An adaptive architecture for causally consistent distributed services

View the article online for updates and enhancements.

An adaptive architecture for causally consistent distributed services*

Mustaque Ahamad†, Michel Raynal‡ and Gérard Thia-Kime‡

† College of Computing, Georgia Tech, Atlanta, GA 30332, USA
‡ IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France

E-mail: `mustaq@cc.gatech.edu`, `raynal@irisa.fr` and `thiakime@irisa.fr`

Abstract. This paper explores causally consistent distributed services when multiple related services are replicated to meet performance and availability requirements. This consistency criterion is particularly well suited for distributed services such as cooperative document sharing, and it is attractive because of the efficient implementations that are allowed by it. A new protocol for implementing causally consistent services is presented. It allows service instances to be created and deleted dynamically according to service access patterns in the distributed system. It also handles the case where different but related services are replicated independently. Another novel aspect of this protocol lies in its ability to use both push and pull mechanisms for disseminating updates to objects that encapsulate service state.

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 required 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 neighbourhood 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 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 addressed 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 neighbourhood 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 service 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 [5] 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 [7]. 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.

This consistency criterion, *causal consistency* (CC), has two main advantages: (1) CC is meaningful for several

applications [1, 4], 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.

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 CC for a set of inter-related services when the services are independently replicated. In other words, related services may run on different sets of server nodes.

This paper is divided into six 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 compares the proposed approach with previous works. Finally, section 6 concludes the paper.

2. Causal consistency

CC, which is based on causality in distributed systems [6], has been explored in a number of contexts [1, 2, 10, 11]. These papers discuss several applications which execute correctly with the consistency guarantees provided by CC.

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 modelled as the sequence $op_i^1, op_i^2, \ldots, op_i^k \ldots$ where op_i^k denotes the kth operation executed by s_i (we simply use op_i to denote an operation when we do not need to know what process executed the operation). Such a sequence defines the local history $\hat{h}_i$ 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. $\hat{h}_i$ is the totally ordered set $(h_i, \rightarrow_i)$.

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

- $H = \bigcup_i h_i$.

- $op1 \rightarrow_H op2$ ($op1$ is causally ordered before $op2$) if:

  (i) $\exists\, s_i:\ op1 \rightarrow_i op2$ (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)

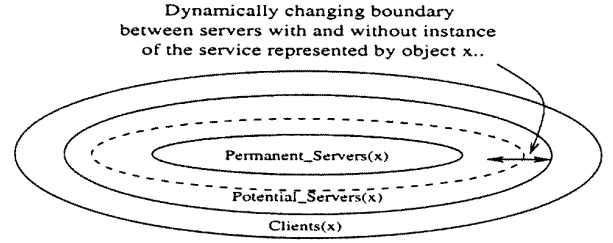 (iii) $\exists\, op3:\ op1 \rightarrow_H op3$ and $op3 \rightarrow_H op2$ (transitivity).

Figure 1. Classes of nodes for an object x.

Two operations $op1$ and $op2$ are *concurrent* in $\hat{H}$ 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. 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 a 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.

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 CC.

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 a 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)

We first consider the simple case where there is a single object that is replicated. Let this object be x. Three sets of nodes are associated with x (figure 1).

- *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 of x. In figure 1, this set is represented by the most interior oval.

- *Potential_Servers(x)* is a statically defined set of nodes $\{s_k, s_l, \dots\}$. 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 into two subsets (separated by a dotted oval in figure 1): 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 section 3.3 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: $s_j \in provider\_for_i(x) \Leftrightarrow attached\_to_j(x) = s_i$.

Finally, 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).
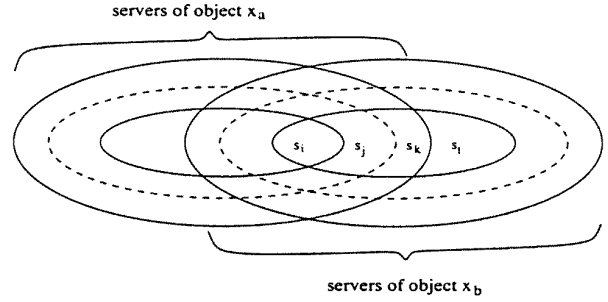

Figure 2. Servers for two objects.

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 2 where permanent and potential servers of two related objects x_a and x_b are represented (dotted ovals have the same meaning as in figure 1). We have:

- node s_i is a permanent server for both x_a and x_b;
- node s_j is a potential server (with a copy) for x_a and a permanent server for x_b;
- node s_k is a potential server (without a copy) for x_a and a 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† 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:

- *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 CC 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.

† 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 CC.

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 [2, 4, 10, 11]. Although a version vector is sufficient for tracking causality in the full replication case where related objects have copies at all server nodes, partial replication, which allows different servers to have copies of different subsets of related objects, requires that information about updates to each object 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 if 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 $O = \{x_1, x_2, \ldots, x_m\}$ be the set of related objects across which consistency needs to be provided and $S = \{s_1, s_2, \ldots, s_n\}$ 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† managed by a node s_i in S. Its meaning is the following one:

$CM_i[a, j]$ = number of updates to x_a issued by s_j

and known by s_i.

