

Automatic Service Availability Management in Asynchronous Distributed Systems

Flaviu Cristian and Shivakant Mishra*
Department of Computer Science & Engineering
University of California, San Diego
La Jolla, CA 92093-0114, USA

Abstract

An Availability Management service is responsible for automatically ensuring that all critical services of a distributed system remain continuously available to users despite node removals and restarts caused by failures, maintenance and growth. We present an Availability Management service for an asynchronous distributed system characterized by unbounded communication delays and by the availability at all nodes of local, nonsynchronized timers that measure the passage of real time with some known accuracy. Examples of such systems are Unix, VMS, VM or MVS based distributed systems connected by local area networks such as Ethernet, token ring, FDDI, or channel-to-channel adapters. The presentation stresses the main ideas behind this new service, and outlines a simple design that depends upon the existence of asynchronous membership and atomic broadcast group communication services.

1 Introduction

With the ever increasing dependence that society and individuals place on computing services, their availability in the presence of failures, scheduled maintenance and horizontal growth becomes of paramount importance. In present systems, responsibility for reconfiguring a system after failures or removal of processors for maintenance rests mostly with the human system operators. Under stress, it is not unlikely that humans make mistakes when attempting repair actions, and this can lead to further failures and service

unavailability. For example, [11] reports that 42 % of the failures in the Tandem distributed systems are caused by human mistakes made during maintenance, operation and configuration.

It is thus interesting to replace the human controlled reconfiguration service by a software implemented Availability Management service that automatically reconfigures a system in the presence of failures and restarts as soon as these events are detected. The purpose of this paper is to explain the main ideas behind an Availability Management service for an asynchronous distributed system. Almost all current distributed systems are asynchronous, thus our design is of wide practical relevance.

To keep things simple and understandable, we omit the description of some important details. For example, we assume that server failures are caused only by node failures and that all services are provided by primary/backup server pairs instead of more general server groups. After presenting a simple design first, we discuss how our simplifying assumptions can be removed in a more realistic implementation that would take into account the possibility that server failures are caused by internal design faults and that not all services that must be made highly available need to be implemented by primary/backup server pairs.

We begin by introducing our basic notions, system structure and assumptions and by stating the requirements that a Service Availability Management service should satisfy. A replicated implementation is then briefly described. A discussion of possible extensions concludes the paper.

2 Basic Concepts

Every computing system is built to provide services to its users. A *service* provided to a user is a system behavior as perceived by that user: a sequence of

*This work was supported by grants from the Powell Foundation Sun Microsystems, the Microelectronics Innovation and Computer Research Opportunities in California, and the Air Force Office of Scientific Research. This paper was published in Proc. 2d Int. Workshop on Configurable Distributed Systems, Pittsburgh, March 1994.

observable outputs triggered by a sequence of service operation invocations made over time. The service specification prescribes future service outputs solely in terms of the operations being currently invoked and the current service state. At any point in time, the current *service state*, defined as being the result of applying the operation invocations completed so far to the initial service state, can be viewed as a summary of the past service behavior. Services with the same set of operations and potential behaviors belong to the same service *type*. In general, each service of a certain type has a current state distinct from that of other services of the same type, depending on the history of operation invocations completed so far. For example, the relational ANSI SQL query and update operations together with the semantics defined for them in an ANSI SQL reference manual define the ANSI SQL relational database service type; if “employees” and “accounting” are two database services of this type, their states at a certain point in time are in general different, depending on the history of updates applied since their creation. Thus, in general, different services of a certain type will have different current states.

The operations defined for a service can only be carried out by a *service implementation* consisting of one or more servers (or objects). A *server* encapsulates private state data by a set of procedures (or methods) that provide the only way for changing and accessing the server’s state. A server is a unit of failure and growth: at any point in time a service implementation has a membership consisting of an integer number of servers. Because servers are defined to be units of failure and growth, a server cannot span several nodes of a distributed system that could fail independently. Service operation invocations result in server procedure executions which can cause the state of the servers implementing the service to change. Since the state of a service is a function of the states of the servers that implement it, such server state changes lead in turn to service state changes.

Object-oriented programming methodology requires that service users not know any details about how the state of a service is represented in terms of server states or how the service operations are implemented by the server procedures. Such implementation details are hidden from users, who need only know the externally visible, abstract service specification [17]. For example, a database service can be implemented by a single database server, by a set of distributed servers that each manages a fragment of the database state, or by a group of redundant distributed database servers that each manages a replica

of the entire database.

