An Adaptive Protocol for Implementing Causally Consistent Distributed Services

Mustaque AHAMAD*
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332 U.S.A
mustaq@cc.gatech.edu

Michel RAYNAL and Gérard THIA-KIME
IRISA - Équipe ADP
Campus de Beaulieu
35042 Rennes Cedex France
{raynal,thiakime}@irisa.fr

Abstract

Distributed services that are accessed by widely distributed clients are becoming common place (*e.g.*, especially services targeted to the home). Such services cannot be provided at the needed level of performance and availability (1) without replicating the service at multiple nodes, and (2) without allowing a "relatively" weak level of consistency among replicated copies of the state of a service. The latter is especially important in large scale systems where communication latencies can be high and communication may not be possible between service instances at all times. In this paper we explore causally consistent distributed services when multiple related services are replicated to meet performance and availability requirements. This consistency criterion is particularly well suited for some distributed services (*e.g.*, cooperative document sharing), and it is attractive because of the efficient implementations that are allowed by it. We develop a new protocol for implementing causally consistent services that has several novel features. First, service instances can be created and deleted dynamically according to service access patterns in the distributed system. Also, when different services are related and consistency is needed across replicated copies of such services, our protocol handles the case where different but related services are replicated independently. Another novel aspect of our protocol lies in its ability to use both push and pull mechanisms for disseminating updates to objects that encapsulate service state.
Key-words: Distributed Services, Causality, Causal Consistency, Client-Server Distributed System.

1 Introduction

Services that are accessed by widely distributed clients are becoming common place (*e.g.*, especially services targeted to the home). Such services cannot be provided at the needed level of performance and availability without replicating the service at multiple nodes of a distributed system. In fact, it is desirable to have an instance of the service located in a neighborhood of clients so access latency and communication costs can be reduced. Since the replicated service instances should still function as one logical service, coordination among service instances is required for correct operation. In this paper, we are interested in developing an adaptive protocol for distributed services that allows service instances to be created and deleted dynamically to match the service access patterns. We consider

*This work was done while the author was visiting IRISA and was supported by an INRIA grant. It was also supported in part by ARPA contract DABT-63-95-C-0125, and by NSF grants CDA-9501637 and CCR-9619371.

systems where services are encapsulated within objects; in that way, the state of a service is defined by the state of the corresponding objects (we will use the words service and object interchangeably). Three main issues have to be solved in such a context.

- *Internal Consistency of a Service.* Replicated instances of a given service must be coordinated in order for the service to be consistent for its users. This is the classical problem of consistency among the replicas of an object.
- *Adaptivity of a Service.* In order to reduce access cost, instances of a given service (*i.e.*, copies of an object) can be created or deleted dynamically. As indicated previously, many clients in a neighborhood may frequently access a service that does not have an instance close to them. It should be possible to create a new service instance at a server node close to these clients. On the other hand, when a service instance is not used much, it may be desirable to delete this instance to reduce the cost of coordination among the services instances.
- *Mutual Consistency among a set of Services.* Services are mutually related because one service may depend on another one. This means that due to client actions, the state of a service can be made dependent on the state of another one (*e.g.*, updating an object from the value of another one). Mutual consistency considers a collection of related services and defines consistency across a set of such services.

Many consistency criteria have been developed for replicated objects. Linearizability [12] is the most used (most of the time in an implicit way) consistency criterion. Intuitively, it says that when reading an object, the value returned must be the last one that has been written into the object, where "last" refers to physical time. This consistency criterion can be weakened by considering "last" referring to logical time; in that case, we obtain sequential consistency [18]. These two consistency criteria are called *strong*. When considering a distributed service, strong consistency criteria could limit scalability of the service as protocols implementing them impose a very strong synchronization on accesses; moreover, strong consistency may not allow requests to be processed in certain conditions when communication cannot be completed (*e.g.*, disconnection). So, in this paper we explore a weaker consistency criterion for distributed services that allows efficient implementations of replicated services.

Two different approaches have been explored for weaker consistency of replicated objects. In weakly ordered shared memory systems, accesses to shared objects are classified as synchronization accesses and data accesses. It is assumed that programs are written such that synchronization accesses, which are strongly consistent, interleave with data accesses such that data races are avoided [7, 1, 10]. When this is assumed, it is not necessary to update copies of a data item at the time it is written. Thus, systems such as release consistency [10], ensure consistency of shared object copies only when certain synchronization operations complete (e.g., release on a lock). Such memory systems provide sequential consistency like guarantees to applications that are properly synchronized or data-race-free. While such assumptions about properly synchronized programs are valid in parallel applications, we take a different approach that does not rely on synchronization operations in defining consistency of replicated objects. We take this approach because (1) locking may not be desirable in large scale systems when sometimes communication may not be possible between a node holding the lock and another one requesting it, and (2) ensuring that applications have sufficient synchronization so they are data-race-free is complex. We explore *causal consistency* (CC) because it appears that it has two main advantages: (1) CC is meaningful for several applications [2, 9, 16, 30], and (2) protocols that implement CC require only weak synchronization that permits the design of efficient implementations.

Thus, in this paper we explore the design of a protocol that can be used to provide causally consistent distributed services.

There exist protocols that implement causal consistency across related objects when (1) each node has copies of all such objects, and (2) the set of nodes that have object copies is static. We outline one such protocol in the next Section to motivate the new protocol that is developed by us. Our protocol is adaptive because it allows the system to adapt to service request patterns by creating and deleting service instances. Furthermore, we allows different objects to be replicated independently and hence different nodes may have copies of different subsets of related objects (e.g., partial replication). These requirements have led us to the following model of a distributed system in which three sets of nodes are associated with each service (*i.e.*, or the objects that implement the service):

- A set of nodes called *Permanent Servers*. Each of these nodes permanently manages an instance of the replicated service. This set constitutes the "core nodes" implementing the service. A permanent server of a given object has an initial copy which is not deleted. Permanent servers ensure that a certain level of availability can be assumed for the service.

- A set of nodes called *Potential Servers*. For a given object, a potential server may or may not have a copy of it at a given time. A copy of the object can be created or destroyed at such a node based on efficiency reasons that define a creation/deletion policy. The potential servers allow a service to adapt to request patterns by creating instances where the service is accessed frequently. This could improve locality, load sharing and reduce communication costs, leading to improved performance.

- A set of nodes called *Clients*. For a given object, a client cannot have a copy of it. This could be due to access control reasons or because of the limited amount of resources available at the clients (*i.e.,* a client may be a cheap network computer (NC) that is used to access remote services). A client can access services provided at permanent or potential servers by using the classical remote procedure call (RPC) mechanism or remote object method invocations as defined in object-oriented systems.

To summarize, the following are the main contributions of the paper:

- Design of an adaptive protocol that allows service instances to be created and deleted dynamically at potential servers (*e.g.*, according to the load and the access patterns of the service).

- Investigation of causal consistency for a set of inter-related services when the services are independently replicated. In other words, related services may have different sets of permanent and potential servers.

The paper is divided into seven sections. Section 2 defines causal consistency for distributed services. Section 3 presents the system architecture. Section 4 describes the protocol implementing causal consistency for a collection of related distributed services. Section 5 presents discussion of the protocol, including performance related issues. Section 6 compares the proposed approach with previous works. Finally Section 7 concludes the paper.

