perform consistency actions for cached object copies, object methods specify if their execution only reads the object state or the state is also updated. In the current version of the system, a cacher process runs at each client which is responsible for communication with the server. It shares memory with the client process where shared objects are allocated space. Although the caching architecture at client nodes could impact object invocation performance, our focus in this paper is on the comparative evaluation of the consistency protocols. Thus, we omit the details of the prototype, which can be found in [24], and focus on its evaluation.

8 Workload

In the absence of traces from interactive applications deployed in a wide-area environment, we considered synthetic workloads to evaluate the consistency protocols. Existing workloads that can be used for such an evaluation fall mainly in two domains – distributed file systems, and the World-wide-web. Our initial evaluation of the system is based on distributed file system workloads. These workloads were chosen because of the richer read/write sharing patterns that exist in them and also because we could use a workload that captured accesses of users cooperating in a geographically distributed software development environment[4]. The first workload is taken from Matt Blaze's thesis on "Caching in large scale distributed systems" [4]. We call this the Princeton workload and it documents file accesses of approximately 250 users collected over a week in the Princeton University Computer Science department. This workload was used to evaluate protocols along the system load aspect of scalability. The second workload [13, 14] is taken from a large industrial distributed file system supporting several thousand users organized into project oriented workgroups and connected together by a hierarchy of local area and wide area networks and spread across several metropolitan areas. This study was conducted by researchers at the University of Toronto. We call this workload as the Toronto workload and it was used to evaluate protocols along the geographic distribution aspect of scalability.

Apart from the above two workloads, we also use information from the Sprite [3] and the xFS [8, 9] traces. We chose to implement a synthetic workload generator because neither usable live loads nor the traces of the Toronto workload (which was uniquely useful for evaluating geographic scalability) were available. Our synthetic workload generator can be described by the following parameters.

Number of Object Invocations Per Client: Each client makes 50000 object invocations. In the Princeton trace, this takes about 16 hours.

Inter-access Times: We generated inter-access times randomly using an average inter-access time. The average inter-access time in the Princeton study is approximately one second but this would force each experiment to run for 16 hours. In order to run many experiments in a controlled manner, we use an inter-access time of one-sixteenth of a second. Reducing inter-access times arbitrarily can have an effect on the accuracy of the results. We will discuss this in the next section.

Creation: Every time an object access is generated, the probability that an object would be created was taken to be 0.5% [5].

[4]There exist workloads from Web servers that support access to dynamic content (available from the URL http://ita.ee.lbl.gov/html/traces.html). Such workloads represent a single writer and multiple reader access patterns. We wanted to explore consistency with distributed servers and when the objects can be updated at multiple nodes. As a result, we did not use the Web workloads.

Previous Operation	Read by Same user	Read by New user	Write by Same user	Write by New user
Read	51.7%	36.3%	10.1%	2.0%
Write	18.7%	10.4%	70.0%	0.9%

Table 1: File inertia (Princeton workload)

Probability	Chosen object accessed in the last n minutes
0.575	25
0.665	50

Table 2: Temporal locality (Princeton Workload)

Inertia: Given that a file was accessed by some client in read or write mode, inertia determines the probability with which the same or different client will next access the file and the probability of the mode of that access. The file inertia in the Princeton trace is given in Table 1.

Entropy: The likelihood that a file will be written decreases sharply after it has been opened for reading several times. We used the entropy suggested in the Princeton workload.

Temporal Locality: We used the numbers in Table 2 from the Princeton workload.

Read-write Sharing: Files that are widely shared at many nodes are good candidates for replication or caching only if they are not expected to change often. We use the quantities from the two workloads in corresponding experiments. We use a read-write ratio of 4:1.