If *redundant* servers running on distinct nodes are used to implement a service that must stay available despite node failures, the user need not know what synchronization and replication policies exist among servers. The *synchronization policy* prescribes how far apart the local server states can get, where the distance between the local states of two servers consists of the difference in the number of updates to the initial state applied so far by them. If the policy is *loose synchronization*, a *primary* maintains the current service state while one or more *backups* maintain past service states. A bound on the distance between loosely synchronized servers can be maintained by periodic checkpoints of the state of the primary to backups. If the policy is *close synchronization*, then the servers act as *peers* by interpreting all service requests in parallel and maintaining their internal states close to each other. The *replication policy* for a service s specifies how many servers should exist for s ¹. For example a replication policy of 2 specifies that 2 redundant servers should be used to implement s . The synchronization and replication policies specified for a service form the *availability policy* for that service. For example the availability policy (loose,2) for a service s prescribes that s should be implemented by a primary/backup server pair.

3 System Model

We assume a distributed system in which multiple nodes are connected by a communication network. These nodes are units of failure and growth; at any point in time a node is perceived as either correctly running or failed by another node. There is no shared memory or common physical clock, but we do assume that each node has access to a local timer to measure the duration of real-time. Servers run on nodes of the system. If a node possesses all physical resources needed for running a service for a certain service, it is called a (potential) *host* for that service. For example a node with enough computing power and memory that can access the disk(s) storing the “employees” database is a potential host for the “employees” database service. The set of all servers that can run in the hosts defined for a service s forms the *team* of servers that can be used to implement s . We assume that the names of distinct nodes are different and totally ordered, and that the nodes have amnesia-crash

¹for simplicity, in this paper we only consider static replication policies, where the number of replicas is constant.

failure semantics: after a crash, a node restarts in a predefined initial state independent of the inputs seen before the crash [8]. We don't assume any particular network topology: it can be point-to-point or broadcast channel based.

Further we assume an *asynchronous communication network*: there is no bound on the message transmission delays between nodes. In such networks, it is impossible for a node p to distinguish between a failure of the communication service between p and q and a failure of q . In such asynchronous environments, it is common to limit communication between team members only to those members that have joined a common group (or view, as it was called in [1]), where two processes p and q will be in the same group if they could communicate in a timely manner when the group was created. Processes that belong to different groups do not communicate with each other and are said to be (logically) *partitioned*. The existence of concurrent groups that do not communicate with each other can lead to divergence of the state of the team members belonging to such groups, if group activity is not restricted to a single distinguished group to be called in what follows the *active* group. To ensure uniqueness, it is common to define an active group as being a group whose membership contains a majority of team members and that was joined after being formed by all its members. The distributed communication service responsible for managing server groups is called a (asynchronous) *membership* service. The service that is used to disseminate updates to group members is called an (asynchronous) *atomic broadcast* service.

An asynchronous membership service enables any active member of a team \mathcal{T} of processes to join and leave dynamically formed groups, so that all members of a group agree on the group membership. Changes in group membership may occur due to node failures and recoveries, and communication service failures. We assume an asynchronous membership service that organizes the groups of active team members into a linear history so that members in successive active groups agree on the history of past active groups. Unlike in a synchronous membership service [7], there is no bound on the time it takes to form a new group after failures or recoveries are detected. Protocols for implementing asynchronous membership services satisfying different properties for point-to-point and broadcast channel based networks are given in [14, 16, 18].

An asynchronous atomic broadcast service enables any member p of a group to broadcast an update u to all members of that group in such a way that u is de-

livered to all active and correct members in the same order. This service ensures the following property for any two team members p and q that belong to the active group: let $\text{hist}(p)$ and $\text{hist}(q)$ be the histories of updates delivered to p and q by the atomic broadcast service since group creation, then either $\text{hist}(p)$ is a prefix of $\text{hist}(q)$ or $\text{hist}(p)=\text{hist}(q)$ or $\text{hist}(q)$ is a prefix of $\text{hist}(p)$ [6]. Notice that unlike a synchronous atomic broadcast service [10], there is no bound on the time it takes to deliver an update to the active team members by an asynchronous atomic broadcast service. Protocols for implementing asynchronous atomic broadcast for point-to-point and broadcast channel based networks are given in [4, 3, 6, 12, 13, 15, 2].