2 Causal Consistency

Causal consistency (CC), which is based on causality in distributed systems [17], has been explored in a number of contexts [16, 5, 2, 23, 15]. These papers discuss several applications which execute

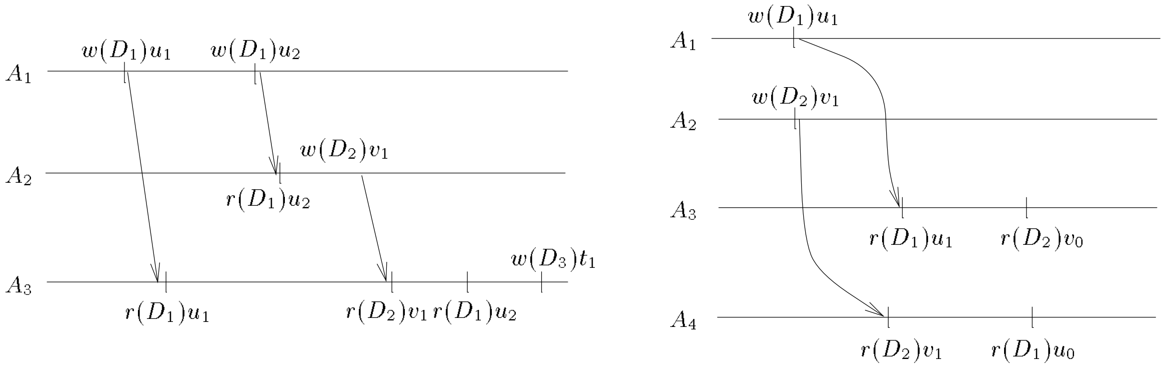

Figure 1: (a) Causally Consistent Reading, and (b) Concurrent Writes and Causal Reads.

correctly with the consistency guarantees provided by CC. Before reviewing the definition of CC, we first motivate it using a document sharing application.

2.1 Motivation

Consider a document sharing system that is used to write a paper cooperatively by a number of authors. Different authors may be responsible for writing different sections of the paper. First consider the execution scenario shown in Figure 1(a). A_1, A_2 and A_3 are the three coauthors that are cooperatively working on various documents that make up the paper. Assume that author A_1 writes a document D_1 that describes an experiment. Author A_2 reads the experiment description and writes the results of the experiment in document D_2. Author A_3 is responsible for writing document D_3 which presents a discussion of the results. Author A_3 needs to read both D_1 and D_2 before writing D_3. Assume that the authors can revise the documents several times. CC guarantees that the authors see the documents in a causally consistent manner. For example, when author A_3 has read value v_1 of D_2, CC guarantees that a future read of D_1 by it will return the revised version u_2 of the document. Thus, even if A_3 had a copy of the older version of D_1 (A_3 may have previously read the older value u_1 of D_1), after the read of D_2, CC ensures that A_3 will not read u_1 because this value has been causally overwritten. Although this example illustrates how consistency requirements are defined in CC, it does not show the flexibility provided by CC which distinguishes it from strong consistency (in fact, the execution in Figure 1(a) is also possible in a system with strong consistency).

Figure 1(b) shows another execution that is only possible in a causally consistent system. This shows four coauthors sharing documents D_1 and D_2. These two documents are written concurrently by authors A_1 and A_2. For example, one could be the motivation for a problem explored in the paper and the second one could be the problem description itself. Two other authors, A_3 and A_4 read these documents to review them and also to write other parts of the paper based on the contents of these documents. In the execution shown, A_3 is able to read the update u_1 of D_1 made by A_1 but the update v_1 by A_2 to D_2 is not seen by A_3 even after it reads D_1 (the value v_0 returned by the read of D_2 is an initial value). Author A_4 sees the updates in the reverse order; it can read changes made to D_2 by A_2 but not the updates made by A_1. These executions are acceptable because the two documents are written concurrently and hence neither A_3 nor A_4 makes any assumptions about which document will be written first. CC exploits this and allows A_3 and A_4 to see these updates in different orders. On the other hand, strong consistency criteria such as sequential consistency or linearizability will not allow this execution because they require that the execution of all operations should be serializable.

Disallowing such executions, which are acceptable in the application domain, requires unnecessary and expensive synchronization that is avoided by CC.

CC is attractive not only because it can meet the sharing needs of many applications but it can also be implemented efficiently. It is possible to complete read and write accesses to causally consistent objects without synchronization among nodes that store copies of the objects. This can lead to scalable protocols because coordination among a large number of nodes is not necessary with causally consistent replicated objects.

CC provides efficient implementations but it does not disallow concurrent updates to an object at different nodes. Furthermore, if the object is concurrently updated with two different values v and v', copies at some nodes may store v while others may store v'. Thus, different copies of the object may not converge to the same value even when there are no more updates to the object. Despite this problem, we believe that CC is useful at the level of object accesses because of the following reasons:

1. In many applications, especially ones that come from the CSCW domain, implicit social protocols are employed which make concurrent updates unlikely. Furthermore, such applications require good response time (e.g., effects of local actions should become visible immediately) which makes it difficult to provide strong consistency. As a result, systems such as COAST [26], DECAF [27] and Villa [4] use optimistic schemes to provide good response time. These systems provide eventual consistency by resolving conflicts when they do arise. CC can be the base consistency level for applications enabled by such systems and conflict resolution techniques can be employed at a higher level when concurrent updates are ordered differently at different nodes.

2. Applications may employ explicit synchronization such as locking to ensure consistency for a sequence of accesses. Such synchronization can prevent concurrent updates. Since accesses to locks must be strongly consistent, lock operations will block when the lock holding node becomes disconnected. Thus, it may be desirable that synchronization is done for only update operations. CC can allow nodes to read causally consistent object copies without acquiring locks which can enhance concurrency without creating update conflicts. If there are no concurrent updates (e.g., concurrent-write-free application), it has been shown that executions permitted by CC can also be obtained in a system that provides strong consistency [2, 24].

2.2 Definition of Causal Consistency

We consider a system composed of a finite set of sequential processes (nodes) $s_1, s_2, \ldots, s_n$ which interact through a finite set O of shared objects. Each object $x \in O$ can be accessed by a read or a write operation. A write into an object defines a new value for the object; a read allows a process to obtain a value of the object. The execution of a write operation that assigns the value v into object x is denoted $w(x)v$ (for simplicity, and without loss of generality, we assume all values written into an object are different). The execution of a read operation of the object x, that returns value v is denoted $r(x)v$.

The execution of process s_i is modeled as the sequence $op_i^1 op_i^2 \ldots op_i^k \ldots$ where op_i^k denotes the k-th operation executed by s_i (we simply use *opi* to denote an operation when we do not need to know what process executed the operation). Such a sequence defines the local history $\widehat{h_i}$ of s_i. Let h_i denote the set of operations executed by s_i and $\rightarrow_i$ be the total order relation on operations issued by s_i. $\widehat{h_i}$ is the totally ordered set $(h_i, \rightarrow_i)$.

An *execution history* (or simply a history) of the system is a partial order $\widehat{H} = (H, \rightarrow_H)$ such that:

- $H = \bigcup_i h_i$
- $op1 \rightarrow_H op2$ ($op1$ is causally ordered before $op2$) if:

  (i) $\exists$ P_i: $op1 \rightarrow_i op2$ (in that case $\rightarrow_H$ is called *process-order* relation)

  (ii) $op1 = w(x)v$, $op2 = r(x)v$ such that $op2$ reads the value written by $op1$ (in this case $\rightarrow_H$ is called *read-from* relation)

  (iii) $\exists$ $op3$: $op1 \rightarrow_H op3$ and $op3 \rightarrow_H op2$ (transitivity)

Two operations $op1$ and $op2$ are *concurrent* in $\widehat{H}$ if $\neg(op1 \rightarrow_H op2)$ and $\neg(op2 \rightarrow_H op1)$.