8.1 Workload Generation

The workload is generated one event at a time by a generator that iterates over the number of clients. The average inter-access time determines the timestamp for the event being generated. The creation rate determines whether the next event should be object creation or object access. If an object needs to be created, one of the servers is chosen with equal probability. In case of an object access, the read-write ratio determines the access type. Temporal locality suggests which object to choose in the list of events previously generated by the client. Once an object is chosen, inertia is used to determine what the next operation of the object will be and who (same or different client) should access the object. Of course, entropy can force the operation to be a read. The data structure associated with an object thus contains the following information - object-id, access time, access type and client name of the next operation, number of successive reads (for entropy), and the list of readers and writers of the object to choose from. The number of readers and writers are decided when the object is created using the numbers for read sharing and read-write sharing; the exact identity of readers and writers is determined with equal probability among the available clients.

Our workload generator faithfully uses values for all but one of the workload parameters from the previous studies (in particular, the Princeton workload). As mentioned earlier, we use an inter-access time that is one-sixteenth of the value in the Princeton workload. Reducing inter-access times arbitrarily can have an effect on the results of the experiments. We attempted to keep the trace faithful to the intended workload using the following two pronged approach.

Firstly, though we were interested in reducing the average inter-access times, we chose it methodically to shorten the experiment durations. A client delays an access until its present time matches the event time. We keep track of the accesses that are delayed in this manner. This number gives us an indication of the amount of control over workload generation; more the number of delayed accesses, more the control. The inter-access time chosen was such that it was the smallest inter-access time using which the clients had to delay before more than 95% of the accesses. We also ensured that access times of the same object on different nodes are sufficiently different to take into account the time it takes the server to contact the writer and send consistency related messages. Finally, we performed experiments using the Princeton workload with average inter-access times of one-fourth of a second (4 times the average inter-access time used in the experiments) and found that the protocols were not sensitive to this change. Since we are interested in a comparative evaluation, we feel that all experiments will be meaningful with the reduced access times.

9 Performance Evaluation

The important performance metrics studied are the average response time for object accesses, the total number of cache misses and server requests, and the time spent at the server to service a cache miss. Server requests are the messages that a server node has to send to service client requests. We measure **server load** per client as a product of cache misses experienced by one client and the **time spent at the server** to service a cache miss. Cache misses and server requests account for the **total number of messages** in the system which represents the network load.

In the performance study reported in this paper, we did not include actions in SC_{lc} that periodically require clients to communicate with a server to advance their views. Thus, client views are advanced only when the client experiences an access-miss or a write-fault. On the other hand, in the invalidation protocol SC_{inv}, cached copies are invalidated synchronously when a write is done on an object. As a result, certain read accesses with SC_{lc} can return older values compared to SC_{inv}. However, SC_{lc} aggressively downgrades access to read-only to an object copy when another client wants to update the object. Since we used a 4:1 read/write mix, clients do experience write-faults and communicate with server nodes. Such communication results in the invalidation of read-only object copies. Similarly, cache replacements result in communication with a server when a replaced object is accessed in the future. However, the results reported here are for a base SC_{lc} protocol without periodic messages for advancing node views. The overhead of such messages will depend on the degree of coupling required between node views and will be explored in our future work.

9.1 Experimental Setup

A total of eleven 167 MHz UltraSPARC-1s were available for controlled experiments. All of them have a main memory of 64MB and are connected by a 155Mbs ATM switch. This cluster was used for scalability evaluation along the dimension of server load. For wide area