The implementation of the asynchronous Availability Management service *depends* [8] directly on the asynchronous atomic broadcast service, which depends on an asynchronous membership service, which in turn depends on an asynchronous datagram communication service. The dependency relations among these services is shown in Figure 1.

In addition to the dependency on the atomic broadcast and membership services, the Availability Management service has a close and natural association with a location transparent, highly available *request/reply transport* service. The latter service ensures that application server migrations are transparent to clients despite communication component failures. As mentioned in [9], this service enables a client of a service s to invoke an operation o by simply passing $s.o$ to the transport service available on the client's node without knowing where a server for s resides. The transport service forwards the request $s.o$ to an s -server and then routes the reply back to the client. This service also masks the failures of the servers of s from the clients as long as at least one server for s remains active. The implementation of this service also depends on the membership and atomic broadcast services (see Figure 1).

4 Informal Description

The goal of the Availability Management service is to enforce automatically the availability policies specified for the services offered by a distributed system without violating any constraints implied by their synchronization policies. For example, if the availability policy for a service s is (close-synchronization, 3), the availability management service is required to maintain three servers in the active processor group, and to start a server at a new processor p in the active group if any of the three servers fail. Since the active members

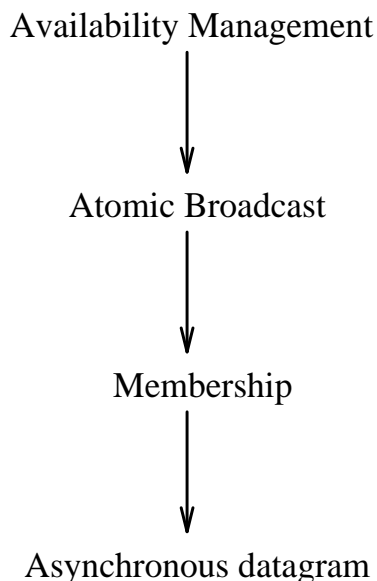


Figure 1: Dependencies among Fault-tolerant Services

cannot distinguish between failures of nonactive members and these being just partitioned, to maintain the availability policy even in the presence of partitions, there are two possible solutions. The first, applicable especially to services with persistent state, assumes that one cannot allow the local states of servers to diverge because of communication partitions. In this case, servers can exist only in one partition (the active one). This can be implemented by requiring that minority members stop any servers running locally. In the second case, there is no harm having servers active in different partitions. For these case, one can envisage enforcing in each partition the service availability policy specified for the service and reducing the number of servers when partitions merge. For the purpose of this paper, we consider only the first case: we require any servers running in minority partitions to be shut down. The underlying request/reply transport service will automatically re-route client requests in the active group to any new server when some failed server does not reply within a certain timeout period, so clients do not see server failures caused by shutdowns. However, the service can be unavailable to clients residing in a non active group.

In general, a distributed system provides many different services with different availability policies, and the availability management service is required to enforce the availability policies of each of these services. For simplicity, we consider that for all services in \mathcal{S} a single availability policy (loose-synchronization, 2) is specified. We further assume that a server for a ser-

vice $s \in \mathcal{S}$ can crash only when the underlying node crashes. From our presentation, it should not be difficult to imagine how the availability management service will automatically enforce other availability policies and how this service will deal with server crashes even when the underlying nodes do not crash.

The state transitions for various operations provided by the Availability Management service depend on whether availability management servers are in an active or non active group. Let $\mathcal{H}(s)$ denote the set of all hosts that can provide a service $s \in \mathcal{S}$. In general, a group with membership \mathcal{M} may be active for one service s , and not active for another service s' , depending on the intersections of \mathcal{M} with $\mathcal{H}(s)$ and $\mathcal{H}(s')$. For simplicity, we only investigate the case when the active group is the only one containing active services. Based on our description, it should be straight-forward to construct an availability manager for the more general model described above.

Let \mathcal{A} denote the set of all nodes in the system, and \mathcal{Maj} denote the set of nodes in the active (majority) group. The Availability Management service provides two kinds of operations to two (or possibly three) concurrent “users”: the human administrator and the Adverse Environment (and possibly time) [5]. The human administrator can invoke operations **start-service**(s), **stop-service**(s), **add-hosts**(s, h), and **remove-hosts**(s, h), $s \in \mathcal{S}$, $h \subset \mathcal{A}$. The Adverse Environment can invoke the **remove-nodes**(n) operation, $n \subset \mathcal{A}$. Finally, the operation **add-nodes**(n), $n \subset \mathcal{A}$, can be invoked by the passage of time if the nodes reboot automatically after a crash, or by human operator otherwise.