CC requires that operations read values of objects that are not *causally* overwritten[1]. More precisely, if operation $op2 = r(x)v$ reads a value written by operation $op1$ (*i.e.,* $op1 = w(x)v$), it should not be the case that there exists another operation $op3$ such that $op3$ is an operation on x with value different from v and $op1 \rightarrow_H op3$ and $op3 \rightarrow_H op2$. In other words, the value read by $op2$ has not been overwritten by another operation that causally follows $op1$ and precedes $op2$. This definition of CC allows each process to view the execution of its operations with respect to update operations of other processes in an order that respects causality. Concurrent operations can be viewed in different orders by different processes. For example, in the execution shown in Figure 1(b), the concurrent update operations executed by A_1 and A_2 are viewed in different orders by A_3 and A_4. The $w(D_1)$ operation is ordered before $w(D_2)$ by A_3 whereas A_4 sees these operations in the reverse order.

Notice that the characterization of CC is independent of the assumed system environment and the particular implementations that are used to ensure it. The read and write operations may be executed locally or remotely by communicating with one or more other nodes. Only the values written by update operations and returned by read operations and the local order in which nodes issue the operations are used in the definition of causal consistency.

2.3 An Implementation with Full Replication

To show that efficient implementations are possible for CC, we review a simple implementation of CC in an environment where all nodes have copies of all the objects and these nodes form a fixed set. Basically, it extends to objects the protocol introduced in [23] that implements causal delivery of messages. Each node s_i is endowed with a version vector $v_i[1..n]$ (where n is number of nodes) such that: $v_i[j]$ = # object updates issued by s_j and known by s_i. v_i is similar to vector clocks defined in [19, 8] and used is a variety of distributed algorithms (e.g., CBCAST delivery [5]). The protocol is the following one.

- read by s_i of an object x:
  s_i reads the local copy of x and returns it.
- write by s_i of an object x with value val_x:
  $v_i[i] := v_i[i] + 1$;
  update the local copy of x with the new value val_x;
  broadcast to all other nodes $update(x, val_x, i, v_i)$.
- when s_i receives a message $update(x, val_x, j, v)$;
  delay the processing of the message until $(v_i[j] + 1 = v[j]$ *and* $\forall k \neq j : v_i[k] \geq v[k])$;
  $v_i[j] := v_i[j] + 1$;

[1] This definition of causal consistency is the one used in [24]. It is slightly different from the one introduced in [2].

update local copy of x with new value val_x.

A proof of this protocol can be found in [2, 23]. It is interesting to note that this protocol provides wait-free implementation of read and write operations: no synchronization is necessary to schedule a read or a write. This is no more the case when the consistency criterion is stronger than causal consistency (*e.g.*, sequential consistency and linearizability) or when objects are not fully replicated. Our goal in this paper is to develop a protocol that provides efficiency for object accesses similar to this simple protocol. However, the assumed environment is more complex because object copies can be created and deleted dynamically and replication of objects may be partial (*i.e.,* not all nodes may have copies of all objects).

3 System Model

3.1 The Underlying Distributed System

The underlying distributed system consists of a set of nodes that communicate by exchanging messages through a network. There is neither shared memory nor a common physical clock. Moreover, each node proceeds at its own speed, communication delays are arbitrary and the network is assumed to be reliable (*i.e.*, no node crash, no message loss). Communication channels are not necessarily FIFO. Albeit important, system failures are not considered here. Thus, the underlying distributed system corresponds to the well-known asynchronous reliable distributed model.

3.2 The Case of One Service (A Single Object)

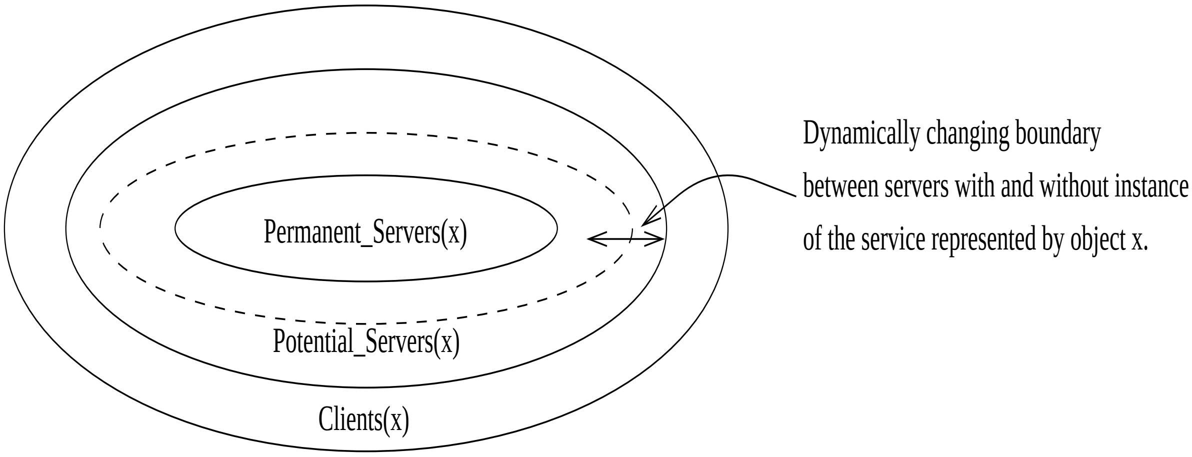

Figure 2: Classes of Nodes for an Object x.

We first consider the simple case where there is a single object that is replicated. Let this object be x. As described in the introduction Section, three sets of nodes are associated with x (Figure 2).

- *Permanent_Servers(x)* is a statically defined set of nodes $\{s_i, s_j, \ldots\}$. Each of these nodes manages a copy of x and all nodes cooperate to keep the copies consistent. This set constitutes the base implementation[2] of x. In Figure 2, this set is represented by the most interior oval.

[2] When considering failures, only a failure or a recovery can modify this set.

- *Potential_Servers(x)* is a statically defined set of nodes $\{s_k, s_l, \ldots\}$. Each of these nodes is not initially provided with a copy of x but it can create or delete (when it has one) such a copy. At any time during the lifetime of the system, this set is partitioned in two subsets (separated by a dotted oval in Figure 2): the members of *Potential_Servers(x)* that have a copy of x and the members that do not currently have its copy.

  Creation of a copy of x by a node of *Potential_Servers(x)* is ruled by a policy that can be defined by the system designer (*e.g.*, if there is a user of x on this node, if a client node requests it to access x, if creation of a copy at this node decreases the load of the network, etc). Section 4 describes the protocol (mechanism) executed when a potential server creates a local copy of an object.

  Deletion of a copy can be defined either by the system designer (*e.g.*, when the copy is no more accessed or requested at this potential server) or by the protocol implementing consistency (see the next subsection and Section 4).

- *Clients(x)* represents the (dynamically defined) set of nodes that do not belong to *Permanent_Servers(x)* $\cup$ *Potential_Servers(x)* but can access the object x. A node in *Clients(x)* accesses x by addressing a permanent or potential server of x through a RPC-like mechanism. Potentially, all nodes of the system that are neither a permanent server nor a potential server of x constitute *Clients(x)*.

From an operational point of view, these three sets of nodes for object x can be seen in the following way: *Permanent_Servers(x)* represents the set of nodes acting as a "stable" memory implementing the consistency and the permanence of x; *Potential_Servers(x)* represents the set of nodes that are allowed to have a "cached value" of x while *Clients(x)* represents the set of nodes that can remotely access x but are not allowed to maintain a copy of x (due to access control or insufficient resource reasons).

According to the previous system model, each potential server s_i of x has the following variables:

- A flag $present_i(x)$ that indicates if s_i currently has a copy of x. When it has one ($present_i(x) = yes$), the protocol ensures that this copy is causally consistent.

- A pointer $attached\_to_i(x)$ that indicates the member of $Permanent\_Servers(x)$ to which s_i sends requests for a copy of x when it needs one. In this paper we assume that the value of this variable remains constant but it could dynamically change to adapt to load changes or to failures of permanent servers.