It is important to note that $CM_i[a, j]$ remains equal to zero 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, then the row $CM_i[a, *]$ represents the 'version' vector of the copy of x_a currently owned by s_i.

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

† 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.

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 one 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 one the entry $CM_i[a, i]$ and broadcasts the 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 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 are currently 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 bth 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):

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

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

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 $CM1$.
- $CM1[b, *] \ || \ CM2[b, *]$ 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.
- $\neg(CM2[b, *] \geqslant CM1[b, *])$ 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 CC for x_b. This is expressed by statement (S1) (figure 3).
- 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 copies if those copies can violate CC (due to the copy of x_a being updated with the new value contained in the message). This is expressed by statement (S2) (figure 3).

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 3. The **delay** statement delays the processing of the concerned message until 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 CC 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. 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.

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 CC 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 statement (S2) disappear; the resulting protocol becomes similar to the one used in the Isis system [2] for delivering causally ordered messages across overlapping groups (the set $Permanent\_Servers(x_a)$ constituting the group associated with x_a).

```
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 ∈ 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.
            C1 is ( (CM_i[a,j] + 1 = CM^a[a,j]) and (∀k ≠ j :  CM_i[a,k] ≥ 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 ( ∀x_b : (x_b ≠ x_a and s_i ∈ Permanent_Servers(x_b)):  (CM_i[b,*] ≥ CM^a[b,*]) )

      s_i ∈ 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 ( ∀x_b : (s_i ∈ Permanent_Servers(x_b)):   (CM_i[b,*] ≥ 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
   ∀x_c: x_c ≠ x_a and (s_i ∈ 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 ¬(CM_i[c,*] ≥ 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 ∈ Potential_Servers(x_a) then present_i(x_a) := yes fi;
         last_writer_i(x_a) := j;
         ∀(b,k) : CM_i[b,k] := max(CM_i[b,k], CM^a[b,k]) od;
end
```

Figure 3. Processing of an *update* message by node s_i.

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[†], 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
```

[†] 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, the create procedure can be executed when s_i wants to get a more recent value of x_a.

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).*

4.5. 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 neighbourhood, 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 a state can be saved locally across multiple creation and deletion operations to avoid the transmission of the 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 neighbourhoods have widely varying access patterns. Our protocol makes it possible to use such a hybrid policy.

5. Comparison with related work

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

The Isis system [2] (and others that include Horus [12], Transis [3], Totem [8], Psynch [9])† provides causally ordered group communication which can be used to maintain CC among replicated objects. To maintain consistency across multiple related objects, causal ordering has to be guaranteed

---

† 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.

---

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 timestamps for each such group will be included in messages and stored at member nodes. 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. 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) 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 CC. 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.

Other related works that explore implementations of CC are described in [1, 10]. 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.

6. Conclusion

This paper focused on developing implementations of distributed services when service instances can be created and deleted dynamically. We investigated CC as the consistency criterion for such services; 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 the 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 [3, 8, 9, 12].

Acknowledgments

References

[1] Ahamad M, Hutto P W, Neiger G, Burns J E and Kohli P 1995 Causal memory: definitions, implementations and programming *Distrib. Comput.* **9** 37–49

[2] Birman K, Schiper A and Stephenson P 1991 Lightweight causal and atomic group multicast *ACM Trans. Comput. Syst.* **9** 272–314

[3] Dolev D and Malki D 1996 The Transis approach to high availability cluster communication *Commun. ACM* **39** 64–9

[4] Fischer M J and Michael A 1982 Sacrificing serializability to attain high availability of data in an unreliable network *Proc. ACM Symp. on Principles of Data Base Systems* pp 70–5

[5] Herlihy M and Wing J 1990 Linearizability: a correctness condition for concurrent objects *ACM Trans. Program. Lang. Syst.* **12** 463–92

[6] Lamport L 1978 Time, clocks and the ordering of events in a distributed system *Commun. ACM* **21** 558–65

[7] Lamport L 1979 How to make a multiprocessor computer that correctly executes multiprocess programs *IEEE Trans. Comput.* C **C28** 690–91

[8] Moser L, Melliar-Smith P M, Agarwal D A, Budhia R K and Lingley-Papadopoulos C A 1996 Totem: a fault-tolerant multicast group communication system *Commun. ACM* **39** 54–63

[9] Peterson L L, Buchholz N C and Schlichting R D 1989 Preserving and using context information in interprocess communication *ACM Trans. Comput. Syst.* **7** 217–46

[10] Raynal M, Schiper A and Toueg S 1991 The causal ordering abstraction and a simple way to implement it *Informat. Process. Lett.* **39** 343–50

[11] Raynal M and Ahamad M 1998 Exploiting write semantics in implementing partially replicated causal objects *Proc. 6th Euromicro Workshop on Parallel and Distributed Processing (January 1998, Madrid, Spain)* pp 157–63

[12] Van Renesse R, Birman K P and Maffeis S 1996 Horus: a flexible group communication system *Commun. ACM* **39** 76–83