The Availability Management service is required to maintain a primary server and a backup server for each service s on distinct nodes in $\mathcal{H}(s) \cap \mathcal{Maj}$ for as long as there exist at least two nodes in $\mathcal{H}(s) \cap \mathcal{Maj}$. If $\mathcal{H}(s) \cap \mathcal{Maj}$ contains only one node, the availability manager must maintain a primary server for s on that node. In addition, if a primary server for service s crashes, the availability manager must first promote the backup server of s to primary and then start a new backup server for s at a different node in $\mathcal{H}(s) \cap \mathcal{Maj}$. The correctness of the (loose-synchronization, 2) policy requires that at any time there be no more than one primary server providing the same service to clients. To ensure this, the availability management service is required to stop all primary and backup servers in non-active (minority) groups before new primary and backup servers can be started in the active group. These requirements must be maintained by the availability management service even when an active group

ceases to be active (in response to **remove-nodes** operations), or when a non active group becomes active (in response to **add-nodes** operations). We are not concerned here with the local state checkpointing protocols followed by a primary and a backup server to maintain a bound on the distance between their local states or after a new backup is started. Our view is that primary/backup checkpointing is an application specific issue that is orthogonal to the system wide service Availability Management issue.

The state of the Availability Management service in a group, that was joined by the set \mathcal{M} of all its members, is recorded by the following constants and variables:

```

const  $\mathcal{A}$ : Set; % the set of all nodes in the system
const  $\mathcal{S}$ : Set; % the set of all services
var  $\mathcal{M}$ : Set-of-A init {}; % set of group members
var  $\mathcal{H}$ :  $\mathcal{S} \rightarrow$  Set-of-A init  $\lambda.\{\}$ ; % hosts for various services
var on:  $\mathcal{S} \rightarrow$  Boolean init  $\lambda.\text{false}$ ; % on( $s$ )= $\text{true}$ , when  $s \in \mathcal{S}$  is started
var primary:  $\mathcal{S} \rightarrow \mathcal{A} \cup \{\perp\}$  init  $\lambda.\perp$ ; % points to the group node hosting primary for  $s$ 
var backup:  $\mathcal{S} \rightarrow \mathcal{A} \cup \{\perp\}$  init  $\lambda.\perp$ ; % points to the group node hosting backup for  $s$ 
var active: Boolean init  $\text{false}$ ; % true, if  $\mathcal{M}$  is the active (majority) group

```

When there are no primary or backup servers for $s \in \mathcal{S}$ in \mathcal{M} , the *primary*(s) and *backup*(s) pointers have the undefined value \perp ($\perp \notin \mathcal{A}$). For each service s and non empty set of hosts \mathcal{P} , we assume the existence of a “**select-host**” function that returns the best host in \mathcal{P} for starting a new server for s . This function may take into account factors such as load balancing, communication pattern among different servers, and the interaction among different services in choosing the best possible host for service s . An example of **select-host**(s, \mathcal{N}) is “*return(smallest_id(N))*”. We also assume the existence of an “**is-active**” function that returns true if the group consisting of the nodes in \mathcal{N} is active and false otherwise. In a general system, this function may additionally take service s as an argument and return true if the group consisting of the nodes in \mathcal{N} is active for service s , and false otherwise.

```

select-host( $s:\mathcal{S}, \mathcal{P}:\text{Set-of-A}$ ) returns  $\mathcal{A} \cup \{\perp\}$ .
is-active( $\mathcal{N}:\text{Set-of-A}$ ) returns Boolean.

```

Since the operations invoked by the administrator, i.e. **start-service**, **stop-service**, **add-hosts**, and **remove-hosts**, do not modify the membership of a group, these operations have null effect if the group is not active. In response to a **start-service**(s) oper-

ation invocation, if the group is active, the availability management service remembers that s has been started and then starts primary and backup servers for s .

```

start-service( $s:\mathcal{S}$ )  $\equiv$ 
  if active = true
  then on( $s$ )  $\leftarrow$  true;
       start-servers( $s, \mathcal{M} \cap \mathcal{H}(s)$ );
  fi;