Similarly, each permanent server s_i for x has the following variable:

- A set $provider\_for_i(x)$ which includes all potential servers of x that receive their copy of x from s_i. So, we have: $s_j \in provider\_for_i(x) \Leftrightarrow attached\_to_j(x) = s_i$.

Lastly, each potential or permanent server for an object x manages the following variable:

- A node name $last\_writer_i(x)$ that contains the identity of the (permanent or potential) node that produced the value of x currently stored by s_i (when s_i is a potential server with $present_i(x) = no$, this value is irrelevant).

3.3 The Case of Multiple Related Services (Objects)

In general, services may be related to each other. For example, a document sharing service may depend on a name service and a file service. This introduces consistency requirements across copies of multiple objects that correspond to the different but related services. Let us consider Figure 3 where permanent and potential servers of two related objects x_a and x_b are represented (dotted lines have the same meaning as in Figure 2). We have :

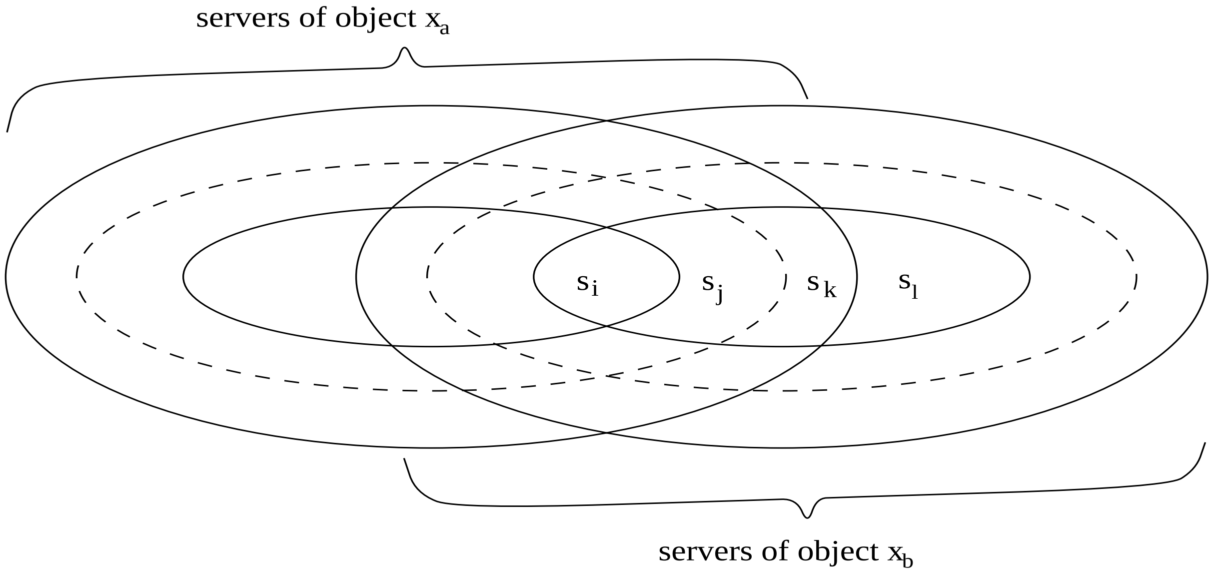

Figure 3: Servers for Two Objects.

- node s_i is permanent server for both x_a and x_b.
- node s_j is potential server (with a copy) for x_a and permanent server for x_b.
- node s_k is potential server (without a copy) for x_a and permanent server for x_b.
- node s_l is a client for x_a and a permanent server for x_b.

A server node for one object (*e.g.*, s_l) can be a client for another object. We will consider in the following, without loss of generality, only nodes that are permanent or potential servers for at least one object in a set of related objects.

The fact that there are distinct sets of servers with copies of different but related objects can create a mutual consistency problem that the underlying consistency protocols must solve. The problem is the following one. Let s be a node that is a server with copies of the two objects x_a and x_b, and let us consider the following situation: s receives an update for x_b and this update depends[3] on some previous update to x_a that has not yet been applied to its copy of x_a. How should s process this update? Two cases are possible according to the status (permanent/potential) of s with respect to x_a:

[3]The precise nature of this *dependence* is related to the consistency criterion. We will see in Section 4.1 how this dependence can be tracked in the case of causal consistency.

- s is a permanent server for x_a (that is the case of s_i in Figure 2). In this case, s has to wait for the update of x_a to arrive before processing the update of x_b. This is necessary to avoid a violation of causal consistency as a result of accessing the old value of x_a after the new value of x_b is read.
- s is a potential server for x_a (that is the case of s_j in Figure 2). In that case, s can invalidate its copy of x_a and process immediately the update of x_b. This ensures that the causally overwritten copy of x_a cannot be accessed at the node. If x_a is accessed by s in the future, a newer copy has to be fetched by it.

The consistency test that solves this problem constitutes the core of the protocol described in the following section.

4 The Consistency Protocol

Section 2 has introduced the consistency criterion (namely causal consistency) and Section 3 has presented a system model for replicating a collection of related distributed services. This Section presents a protocol implementing this consistency criterion in this architecture.

4.1 Tracking Causality

The most fundamental data structure to track causality relations and implement causally consistent objects is a two dimensional matrix of integers [9, 30, 5, 23]. Although a version vector is sufficient for tracking causality in the full replication case that was discussed previously, partial replication requires that updates to objects be recorded separately. For example, consider server node s_i with copies of objects x_a and x_b, and server node s_j with copies of objects x_b and x_c. Assume that s_i updates x_a and then x_b, and sends the updated value of x_b to s_j. If only a version vector is used, s_i will increment its entry twice when it performs the two updates. When s_j receives the updated value of x_b, it will not be able to know if the two updates were both to x_b or one was to an object that s_j does not store. In the former case, s_j has to wait for the previous update to arrive whereas such an update will never arrive in the latter case because s_j does not store a copy of the updated object. To be able to distinguish between these cases, the updates to different objects have to be recorded separately which requires a matrix structure for tracking the causality information.

Let $O = \{x_1, x_2, ..., x_m\}$ be the set of related objects across which consistency needs to be provided and $S = \{s_1, s_2, ...s_n\}$ be the set of all nodes that are permanent or potential servers for at least one object in O. $CM_i[1..m, 1..n]$ is the causality matrix[4] managed by a node s_i in S. Its meaning is the following one:

$$CM_i[a, j] = \# \text{ updates to } x_a \text{ issued by } s_j \text{ and known by } s_i.$$

It is important to note that $CM_i[a, j]$ remains equal to 0 when $s_j \notin Permanent\_Servers(x_a) \cup Potential\_Servers(x_a)$ (those entries of the matrix can be saved; this can allow a compact representation of the matrix). If for x_a, the node s_i is a permanent server or a potential server with a copy,

[4]The causality matrix has entries for only those nodes that are a potential or permanent server for at least one object in a group of related objects. Entries are not kept for nodes that are clients only. We assume that pure clients do not directly communicate with each other and hence causality information is not propagated by communication between clients.

then the row $CM_i[a,*]$ represents the "version" vector[5] of the copy of x_a currently owned by s_i. It has to be kept in mind that this matrix is a conceptual tool that makes it easy to describe the protocol; the cost of maintaining this data structure and optimizations that can be used to reduce this cost are described in Section 5.

4.2 Mechanism for Propagating Updates

To maintain consistency of object copies, a node has to systematically add new values of objects that it stores. Such values can be received in messages that are sent to propagate new values of an object that is written by a server, or a potential server could request an object copy when it does not have one. The propagation of updates to other servers as well as the processing of such messages depends on whether a node is a permanent or potential server for an object. Since a permanent server of an object x_a provides long-term storage for it, updates to x_a need to be sent to its permanent servers. On the other hand, a potential server for x_a only needs to receive updates to x_a when it has its copy and even in this case, the updates may not be sent if the server fetches consistent values when it needs them.