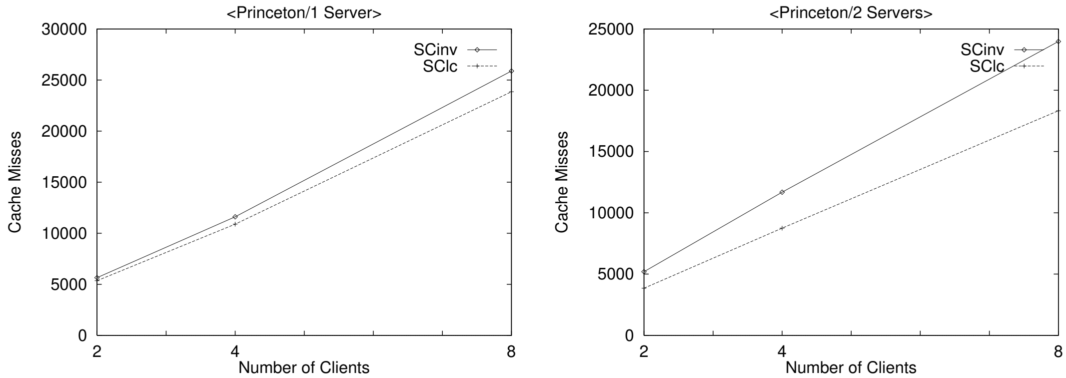

Figure 6: Cache misses

response time results, we measured communication costs between machines located at two different campuses in a city. These costs were used in estimating access response times for this case. Below, we discuss system parameters that can affect the results of our experiments.

- **Size of data transfer:** We used the average size of file transfer found in the Toronto study which is 16KB.
- **Client cache:** We define a full cache to be one increasing whose size will not reduce the number of cache misses. In previous studies [4, 8], a cache size that is half the size of a full cache is chosen. We also chose a cache that was half the size of a full cache. The full cache size in our case is smaller compared to the full cache of 512 files in the Princeton study [4], and 1024 files in [3, 8]. This is because the total number of objects in the workload is much smaller (about 1500 objects) due to the fact that all the objects created in our workload are active. The study by Bodnarchuk and Bunt [5] also supports the use of a small number of objects. We implemented the LRU cache replacement policy.
- **Server cache and disk delay:** We emulate an object server by using statistics from previous studies [8]. We assume 20% server cache hit ratio and 14 ms. average disk access time.

9.2 Results

We now present the results of our experiments that evaluate the protocols along the two dimensions of scale - system load and geographic distribution. The titles for the graphs are in the following format: $< W/X >$, where W is the workload used and X is the number of servers or clients.

System Scale: Figure 6 shows cache misses for the Princeton workload for the 1-server and 2-server cases respectively. The number of cache misses is lower for $\mathrm{SC_{lc}}$ than $\mathrm{SC_{inv}}$ but the difference is not significant. This implies that clients do request newer object copies in the