```

When a **stop-service**(s) request is received from the system administrator and if the group is active, the primary and backup servers for s are shut down and the service s is turned off.

```

stop-service( $s:\mathcal{S}$ )  $\equiv$ 
  if active = true
  then on( $s$ )  $\leftarrow$  false;
       if backup( $s$ )  $\neq \perp$ 
       then stop server for  $s$  on backup( $s$ )
          backup( $s$ )  $\leftarrow \perp$ 
       fi;
       if primary( $s$ )  $\neq \perp$ 
       then stop server for  $s$  on primary( $s$ )
          primary( $s$ )  $\leftarrow \perp$ 
       fi;
  fi;

```

When new hosts for some service s are added to a group that is active, the availability management service includes these nodes in $\mathcal{H}(s)$ and if s is ‘on’, it attempts to start primary and backup servers for s if there is no primary server for s , and attempts to start a backup server for s if there is no backup server for s .

```

add-hosts( $s:\mathcal{S}, h:\text{Set-of-A}$ )  $\equiv$ 
  if active = true
  then  $\mathcal{H}(s) \leftarrow \mathcal{H}(s) \cup h$ ;
       if on( $s$ )
       then start-servers( $s, \mathcal{M} \cap \mathcal{H}(s)$ );
       fi;
  fi;

```

When the administrator removes some hosts for some service s from a group that is active, the availability management service takes actions based on one of the following three cases: (i) both primary and backup servers are running on some removed hosts, (ii) only the backup server for s is running on some removed host, and (iii) only the primary server is running on some removed host. In case (i), the availability management service stops both the primary and backup

servers for s and then attempts to start primary and backup servers for s on the remaining hosts for s . In case (ii), the availability management service stops the backup server for s and then attempts to start a backup server for s on the remaining hosts for s . Finally, in case (iii), the availability management service stops the primary server for s , promotes the backup server for s to primary, and then attempts to start a backup server for s on the remaining hosts for s .

```

remove-hosts( $s:\mathcal{S}, h:\text{Set-of-A}$ )  $\equiv$ 
  if  $active = \text{true}$ 
  then  $\mathcal{H}(s) \leftarrow \mathcal{H}(s) - h$ ;
    if ( $backup(s) \in h$ ) and ( $primary(s) \in h$ )
    then stop server for  $s$  on  $backup(s)$ 
       $backup(s) \leftarrow \perp$ 
      stop server for  $s$  on  $primary(s)$ 
       $primary(s) \leftarrow \perp$ 
      start-servers( $s, \mathcal{H}(s) \cap \mathcal{M}$ );
    fi;
    if ( $backup(s) \in h$ ) and ( $primary(s) \notin h$ )
    then stop server for  $s$  on  $backup(s)$ 
       $backup(s) \leftarrow \perp$ 
      start-backup( $s, \mathcal{H}(s) \cap \mathcal{M} - primary(s)$ )
    fi;
    if ( $primary(s) \in h$ ) and ( $backup(s) \notin h$ )
    then stop server for  $s$  on  $primary(s)$ 
       $primary(s) \leftarrow \perp$ 
      promote-backup( $s, \mathcal{H}(s) \cap \mathcal{M}$ )
    fi;
  fi;
fi;

```

The operations invoked by adverse environment, i.e. **add-nodes** and **remove-nodes**, modify the membership of a group. When **add-nodes** is invoked, a new group is formed and the actions taken by the availability management service depend on one of the following three cases: (i) the new group is not active, (ii) the new group is active, and there exists a node n in the new group such that the last group to which n was joined is the preceding active group, and (iii) the new group is active, and there does not exist any node n in the new group such that the last group to which n was joined is the preceding active group. In case (i), this operation has null effect. In case (ii), the values of the variables \mathcal{H} , on , $primary$, and $backup$ in the state of the availability management service of the preceding active group are the most recent values, and so the availability management service first adopts these values, and then attempts to start servers for all services that are ‘on’. Finally, in case (iii), the values of the variables \mathcal{H} and on in the state of the availability management service of the preceding active group