A new value of object x_a received at server s_i also brings new causality information based on when the value was written. As indicated in Section 3.3, some of the object copies existing at s_i could become inconsistent according to this causality information. If s_i is a permanent server for a set of objects, it needs to maintain their copies and hence must wait for causally preceding updates to arrive before the new value of x_a is added. Since s_i may not receive updates for objects for which it is a potential server, it is not correct to wait for those updates. Our protocol handles this problem by removing overwritten copies of such objects when it processes the updated value of x_a. Since how updates are propagated and processed constitute the most important parts of the protocol, we describe them in detail.

Updates from Potential Members to Permanent Members

When a potential member s_j for x_a has its copy and updates x_a with a new value new_val_a, it increments by 1 the entry $CM_j[a,j]$ and sends the message $newvalue(a, new\_val_a, CM^a)$ to the permanent member $attached\_to_j(x_a)$. Matrix value CM^a is the current value of CM_j and constitutes the causal timestamp of this update.

Let us now consider how a node s_i (member of $Permanent\_Servers(x_a)$) handles a message $newvalue(a, new\_val_a, CM^a)$ which is received from the node s_j (by construction, s_j belongs to $provider\_for_i(x_a)$). Then, s_i broadcasts the message $update(j, a, new\_val_a, CM^a)$ to all the members (including itself) of $Permanent\_Servers(x_a)$.

Updates from Permanent Members to Permanent Members

When a permanent member s_i for x_a updates its copy with a new value new_val_a, it increments by 1 the entry $CM_i[a,i]$ and broadcasts message $update(i, a, new\_val_a, CM^a)$ to all the members of $Permanent\_Servers(x_a)$ (including itself). As in the previous case, this ensures all permanent servers

[5] Such version vectors are used in [21] to detect mutual inconsistencies among the copies of a single object. It is interesting to note that the strong synchronization necessary to implement consistency criteria (such as sequential consistency) stronger than CC makes it possible to shrink each version vector to a single version number (that is equal to the sum of all components of the corresponding vector).

for x_a will receive the *update* message informing them that the node s_i has written a new value for x_a. How a permanent server processes this message is described in the next section.

Updates from Permanent Members to Potential Members

Two nodes that are potential servers for an object x_a do not communicate directly as far as x_a is concerned. Updates of x_a are disseminated to them from permanent servers of x_a. To do this dissemination two different mechanisms are possible according to the access pattern of the object at the potential server.

- In the *push* based approach a permanent server node s_i that receives an *update* message for x_a systematically forwards this message to the nodes included in $provider\_for_i(x_a)$ which currently are accessing a copy of x_a. This is desirable when the potential servers attached to s_i frequently use the new values of the object.

- In the *pull* based approach a node s_i that is a permanent server for x_a does not forward update messages to members of $provider\_for_i(x_a)$. It is up to a potential server s_j to send a request to the permanent server node $attached\_to_j(x_a)$ to get a more up to date value of x_a. This could provide better performance if s_j accesses x_a infrequently and hence does not want to process all updates to x_a.

Both mechanisms can coexist. Some objects can be updated with the push based approach while others are updated with the pull based approach.

4.3 To Wait or to Invalidate

Consistency checks are based on vector comparisons. A vector is actually a row of a causality matrix: the vector $CM1[b,*]$ designates the *b-th* row of $CM1$ (*i.e.,* a version vector associated with a value of object x_b). The following three comparison operators are used (*greater than or equal*, *greater than* and *concurrent*, respectively):

- $(CM1[b,*] \geq CM2[b,*]) \equiv (\forall k : CM1[b,k] \geq CM2[b,k])$
- $(CM1[b,*] > (CM2[b,*]) \equiv ((CM1[b,*] \geq CM2[b,*])$ **and** $(CM1[b,*] \neq CM2[b,*]))$
- $(CM1[b,*] \mid\mid CM2[b,*]) \equiv (\neg(CM1[b,*] \geq CM2[b,*])$ **and** $\neg(CM2[b,*] \geq CM1[b,*]))$

The reader can check that we have from these definitions:
- $\neg(CM2[b,*] \geq CM1[b,*]) \equiv ((CM1[b,*] > (CM2[b,*])$ **or** $(CM1[b,*] \mid\mid CM2[b,*]))$

In our implementation these vector comparisons have the following meanings. Let us consider two updates of x_b timestamped $CM1$ and $CM2$, respectively.

- $CM1[b,*] > CM2[b,*]$ means that the value of x_b associated with the timestamp $CM2$ is causally overwritten by the value associated with the timestamp $CM1$.

- $CM1[b,*] \mid\mid CM2[b,*]$ means that the value of x_b associated with the timestamp $CM2$ and the one associated with the timestamp $CM1$ have been produced by two concurrent updates.

- $\neg(CM2[b,*] \geq CM1[b,*])$ means that the value of x_b associated with timestamp $CM2$ is either causally overwritten (*i.e.*, $CM1[b,*] > CM2[b,*]$) by or concurrent (*i.e.*, $CM1[b,*] \,||\, CM2[b,*]$) with the update that produces the value with timestamp $CM1$.

Let us consider a node s_i that receives a message $m = update(j, a, new\_val_a, CM^a)$. The previous section has described how this message is forwarded to other nodes. This section describes how s_i processes this message. Note that in such an *update* message j is the identity of the node that produced the updated value contained in the message.

The problem that has to be solved has been mentioned at the end of Section 3.3. Informally:

- First, for all the objects x_b for which s_i is a permanent server ($s_i \in Permanent\_Servers(x_b)$), s_i has to delay the update of the local copy of x_a in order to avoid violation of causal consistency for x_b. This is expressed by statement (S1) (Figure 4).

- Second, for all the objects x_c such that $s_i \in Potential\_Servers(x_c)$ *and* $present_i(x_c) = yes$, s_i can immediately invalidate their local copy if those copies can violate causal consistency (due to the copy of x_a being updated with the new value contained in the message). This is expressed by statement (S2) (Figure 4).

The causality relations currently known by s_i (they are described by CM_i) and those piggybacked by the message m (namely, CM^a), allow a formal description of *update* message processing as described in Figure 4. The **delay** statement delays the processing of the concerned message till some conditions are satisfied; those conditions (conditions C1, C2, C3 and C4 in statements S1 and S2) are on the value of CM_i. When the processing of an *update* message is delayed, other *update* messages can be processed if their conditions are satisfied. (Lines of comment are preceded by %.)

Statement (S1) ensures that all updates that causally precede the value of x_a in m (*i.e.*, precede the write operation that produced this value) and that concern objects for which s_i is a permanent server are processed before m. This guarantees causal consistency for objects for which s_i is a permanent server. If s_i is a potential server for x_a and the received value of x_a is causally overwritten according to the information in CM_i (see the test $CM_i[a,*] > CM^a[a,*]$), it is discarded and no processing is necessary. This is possible because s_i need not process all updates to objects for which it is a potential server[6].

Basically statement (S2) is similar to statement (S1). It ensures that causal consistency is not violated for objects for which s_i is a potential server. Let us consider an object x_c for which s_i is a potential server. If s_i was a permanent server for it, s_i should wait until all causally preceding updates have been applied to its copy of x_c (see condition $C2$). As s_i is only a potential server for x_c, it does not have to wait for these updates to arrive; the price it has to pay to ensure causal consistency is only to invalidate its current copy of x_c, so the condition $C4$ follows. The statement (S2) uses a very conservative policy (condition C4) to delete local copies of objects. A less conservative policy could be used by replacing C4 by the less constrained condition $C4'$ where $C4'$ **is** ($present_i(x_c) = yes$ **and** $CM^a[c,*] > CM_i[c,*]$).