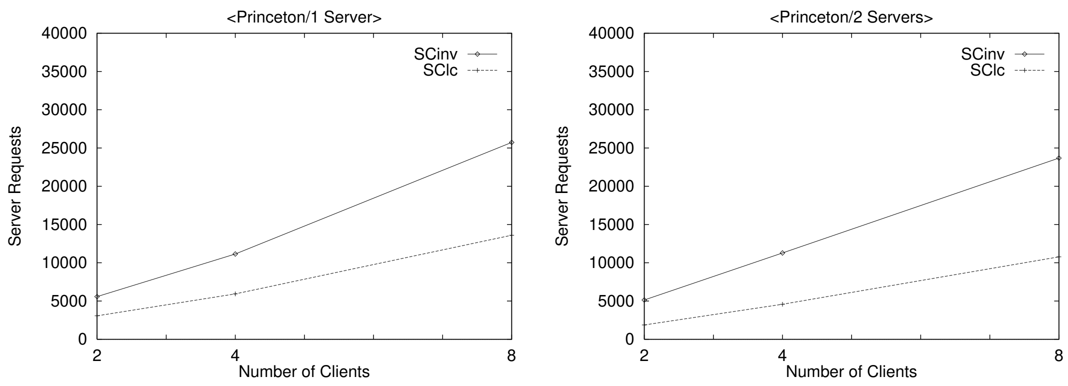

Figure 7: Server requests

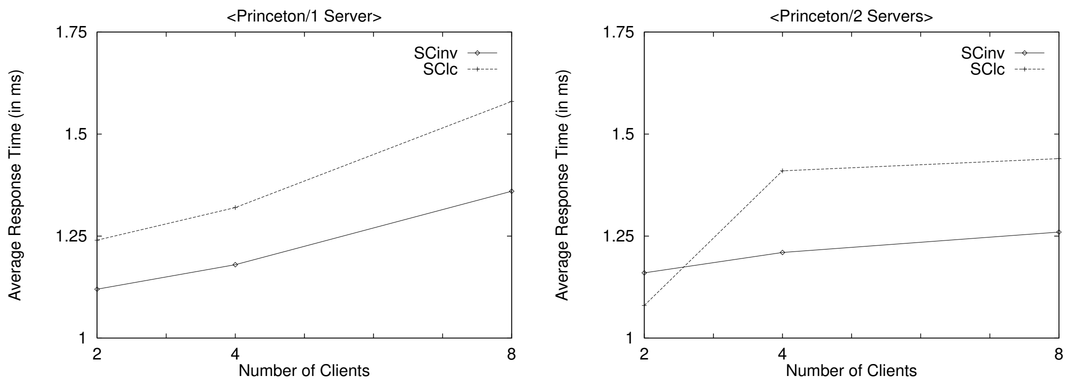

Figure 8: Average response time

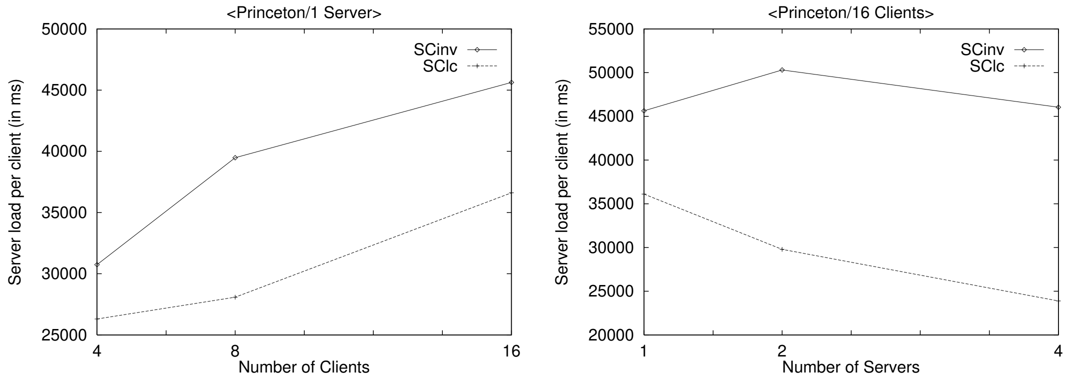

Figure 9: Server load per client

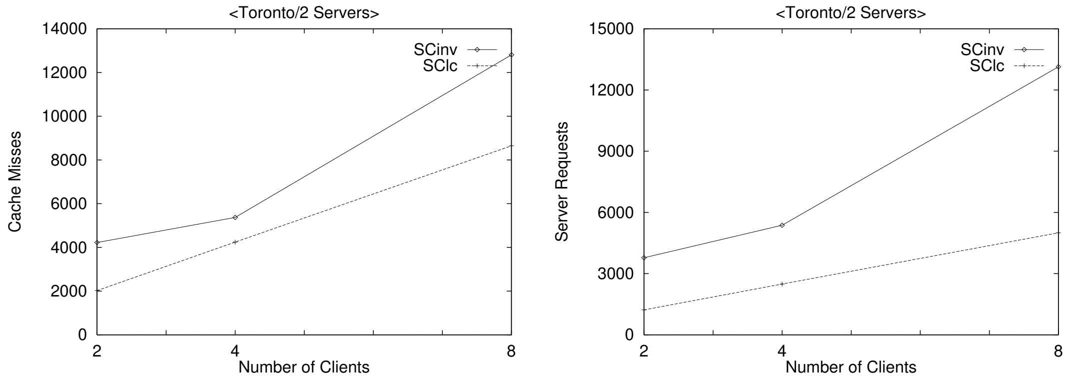

Figure 10: Cache misses / Server requests

SC_{lc} protocol and do not frequently return old values. The difference in the number of server requests between SC_{lc} and SC_{inv} is quite significant as seen in Figure 7. This is expected since in SC_{lc}, when a client's write request reaches the server due to a write-miss or write-fault, the server sends out a message only to the current owner of the object if one exists. In the SC_{inv} case, however, the server sends out invalidations to all clients that have read-only copies in their caches.