are the most recent values; the values of the variables $primary$ and $backup$ however may have been modified in the non active groups because the availability management service stops all servers running on nodes in a non active group. So, in this case the availability management service adopts the values of the variables \mathcal{H} and on from the availability management service state of the previous active group, resets the variables $primary$, and $backup$ to their initial values, and then attempts to start servers for all services that are ‘on’. In all these cases, the variables \mathcal{M} and $active$ are appropriately modified.

```

add-nodes( $n:\text{Set-of-A}$ )  $\equiv$ 
   $\mathcal{M} \leftarrow \mathcal{M} \cup n$ ;
   $active \leftarrow \text{is-active}(\mathcal{M})$ ;
  if  $active = \text{true}$ 
  then if  $\exists n \in \mathcal{M}$  such that the last group to which
     $n$  was joined is the preceding active group
    then adopt  $\mathcal{H}$ ,  $on$ ,  $primary$ , and  $backup$  from
      the state of the preceding active group
    else adopt  $\mathcal{H}$  and  $on$  from
      the state of the preceding active group
       $primary \leftarrow \lambda.\perp$ 
       $backup \leftarrow \lambda.\perp$ 
    fi;
  for all  $s \in \mathcal{S}$ 
  do if  $on(s)$ 
    then start-servers( $s, \mathcal{M} \cap \mathcal{H}(s)$ )
    fi;
  od;
fi;

```

When **remove-nodes** is invoked, the actions taken by the availability management service depend on whether the new group (after removal of the nodes) is active. If the new group is not active, the availability management service stops all servers running on the nodes in the new group. If the new group is active, the actions taken by the availability management service, for each service s that has been started, depend on one of the following three cases: (i) both primary and backup servers for s were running on some removed nodes, (ii) only the backup server for s was running on some removed node, and (iii) only the primary server for s was running on some removed node. In case (i), the availability management service stops both the primary and backup servers for s and then attempts to start primary and backup servers for s on the remaining nodes. In case (ii), the availability management service stops the backup server for s and then attempts to start a backup server for s on the remaining nodes. Finally, in case (iii), the availability management service stops the primary server for

s , promotes the backup server for s to primary, and then attempts to start a backup server for s on the remaining nodes.

```

remove-nodes( $n$ :Set-of-A)  $\equiv$ 
   $\mathcal{M} \leftarrow \mathcal{M} - n$ ;
   $active \leftarrow \text{is-active}(\mathcal{M})$ ;
  if  $active = \text{false}$ 
  then for all  $s \in \mathcal{S}$ 
    do if  $backup(s) \neq \perp$ 
      then if  $backup(s) \in \mathcal{M}$ 
        then stop server for  $s$  on  $backup(s)$ ;
        fi;
         $backup(s) \leftarrow \perp$ ;
      fi;
      if  $primary(s) \neq \perp$ 
      then if  $primary(s) \in \mathcal{M}$ 
        then stop server for  $s$  on  $primary(s)$ ;
        fi;
         $primary(s) \leftarrow \perp$ ;
      fi;
    od;
  else for all  $s \in \mathcal{S}$ 
    do if ( $backup(s) \in n$ ) and ( $primary(s) \in n$ )
      then  $primary(s) \leftarrow \perp$ ;
       $backup(s) \leftarrow \perp$ ;
      start-primary( $s, \mathcal{H}(s) \cap \mathcal{M}$ );
      start-backup( $s, \mathcal{H}(s) \cap \mathcal{M} - primary(s)$ );
      fi;
    if ( $backup(s) \in n$ ) and ( $primary(s) \notin n$ )
      then  $backup(s) \leftarrow \perp$ ;
      start-backup( $s, \mathcal{H}(s) \cap \mathcal{M} - primary(s)$ );
      fi;
    if ( $primary(s) \in n$ ) and ( $backup(s) \notin n$ )
      then  $primary(s) \leftarrow \perp$ ;
      promote-backup( $s, \mathcal{H}(s) \cap \mathcal{M}$ );
      fi;
    od;
  fi;

```

The state transitions for **start-servers**, **start-primary**, **start-backup**, and **promote-backup** are defined as follows:

```

start-primary( $s$ : $\mathcal{S}$ ,  $\mathcal{N}$ :Set-of-A)  $\equiv$ 
  if  $primary(s) = \perp$  and  $\mathcal{N} \neq \{\}$ 
  then  $primary(s) \leftarrow \text{select-host}(s, \mathcal{N})$ ;
  start primary server for  $s$  on  $primary(s)$ 
  fi;
start-backup( $s$ : $\mathcal{S}$ ,  $\mathcal{N}$ :Set-of-A)  $\equiv$ 
  if  $backup(s) = \perp$  and  $\mathcal{N} \neq \{\}$ 
  then  $backup(s) \leftarrow \text{select-host}(s, \mathcal{N})$ ;
  start backup server for  $s$  on  $backup(s)$ 