Statement (S3) updates the control and data context of s_i according to the message m received.

It is interesting to note that in the particular case where there are no potential servers for any object (*i.e.*, $\forall\, x_a \in O : Potential\_Servers(x_a) = \phi$), the second part of the statement (S1) and the

[6]As the received value for x_a has been defined by s_j the following simpler test, namely, $CM_i[a,j] > CM^a[a,j]$, can be used instead of $CM_i[a,*] > CM^a[a,*]$ to determine if this new value is causally overwritten.

when s_i **receives** $update(j, a, new\_val_a, CM^a)$
begin

% (S1) *Delay Part:*
% *ensures causality of updates on objects for which s_i is a permanent server is not violated*
case
$s_i \in Permanent\_Servers(x_a)$ **then**
delay the processing **until** ($C1$ **and** $C2$) **where**
% $C1$ states that all updates to x_a that causally precede this one have been
% locally reported (note that $C1$ is the same condition as the one used in the
% case of full replication, see Section 2.3, with the vector $CM_i[a,*]$ replacing v_i).
$C1$ **is** ($(CM_i[a,j]+1 = CM^a[a,j])$ **and** $(\forall k \neq j:\ CM_i[a,k] \geq CM^a[a,k])$)
% $C2$ states that all updates on all objects x_b, distinct from x_a and for which
% s_i is a permanent server, that causally precede this update have been locally
% reported (these updates are revealed by the vector $CM^a[b,*]$).
$C2$ **is** ($\forall x_b : (x_b \neq x_a$ **and** $s_i \in Permanent\_Servers(x_b))$: $(CM_i[b,*] \geq CM^a[b,*])$)

$s_i \in Potential\_Servers(x_a)$ **then**
if $(CM_i[a,*] > CM^a[a,*])$ **then exit fi**;
% When a potential server s_i for an object x_a receives a value new_val_a that it
% knows is causally overwritten (this is revealed by the vector comparison)
% it does not consider it. The **exit** statement terminates the processing
% of the message. In the other case the processing continues.
delay the processing **until** ($C3$) **where**
% $C3$ expresses (similarly to $C2$) that all causally preceding updates on objects
% x_b, for which s_i is a permanent server, have been locally reported.
$C3$ **is** ($\forall x_b : (s_i \in Permanent\_Servers(x_b))$: $(CM_i[b,*] \geq CM^a[b,*])$)
end case;

% (S2) *Invalidation Part:*
% *ensures causality of updates on objects for which s_i is a potential server is not violated*
$\forall x_c$: $x_c \neq x_a$ **and** $(s_i \in Potential\_Servers(x_c))$:
do % $C4$ expresses that the current copy of x_c owned by s_i is obsolete or
% concurrent as revealed by the causal timestamp CM^a, so it is deleted.
$C4$ **is** $(present_i(x_c) = yes$ **and** $\neg(CM_i[c,*] \geq CM^a[c,*])$;
if $C4$ **then** $present_i\ (x_c) := no$ **fi**; /* Local copy of x_c is invalidated */
od;

% (S3) *Update Part (of the context of s_i)*
Install the new value new_value_a of x_a;
if $s_i \in Potential\_Servers(x_a)$ **then** $present_i(x_a) := yes$ **fi**;
$last\_writer_i(x_a) := j$;
$\forall (b,k) : CM_i[b,k] := max(CM_i[b,k], CM^a[b,k])$ **od**;
end

Figure 4: Processing of an *update* Message by Node s_i.

statement (S2) disappear; the resulting protocol becomes similar to the one used in the Isis system [5] for delivering causally ordered messages across overlapping groups (the set $Permanent\_Servers(x_a)$ constituting the group associated with x_a).

4.4 Creation and Deletion of Objects

The procedures CREATE and DELETE are used by potential servers of objects to add or remove a copy of an object. The simplest procedure is the deletion of a copy of an object x_a by one of its potential servers s_i:

> **procedure** DELETE(a);
> **begin** $present_i(x_a) := no$;
> deallocate the local copy of x_a;
> inform $attached\_to_i(x_a)$ that the copy has been deleted
> **end**

The last line allows the node $attached\_to_i(x_a)$ to update its data structures so it no longer disseminates updates of x_a to s_i when a push policy is used.

When a node $s_i \in Potential\_Servers(x_a)$ wants to create a new copy[7], it uses the following procedure.

> **procedure** CREATE(a);
> **begin** **let** s_j **be** $attached\_to_i(x_a)$;
> **send** $request(a)$ **to** s_j;
> **wait** $update(lastwriter_a, a, new\_val_a, CM^a)$ and
> execute statements (S1), (S2) and (S3) (described in Section 4.3)
> **end**

The permanent server s_j that receives a request for a copy of object x_a sends back the following message: $update(last\_writer_j(x_a), a, current\ value\ of\ x_a, CM_j)$. (See the discussion on liveness of the protocol in Section 4.6).

4.5 Accessing Objects

We did not explicitly describe how read and write accesses are completed. A server that has a copy of an object can complete these accesses locally. If the copy is updated, update messages are sent as described. When a potential server does not have a copy of an object, it makes use of the CREATE procedure to obtain a copy of the object and then the access is completed. Although a node that is neither a permanent nor a potential server for any object is not included in the causality tracking matrix, it keeps the value of the matrix that was received in the last communication it had with a server. This value is sent with a request and a server completes the access only after ensuring that the copy being accessed is causally consistent according to the information known to the client.

[7] When s_i does not have a copy of x_a, this creation can be initiated by itself, by a client that requests it to remotely access x_a, or by the underlying load balancing mechanism. When s_i has a copy of x_a, this creation can be due to s_i itself when it wants to get a more current value of x_a.

4.6 Correctness

A complete and formal proof of correctness for the protocol is beyond the scope of this paper due to size limitation. We provide informal arguments that show how a correctness proof can be constructed. Any such proof must address two aspects of the protocol: safety and liveness. Safety ensures that causally overwritten values are never read in a system following the protocol. The liveness part must demonstrate that requests will complete and they will access more recent values of objects as the system execution proceeds.

The fact that our protocol satisfies the safety requirement follows from the use of causality matrices. Each time a new value of an object x_a is added at node s_i, it could bring new causality information which can make the copies existing at s_i causally overwritten. This is avoided using the various consistency tests. If s_i is a permanent member for some objects (statement S1), the new value of x_a is not added until causally preceding updates to these objects have been reported (the control information about these updates is included in the matrix timestamp CM^a received with new_value_a and used in conditions $C1$, $C2$ and $C3$). If s_i is a potential member for another object x_c (statement S2) and this copy is found to be causally overwritten or concurrent (condition $C4$), it is deleted by s_i. The update of the causality matrix in statement (S3) ensures that a causally overwritten copy of the deleted object x_c will not be installed at s_i in the future and hence causally inconsistent values cannot be read.

The liveness part of the proof is more complex. One aspect of the proof must demonstrate that the various **delay** statements induced by the consistency tests do terminate. This follows from the fact that a delay at node s_i lasts until causally preceding updates arrive for objects for which s_i is a permanent server. Since all updated values are sent to permanent servers of an object and communication is reliable, these updates will arrive and be applied. (A more precise argument can be constructed similar to the argument that all messages are delivered by the CBCAST protocol of Isis [5].) Notice that updates to objects for which s_i is a potential server may not be received. However, copies of such objects are deleted (statement S2) when causally overwritten and hence they do not contribute to the delay experienced in processing a message. When a potential server s_i requests a copy of object x_a from a permanent server, the received copy may not reflect all updates to x_a known to s_i (from $CM_i[a,*]$). In this case, s_i will discard the received value (statement (S1)). If updates to x_a stop, the permanent server's copy will include all updates known to s_i and s_i will receive a copy that is installed by it. To ensure the same in the presence of ongoing updates, s_i should not process messages that can advance CM_i until it installs a copy of x_a.