Surprisingly, the difference in the above metrics does not show up in the average response times shown in Figure 8. Clearly, SC_{lc} incurs additional computation overhead at client and server sides to do the local consistency checks and to process vector times. Moreover, the network in our case is a 155 Mbs ATM which reduces the effect of comparatively higher number of server requests in the SC_{inv} case. In the experiments conducted, the time taken at the server in the SC_{lc} case is 6% more than in the SC_{inv} case. The response time results are different in the system where wide area communication costs are incurred.

The results presented so far show that while SC_{inv} entails about 51% more messages than

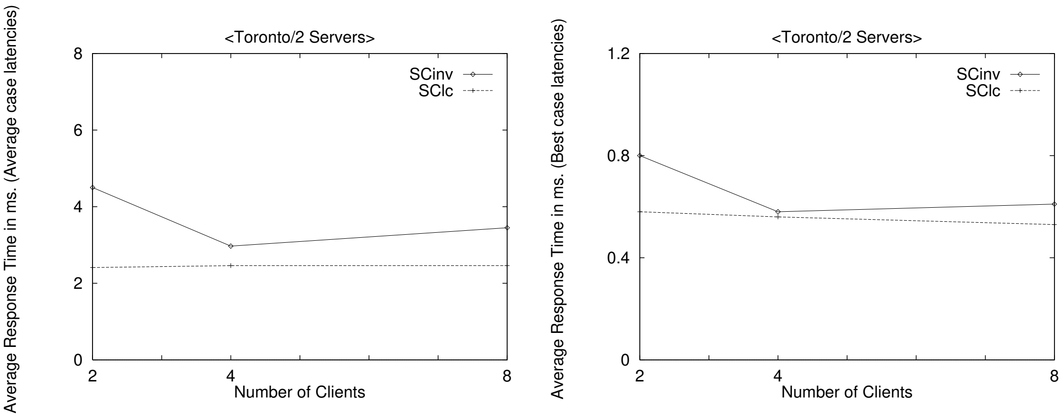

Figure 11: Estimated response times

SC_{lc} on an average, SC_{lc} incurs about 6% more processing overhead than SC_{inv} at the server per client request. However, with increasing network bandwidths, it may be more desirable to reduce server load rather than the number of messages [9]. In Figure 9, we plot server load per client as a product of cache misses experienced by one client and the time spent at the server to service a cache miss. SC_{inv} imposes 12% more load on the server per client than SC_{lc}. Although we did not have a larger system for doing controlled experiments, we believe the difference between the two protocols will become increasingly significant as the number of clients in the system increase.

Geographic Scale: Figure 10 shows the cache misses and server requests respectively for SC_{lc} and SC_{inv} using the Toronto workload. We used the controlled cluster environment to create a trace of cache miss and server request messages. We used this trace in a simulation with wide-area communication costs to determine the access response times[5]. The comparative advantages are similar but the absolute numbers are lower than the Princeton workload because the Toronto workload demonstrates much less sharing. However, this sharing is at the workgroup level. Thus, each of the cache misses and server requests experience inter-workgroup latencies. We calculated the average response times by factoring in the inter-workgroup latencies.

We measured the minimum latency for a Sun RPC round trip between two machines located in two different universities within the same city. The latencies were 5.2 ms and 129 ms for a 12 byte and 16KB messages respectively. Since the workgroups in the Toronto study were either in different locations within the same city or in different cities, we used these times to plot the average response times in Figure 11. In this case, all messages carrying data were assumed to take the time it takes for a 16KB message and all control messages were assumed to take the time it takes for a 12 byte message. Finally, [9] suggests that even if wide-area link bandwidths are no longer a factor and wide-area latencies decrease, the best case scenario would be where the latency is equal to disk access time (we used 14 ms in our studies). We plotted the average response times by factoring in the best-case wide-area latency in Figure 11.

[5]We calculated the average response times by factoring in the measured communication costs. Actual measurements did not provide meaningful results because of lack of control over the environment.