```

```

  fi;
promote-backup( $s$ : $\mathcal{S}$ ,  $\mathcal{N}$ :Set-of-A)  $\equiv$ 
  if  $backup(s) \neq \perp$ 
  then promote backup server for  $s$  on  $backup(s)$ ;
   $primary(s) \leftarrow backup(s)$ ;
   $backup(s) \leftarrow \perp$ ;
  start-backup( $s, \mathcal{N} - \{primary(s)\}$ );
  fi;
start-servers( $s$ : $\mathcal{S}$ ,  $\mathcal{N}$ :Set-of-A)  $\equiv$ 
  if  $active = \text{true}$ 
  start-primary( $s, \mathcal{N}$ );
  if  $primary(s) \neq \perp$ 
  then start-backup( $s, \mathcal{N} - \{primary(s)\}$ );
  fi;
  fi;

```

These state transitions are required to be performed by the Availability Management service in every group as long as there is at least one active node running in the group. If concurrent failures and restarts occur, the corresponding state transitions are required to occur in some serial order. An availability manager satisfying the above requirements ensures continuous availability of a service to the clients in the active group. The service is unavailable to clients in a non active group, but it becomes available to them as soon as they join an active group following a membership change.

5 Replicated Availability Management Service

To ensure that the Availability Management service is itself available in every group in the system as long as there is at least one active node, the service state introduced in the previous section is maintained in every group by replicating it at every active node in the group. Thus, the team of Availability Management servers is hosted by the set \mathcal{A} of all nodes of the system, and the state of the Availability Management service in the system consists of one service state for each group in the system and various constants and variables described in the last section replicated at every node in a group. These replicated variables are managed at any point in time by a group of active closely synchronized Availability Management servers. These servers are initialized after the underlying node membership and atomic broadcast services are initialized at node startup. While servers for these services exist at any node, the Administrator Command Interpreter service is implemented by a server running in only one of the nodes (the administrator can login on another

node if that node fails, or if the group to which it belongs becomes non-active). When joining the active group of Availability Management servers, a server receives the membership of the group ($newN$), the `group_id` of the preceding active group (pag), a mapping from each group member to the `group_id` of the last group joined by it ($pg: \mathcal{A} \rightarrow \mathcal{N}$), and the state of the availability management service of the preceding active group (pag_state). Based on this, each server independently constructs the state of the Availability Management service. Each Availability Management server has access to the identity of its underlying node by invoking a predefined function $myid$.

The implementation of the Availability Management service $depends$ directly on the membership and atomic broadcast services described in Section 3: any update to a replicated state variable is either a result of atomic broadcast message arrival or of a membership change notification that appears to the replicated Availability Management servers as an atomic broadcast message arrival. Because all updates to the replicated state variables are received *in the same order* by all active Availability Management servers in the active group, after an Availability Management server j joins the active group, its local state variables \mathcal{M} , \mathcal{H} , $primary$, $backup$, on , $active$ will go through the *same sequence* of values as the local variables of any other Availability Management server m that is also a member of the same group, and this will remain true until j or m fail, or until the communication service between j and m fails. Thus, when any two members of the active group learn about the same event, such as an administrator command invocation or a change in the membership of active nodes, they have *identical local states*, so they reach *identical decisions* about what has to be done. For example after a failure of a node hosting the primary server for s , all Availability Management servers in the group containing that host decide that the management server running in the $backup(s)$ node, say p will have to promote the backup server to primary and the management server running in the node q returned by $select_host(s, \mathcal{M} \cap \mathcal{H}(s) - backup(s))$ will have to start a new backup for s . So, all availability management servers update their state according to these decisions and they all ask themselves whether they are the ones running in the p or q nodes by evaluating the $myid = backup(s)$ and $myid = select_host(s, \mathcal{M} \cap \mathcal{H}(s) - \{backup(s)\})$ expressions, respectively. The managers running in the nodes where these expressions evaluate to true then do the “real” work by locally promoting the backup server for s and by starting a backup server

for s , respectively. It is crucial that the value of an invocation of $select_host$ function, or is_active function depend only upon the replicated state variables maintained by each Availability Management server, so that any invocation yields the same result when invoked in response to the same event at any two members of the same group of the active Availability Management servers. In order to avoid inconsistencies introduced in the replicated state due to some operations being executed while there is a membership change, a boolean variable $freeze$ is used. This variable is set to true between the moment a node learns of some group creation and the moment the node joins the new group.