Permanent servers will continue to receive newer values of objects produced by update accesses because such values are sent to these servers using a push mechanism (see Section 4.2). The same will be true for a potential server for which a push mechanism is used for update propagation. However, if a potential server employs a pull mechanism, it is possible that no updates are received by it and it continues to access copies existing at the node without communicating with other servers (this will happen only when the node is not a permanent server for any object). In this case, a periodic refresh message needs to be sent to such potential servers to ensure that they delete older copies of objects and install more recent ones.

5 Discussion

We have described the basic protocol that maintains consistency of a set of objects. To use this protocol in a real system, it needs to be enhanced by several optimizations and additional policies.

The primary source of perceived inefficiencies of the protocol is the matrix data structure. We first argue that in practical systems, its size will not be too large. We also describe several techniques that could be used to reduce its size. We then briefly outline the policies that need to be implemented to exploit the benefits made possible by the protocol. Finally, we address a variety of other issues that are relevant for a system that implements causally consistent distributed services.

5.1 Cost of Capturing Causality

In the protocol, a matrix is employed to capture causality information. The size of this matrix is $m \times n$ where m is number of related objects for which causal consistency has to be provided and n is the number of servers where instances of these objects can reside. Clearly, to reduce the size of the matrix, the number of related objects should be small. This will be the case when coarse-grained objects are shared and a service depends only on a small number of other services. For example, a document sharing service may depend only on a name service and a file service. Thus, with practical distributed services, m will not be too large. Also, to reduce the matrix size, several objects can be aggregated into one logical object. This, however, would disallow their independent replication. Other such tradeoffs also exist. For example, if objects are considered related unnecessarily, not only the matrix size increases but performance of the system will suffer because consistency may be provided across objects when it is not necessary. On the other hand, the matrix size can be reduced by not grouping objects even when some relationships exist between them. The consistency needed across such objects can be ensured at a higher level (*e.g.*, by programmers of these services).

The number of columns, n, in the matrix corresponds to servers where related object copies could be created. Actually, our protocol makes use of an entry in the matrix to only record the number of updates a server node has done to a given object that causally precede the current access. By limiting that only some of the servers be allowed to perform write accesses, it is possible to reduce the number of columns in the causality matrix. For example, it could be required that updates be executed only at permanent servers. In this case, potential servers can access the local copy of an object for read accesses but must submit updates to permanent servers. When writes are fewer compared to reads, benefits of replicated service instances at potential servers will be realized for most accesses in the system even when writes are performed at permanent servers.

It is possible to compress the matrix without impacting correctness of the protocol. For example, if writes are infrequent, few entries of the matrix will change between successive communications between nodes. In this case, only the changed entries, which can be stored in an update list, need to be sent. Other optimizations explored by Ruget in [25] can also be used to reduce the size of the matrix.

We feel that the cost of the matrix is fundamental to providing flexible and efficient implementations of replicated objects. Reducing the number of rows by grouping objects restricts the ability to replicate them independently. On the other hand, this overhead can be avoided if strong consistency is provided (e.g., a version number is sufficient in that case). However, the communication cost of strong consistency protocols will impact the efficiency of the services implemented using replicated objects.

5.2 Creation and Deletion of Service Instances

The protocol presents a set of mechanisms: how to create and delete service instances to improve performance, and how to disseminate updated values to potential servers using a pull or push mechanism. Policies that determine when these mechanisms should be used can have significant impact on the overall performance of the system. For example, one policy should monitor access patterns in the system. If it detects that there are lot of access requests and a service instance does not exist in a neighborhood, one should be created at a potential server. The creation of such a server instance could have significant overhead because copies of the objects that implement the service must be transmitted to the potential server. Thus, the object size and the expected number of invocations of the service must both be considered in deciding when to create a service instance at a potential server node. If the object state is large and most of it is read-only, such state can be saved locally across multiple creation and deletion operations to avoid the transmission of entire object state each time the service is created.

A similar policy is necessary to decide when a push or a pull mechanism needs to be used to disseminate new object values to potential servers that have copies of an object. Such a policy needs to monitor access patterns to detect if a potential server node frequently has to fetch new object copies. If this is the case, the permanent server from which the potential server creates its copy, should push new values it receives to the potential server. On the other hand, if a server accesses an object infrequently and several updates may be performed between consecutive accesses, it is desirable that not all updates be sent to the server. In this case, if latency of a pull operation can be tolerated, better performance can be achieved if the potential server pulls an object copy from a permanent server when it needs to access the object. Of course, a hybrid policy, which makes use of both pull and push mechanisms is what will be used in a system where different client neighborhoods have widely varying access patterns. Our protocol makes it possible to use such a hybrid policy.

5.3 Failures

Replicated services can improve service access performance and can provide high availability for the service. In this paper, we have focused on the performance of accessing a service and thus creation and deletion of service instances is driven by access patterns. When creation and deletion of service instances is due to failures, additional mechanisms are necessary. In particular, we have not addressed the dynamic creation and deletion of permanent servers which is necessary for dealing with failures (e.g., when the number of permanent servers falls below a threshold, it may be necessary to create a new one).

6 Comparison with Related Work

We have presented a protocol for maintaining causal consistency among replicated copies of objects that implement a set of related services. As mentioned earlier, several papers have presented implementations of causal consistency. In this section, we compare our protocol with these implementations.

The Isis system [5] (and others that include Horus [29], Transis [6], Totem [20], Psynch [22])[8] provides causally ordered group communication which can be used to maintain causal consistency among replicated objects. For example, a *multicast group* can be associated with nodes that have copies of an object and updates to this object can be propagated by using a primitive like CBCAST [5]. To maintain consistency across multiple related objects, causal ordering has to be guaranteed across messages that are sent to different groups. The control information required by the Isis CBCAST implementation in this case is the same as the matrix used by our protocol. This is due to the fact that vector time-stamps for each such group will be included in messages and stored at member nodes.

[8] It is important to note that these systems also put strong emphasis on *reliability* and hence address failures. Since we do not address fault-tolerance in this paper, the comparison with these systems is not complete.

The protocol we developed exploits the fact that object values can be overwritten and hence it does not require that messages containing values of all updates to an object be received by the object's potential servers. A more recent value of an object can be installed at such a server without waiting for an older value to arrive. If ordered message protocols are used to provide causal consistency, such information is not exploited by the protocol and it will not deliver a message with newer value before delivering all causally preceding messages. Thus, our protocol allows a potential server to control the update propagation rate based on the communication resources available and the timeliness requirements for propagating new object values.

The creation and deletion of service instances at potential servers will result in *view change* when CBCAST is used. View changes are costly because strong consistency (called View Synchronous Communication, VSC in short) is required for the control object that corresponds to group membership. In our protocol, we can separate the cases when view changes happen due to creation and deletion of service instances at potential servers, which are driven by performance concerns, from view changes that are triggered by failures. In the former case, which is addressed in this paper, we can make the view changes *light-weight* operations (they require communication with just one permanent server). This is possible because our protocol makes use of the read/write semantics of objects and employs both pull and push styles of communication to ensure causal consistency. Thus, potential servers can miss some updates when view change is not synchronous but still can pull a consistent value of the object when they need to access it. Such flexibility is not easily provided when group communication is used to ensure causal consistency in a dynamic environment. Although we do not use a view change protocol when potential servers create and delete service instances, such a protocol will be necessary in the less frequent cases when permanent servers have to be added or deleted or to handle failures of permanent servers.