While the observations on the total number of messages in the Princeton workload case hold here also, the difference in average response times are more dramatic. SC_{inv}'s average response time is about 39% worse than that of SC_{lc} in the best case scenario where wide-area link bandwidths are no longer a factor and wide-area latency decreases to a quantity comparable to disk seek time. This demonstrates the advantages of SC_{lc} over SC_{inv} along the dimension of geographic distribution.

9.2.1 Discussion

The results of our performance study suggest that a protocol based on the local consistency approach can indeed scale better than an invalidation based protocol. The first workload (Princeton workload) helped us evaluate the protocols along the system load dimension of scalability. In studies using this workload, the invalidation protocol SC_{inv} entailed up to 51% more messages and 15% more server load per client than our LC based consistency protocol. As the number of users increases, this difference would impact network and server load significantly, thus showing the advantages of SC_{opt} over SC_{inv} along this dimension of scale.

The second workload (Toronto workload) was taken from an industrial setting with users working in workgroup oriented groups spread over several metropolitan areas. This helped us evaluate our protocols along the other dimension of scalability - geographic distribution. In this case, SC_{opt} not only reduces the network load but also significantly decreases the access response time.

To truly demonstrate the scalability of the LC based protocols, we need to experiment with much larger system. Unfortunately such a system where controlled experiments can be done was not available to us. Also, we need to quantify the degree of "staleness" of copies that can be accessed with SC_{lc}. Clearly, the reduced cache misses are due to the fact that copies are not invalidated immediately when an object is updated[6]. In our future work, we will explore the combined overhead of SC_{lc} as well as messages that are needed for advancing node views. Similarly, we will use a larger system and simulations to explore scalability with large number of nodes in the system.

10 Related Work

Our work is related to consistency protocols that have been developed for various systems which include the Web, distributed file systems, distributed object systems and distributed shared memory systems. We enumerate some of the related systems (detailed comparisons are not included due to space constraints). Consistency issues in the context of the Web have been discussed in [17, 29, 40]. The protocols studied include ones based on time-to-live (TTL), which only provides weak consistency, and invalidation and polling based protocols for strong consistency. Although it is argued in [29] that strong consistency can be provided using

[6] Reads can return older values when writes do not result in synchronous invalidation of cached copies. The number of cache misses is not significantly different in the two protocols for many of the data points in the experiments. This suggests that although SC_{lc} does permit stale copies to be accessed, only a small number of accesses actually read old data.

invalidation based protocols, the protocols we develop take a fundamentally different approach where accesses can be serialized; yet, update frequency and currency of cached information can be controlled based on resource availability. Furthermore, we permit multiple writes to an object and ensure consistency across different but related objects. In the evaluation we have done, SC_{lc} provides better performance than the invalidation based protocol SC_{inv} when an application can tolerate some lag in node views and the current global view.

Distributed file systems also employ caching to employ performance. The xFS [8] system makes use of an invalidation based protocol to ensure cache consistency. The call-backs of AFS [19] also have a similar effect. The Bayou [37] and Coda [22] systems address replication, caching and consistency in mobile environments. Bayou addresses several levels of consistency (e.g., session guarantees [36]). Although weaker consistency levels do permit more scalable implementation, we focus on scalable implementations of strong consistency that avoid conflicting updates to an object. Similar to the vector read and write times used by SC_{lc}, Bayou makes use of vector times associated with updates to control the order in which these updates are processed by servers. Also, Bayou mainly considers server replication and does not address partial caching of related objects at clients[7]. We feel that for clients, which have dynamic access patterns, a *pull-based* approach explored by us, where the client is able to dynamically request and cache a subset of objects that are of interest to it, is more appropriate.

The Coda file system employs both server replication and client caching of files. Clients are allowed to work with cached copies to deal with disconnection. It is possible in Coda to have updates that cannot be serialized. Such conflicting updates are detected and can be resolved by the system or reported to the user. The isolation-only transactions of Coda provide serializability by delaying the commit of transactions until they can be validated. We share the goals of Coda by allowing clients to access cached objects without communication with servers. Since we guarantee serialization of all object accesses, our protocol does not allow conflicting updates. The advancing of views in our protocol when a client communicates with a server is similar to reconciliation in Coda.