In the lazy replication protocol [16], causal consistency is provided among replicated copies along with operations that provide stronger consistency guarantees. There are two major issues that this protocol does not handle which are addressed by our protocol. First, consistency across multiple related objects is not considered and hence vector time-stamps are sufficient in the lazy replication protocol (consistency is provided for a single object or multiple objects in a fully replicated system as the one described in Section 2.3). Also, the set of service instances is static whereas we allow dynamic behavior using potential servers. Finally, the use of both pull and push mechanisms ensures that all updates need not be propagated to all nodes having copies of the updated object (nodes that do not receive updates can pull them on demand).

Other related works that explore implementations of causal consistency are described in [23, 2, 15]. They do not explore creation and deletion of server instances and most protocols are either pull or push based. Thus, they include a particular policy for propagating updates whereas our protocol can exploit a policy that makes use of both according to service access patterns.

The work in [3] also divides nodes into categories that bear some resemblance to permanent and potential servers, and clients. For example, the members of the *core group* can be viewed as permanent servers. However, the emphasis of the paper is on developing a communication infrastructure for information dissemination in large scale systems. Thus, protocols that implement consistency for shared objects are not explored. However, their results can be applied in our work. In particular, permanent servers can be created and deleted using the view management techniques developed for core members.

Weaker consistency levels have been also explored by other systems that include Ficus [11], Bayou [28] and others. Dynamic object copies due to caching are also permitted in systems such as Coda [14] and Rover [13]. Although they are relevant to the work described in this paper, the problem explored by us (*i.e.*, implementation of causal consistency for several related objects) is different from what is explored in these systems.

7 Conclusion

Distributed services that are accessed by widely distributed clients are becoming common place (*e.g.*, specially services targeted to the home). Such services cannot be provided at the needed level of performance and availability without replicating the service at multiple nodes and without allowing a "relatively" weak definition of consistency.

This paper focused on developing implementations of distributed services when service instances can be created and deleted dynamically. We investigated causal consistency as the consistency criterion for such services and have shown that such a consistency criterion is useful for some distributed services (*e.g.*, cooperative document sharing); moreover an efficient protocol that implements it has been developed. Three types of nodes have been identified, namely, permanent server, potential server and client. The state of a service is encapsulated in an object which is replicated on a certain number of permanent servers. Potential servers of a service can also have a copy of its state but these copies are not permanent (they can be created and deleted dynamically according to performance or locality motivations). A client node can access a service but does not manage an instance of it. Our protocol makes replication of objects transparent to users as they perceive a causally consistent set of services. The proposed protocol is novel in that it allows the system to adapt to service request patterns by dynamically creating and deleting service instances at potential servers. It could also use both pull and push based mechanisms for propagating updates to objects.

We are now pursuing this work by addressing fault-tolerance issues. In long term execution of the system, it is also desirable to add new permanent servers and to delete existing ones. Such membership changes can also be induced by failures. The reliability of the permanent servers of a service, which constitute a group, can be addressed using concepts and techniques described in [6, 20, 22, 29]. Fault-tolerance problems related to potential servers of a service can be solved by using solutions inspired from [3]. We will explore the application of these techniques to our protocol in the future.

References

[1] Sarita V. Adve and Mark D. Hill. Weak ordering - a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[2] M. Ahamad, P.W. Hutto, G. Neiger, J.E. Burns, and P. Kohli. Causal Memory: Definitions, Implementations and Programming. *Distributed Computing*, 9:37-49, 1995.

[3] O. Babaoğlu and A. Schiper. On Group Communication in Large Scale Distributed Systems. *ACM Operating Systems Review*, 29(1):62-76, Jan. 1995.

[4] Sumeer Bhola, Sri Doddapaneni, and Bodhi Mukherjee. Villa: Event and state sharing mechanisms for building interactive collaborations. Submitted for Publication.

[5] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, 1991.

[6] D. Dolev and D. Malki. The Transis Approach to High Availability Cluster Communication. *Communications of the ACM*, 39(4):64-69, 1996.

[7] M. Dubois, C. Scheurich, and F. A. Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, June 1986.

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

[9] M.J. Fischer and A. Michael. Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network. In *Proc. ACM Symposium on Principles of Data Base Systems*, 1982, pp. 70-75.

[10] 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.

[11] R. G. Guy, J.S. Heidemann, W. Mak, T. W. Page, G. J. Popek, and D. Rothmeier. Implementation of the Ficus Replicated File System. In *USENIX Conference Proceeding*, June 1990.

[12] M. Herlihy and J. Wing. Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[13] 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. 15th ACM Symposium on Operating Systems Principles*, December 1995.

[14] J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *Proc. 13th ACM Symposium on Operating Systems Principles*, Pacific Grove CA, December 1995, pp. 213-225.

[15] R. Kordale and M. Ahamad. A Scalable Technique for Implementing Multiple Consistency Levels. *In Proc. of 16th IEEE International Conference on Distributed Computing Systems (ICDCS)*, May 1996.

[16] R. Ladin, B. Liskov, L. Shrira and S. Ghemawat. Providing Availability Using Lazy Replication. *ACM Transactions on Computer Systems*, 10(4):360–392, 1992.

[17] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[18] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C28(9):690–691, 1979.

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

[20] L. Moser, P.M. Melliar-Smith, D.A. Agarwal, R.K. Budhia and C.A. Lingley-Papadopoulos. Totem: a Fault-Tolerant Multicast Group Communication System. *Communications of the ACM*, 39(4):54-63, 1996.

[21] D.J. Parker *et al.* Detection of Mutual Inconsistency in Distributed Systems. *IEEE Transactions on Software Engineering*, SE9(3): 240–246, 1983.

[22] L.L. Peterson, N.C. Buchholz and R.D. Schilchting. Preserving and Using Context Information in Interprocess Communication. *ACM Transactions on Computer Systems*, 7(3):217–246, 1989.

[23] M. Raynal, A. Schiper and S. Toueg. The Causal Ordering Abstraction and a Simple Way to Implement it. *Information Processing Letters*, 39:343–350, 1991.

[24] M. Raynal and A. Schiper. From Causal Consistency to Sequential Consistency in Shared Memory Systems. *Proc. 15th Int. Conf. FST&TCS (Foundations of Software Technology and Theoretical Computer Science)*, Bangalore, India, Springer-Verlag LNCS 1026, (P.S. Thiagarajan Ed.), pp. 180-194, dec. 1995.

[25] F. Ruget. Cheaper Matrix Clock. In *Proc. 8th Int. Workshop on Distributed Algorithms*, Springer-Verlag LNCS 857 (G. Tel and P. Vityani Eds), pp. 355–369, 1994.

[26] Christian Schuckmann, Lutz Kirchner, Jan Schummer, and Jorg M. Haake. Designing object-oriented synchronous groupware with COAST. In *ACM CSCW'96*, 1996.

[27] Robert Strom, Guruduth Banavar, Kevan Miller, Atul Prakash, and Michael Ward. Concurrency control and view notification algorithms for collaborative replicated objects. In *Proceedings of the 17th ICDCS*, 1997.

[28] D. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. Welch. Session Guarantees for Weakly Consistent Replicated Data. In *Proc. of International Conference on Parallel and Distributed Data Systems*, September 1994, pp. 140-149.

[29] R. Van Renesse, K.P. Birman and S. Maffeis. Horus: a Flexible Group Communication System. *Communications of the ACM*, 39(4):76-83, 1996.

[30] G.T. Wuu and A.J. Berntsein. Efficient Solutions to the Replicated Log and Dictionary Problems. In *Proc. 3rd ACM Symposium on Principles of Distributed Computing*, Montreal, 1984, pp. 233-242.