Thor [28] and Rover [20] employ object caching and strive to provide strong consistency. Conflicting accesses in Rover could lead to consistency violation (the user must repair the effects of such violations). Although Thor allows a transaction to execute with locally cached copies of objects, it ensures serializability of transactions by validating them prior to their commit. In interactive applications, it is desirable that effects of local operations become visible to users as they are executed. Such observation of the operation by the external world is problematic when an optimistic technique such as the one employed by Thor is used. However, when conflicts are not common and they can be resolved when they do occur, the optimistic protocol employed by Thor can provide better performance. Also, we only address consistency at the level of object accesses whereas Thor supports transactions that can include a sequence of accesses.

In scalable shared memory systems (e.g, Stanford DASH [12]) and software implementations of distributed shared memory [21], consistency actions are delayed until certain synchronization operations complete. Our goal in this work is to develop consistency protocols that enhance scalability by reducing communication with a large number of nodes without relying on explicit

[7] Although a server instance can be run at a client node in Bayou, the assumption that a server stores copies of all related objects does not allow a client to cache only the objects that it actually accesses.

synchronization such as locking.

Other systems that address scalable access to objects also exist. The ITV [34] system supports a video-on-demand application and makes use of objects to implement communication between clients and servers. However, caching and consistency issues are not addressed in this system. The Electra system [26] implements replicated objects using group communication. Group communication with atomic ordering is required for strong consistency which is costly. Furthermore, caching and uncaching can result is membership change which require expensive protocols.

11 Concluding Remarks

We have explored scalable consistency protocols that provide flexible coupling between nodes that have copies of related objects while guaranteeing that all accesses can be serialized (e.g., no conflicting updates are done to copies of an object). We have developed a novel local consistency approach that allows protocols to be developed which never require communication or synchronization with all nodes that have copies of an object being accessed. These protocols are based on the philosophy that a node must control what is of interest to it and take local steps to ensure consistency of objects accessed by it. Thus, nodes do not need to be aware of all accesses to objects that are cached by them. This approach also allows nodes to control the currency of cached object copies according to resource availability and application needs.

We have developed protocols for supporting strong consistency that can work with distributed servers and can implement consistency across a group of related objects. By implementing a prototype system that supports caching of distributed objects, we have explored the benefits of the new protocols. Although creating a controlled environment that accurately represents a large scale system was not possible, by developing a sophisticated workload and by conducting a variety of experiments, we are able to show that the flexibility offered by our protocols can provide improved scalability.

Our future work will focus on more detailed evaluation of the protocols by including richer workloads and via simulations. We will also explore the local consistency based approach in the implementation of other levels of consistency.

References

[1] S. V. Adve and M. H. Hill. Implementing sequential consistency in cache-based systems. In *Proc. of Intl. Conf. on Parallel Processing*, 1990.

[2] M. Ahamad, G. Neiger, P. Hutto, P. Kohli and J. Burns. Causal Memory: Definitions and Implementations. *Distributed Computing*, 1995.

[3] M. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, J. K. Ousterhout. Measurements of a distributed file system. In *Proc. of the 13th Symp. on Oper. Sys. Principles*, 1991.

[4] M. Blaze. Caching in large-scale distributed file systems. *PhD Thesis*, Princeton, 1992.

[5] R. R. Bodnarchuk and F. B. Bunt. A synthetic workload model for a distributed system file server. ACM SIGMETRICS 1991.

[6] V. Cate. Alex - a global file system. In Proc. of 1992 USENIX File System Workshop, 1992.

[7] A, Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz and K. Worrel. A hierarchical internet object cache. *1996 Usenix Technical Conference*, Jan. 1996.

[8] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative caching: using remote client memory to improve file system performance. In *Proc. of OSDI*, 1994.

[9] M. D. Dahlin, C. J. Mather, R. Y. Wang, T. E. Anderson, and D. A. Patterson. A quantitative analysis of cache policies for scalable network file systems. In *Proc. of ACM SIGMETRICS*, 1994.

[10] K. Eswaran, J. N. Gray, R. A. Lorie and I. L. Traiger. The notions of consistency and predicate locks in dtabase systems. *Communications of the ACM*, 19,11 (Nov. 1976).

[11] C. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Australian Computer Science Conf.*, 1988.

[12] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *ISCA* 1990.

[13] D. S. Gill, S. Zhou, and H. S. Sandhu. A case study of file system workload in a large-scale distributed environment. *University of Toronto Technical Report*, 1994.

[14] D. S. Gill, S. Zhou, and H. S. Sandhu. A case study of file system workload in a large-scale distributed environment. In Proc. of ACM Sigmetrics, May 1994.

[15] C. G. Gray and D. R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proc. of 12th ACM Symposium on Operating Systems Principles (SOSP)*, 1989.

[16] R. G. Guy, J. S. Heidemann, W. Mak, T. W. Page Jr., G. J. Popek, and D. Rothmeier. Implementation of the Ficus replicated file system. In *USENIX Conf.*, June 1990.

[17] J. Gwerzman and M. Seltzer. World-wide web cache consistency. *1996 Usenix Technical Conference*, January 1996..

[18] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages*, 12(3):463–492, July 1990.

[19] J. H. Howard et. al., Scale and performance in distributed file systems. *ACM Trans. on Computer Systems*, Feb., 1988.

[20] A. D. Joseph, A. F. de Lespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek. Rover: A toolkit for mobile information access. In *Proc. of 15th SOSP*, 1995.

[21] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proc. of the 19th International Symposium of Computer Architecture,* 1992.

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

[23] R. Kordale. System support for scalable services. *PhD Thesis*, Georgia Inst. of Technology, 1997.

[24] R. Kordale and M. Ahamad. Object Caching in a CORBA Copliant System. Usenix Systems Journal, 1996.

[25] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, Sep 1979.

[26] S. Landis, and S. Maffeis. Building reliable distributed systems with CORBA. In *Proc. of USENIX Conference on Object-Oriented Technologies and Systems(COOTS)*, 1995.

[27] A. R. Lebeck, and D. A. Wood. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995.

[28] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber., U. Maheshwari and L. Shrira. Safe and Efficient Sharing of Persistent Objects in Thor. *ACM SIGMOD*, 1996.

[29] C. Liu and P. Cao. Maintaining strong consistency in the World-wide Web. *Proc. of International Conference on Distributed Computing*, May 1997.

[30] J. Misra. Axioms for memory access in asynchronous hardware systems. *ACM Transactions on Programming Languages and Systems*, 8(1):142–153, January 1986.

[31] M. Mizuno, M. Raynal, and J. Z. Zhou. Sequential Consistency in distributed systems. Proc. Intl. Workshop on "Unifying Theory and Practice in Dist. Systems.

[32] F. Mattern. Time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, 1989.

[33] J. Mogul. Hinted caching in the Web. *ACM SIGOPS Workshop*, 1996.

[34] M. N. Nelson, M. Linton and S. Owicki. A highly available, scalable ITV system. *In Proc. of 15th SOSP*, 1995.

[35] M. Raynal, and A. Schiper. From causal consistency to sequential consistency in shared memory systems. *IRISA TR#*: 926.

[36] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. In *IEEE Symp. on Parallel and Dist. Information Sys.*, 1994.

[37] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H.Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *ACM SOSP*, 1995.

[38] F. J. Torres-Rojas and M. Ahamad. Plausible Clocks: Constant size logical clocks for distributed systems. In *Proceedings of the 10th Workshop on Distributed Algorithms (WDAG)*, Italy, October 1996.

[39] R. Y. Wang and T. E. Anderson. xFS: A wide area mass storage file system. In Proceedings of the 4th Workshop on Workstation Operating Systems, Oct 1993.

[40] K. Worrell. Invalidation in large scale network object caches. Master's thesis. University of Colorado, 1994.