After initializing the state variables \mathcal{M} , \mathcal{H} , on , $primary$, $backup$, $freeze$, and $active$, an Availability Management server enters an infinite loop inside which it waits for the following event types: an up call from the membership service that is a notification of a change in the membership of active nodes, an up call from the atomic broadcast service telling about an update to the replicated state variables, or down call from the Command Interpreter running in the same node, that informs the server about a command issued by the system administrator. The code that implements the reactions to these events must be implemented so as to be atomic with respect to synchronization (for simplicity we do not deal with synchronization issues related to making the parallel interpretation of these events serializable).

task Availability-Manager \equiv

```

const  $\mathcal{A}, \mathcal{S}$ : Set;
var  $\mathcal{M}$ : Set-of-A init {};
var  $\mathcal{H}$ :  $\mathcal{S} \rightarrow$  Set-of-A: init  $\lambda.\{\}$ ;
var  $on$ :  $\mathcal{S} \rightarrow$  Boolean init  $\lambda.false$ ;
var  $primary$ :  $\mathcal{S} \rightarrow \mathcal{A} \cup \{\perp\}$  init  $\lambda.\perp$ ;
var  $backup$ :  $\mathcal{S} \rightarrow \mathcal{A} \cup \{\perp\}$  init  $\lambda.\perp$ ;
var  $freeze$ : Boolean init false;
var  $active$ : Boolean init false;

```

initialize($\mathcal{M}, \mathcal{H}, on, primary, backup, active, freeze$);

loop

when receive-from-administrator(command):

if $active = true$ and $freeze = false$

then case command of:

start-service(s):

atomically-broadcast (“*start-service*”, s);

stop-service(s):

atomically-broadcast (“*stop-service*”, s);

```

    add-hosts( $h, s$ ):
      atomically-broadcast (“add-hosts”,  $h, s$ );
    remove-hosts( $h, s$ ):
      atomically-broadcast (“remove-hosts”,  $h, s$ );
  endcase;
fi;

```

when receive-atomic-broadcast(message) from p :

```

case message of:
  (“start-service”,  $s$ ):
     $on(s) \leftarrow \text{true}$ ;
    start-servers( $s, \mathcal{M} \cap \mathcal{H}(s)$ );
  (“stop-service”,  $s$ ):
     $on(s) \leftarrow \text{false}$ ;
    if  $backup(s) \neq \perp$ 
    then if  $backup(s) = myid$ 
      then locally stop server for  $s$ 
    fi;
     $backup(s) \leftarrow \perp$ ;
  fi;
  if  $primary(s) \neq \perp$ 
  then if  $primary(s) = myid$ 
    then locally stop primary server for  $s$ 
  fi;
   $primary(s) \leftarrow \perp$ ;
  fi;
  (“add-hosts”,  $h, s$ ):
     $\mathcal{H}(s) \leftarrow \mathcal{H}(s) \cup h$ ;
    if  $on(s)$ 
    then start-servers( $s, \mathcal{M} \cap \mathcal{H}(s)$ );
  fi;
  (“remove-hosts”,  $h, s$ ):
     $\mathcal{H}(s) \leftarrow \mathcal{H}(s) - h$ ;
    if  $backup(s) \in h$ 
    then if  $backup(s) = myid$ 
      then locally stop backup server for  $s$ ;
    fi;
     $backup(s) \leftarrow \perp$ 
    start-backup( $s, \mathcal{H}(s) \cap \mathcal{M} - primary(s)$ )
  fi;
  if  $primary(s) \in h$ 
  then if  $primary(s) = myid$ 
    then locally stop primary server for  $s$ ;
  fi;
   $primary(s) \leftarrow \perp$ 
  promote-backup( $s, \mathcal{H}(s) \cap \mathcal{M}$ )
  fi;
endcase;

```

when receive-membership-notification(change):

```

case change of:
  (“freeze”):

```

```

     $freeze \leftarrow \text{true}$ ;
  (“new group”,  $newN, pag, pg, pag\_state$ ):
     $\mathcal{M} \leftarrow newN$ ;
     $active \leftarrow \text{is-active}(\mathcal{M})$ ;
     $freeze \leftarrow \text{false}$ ;
    if  $active = \text{false}$ 
    then for all  $s \in \mathcal{S}$ 
      do if  $backup(s) \neq \perp$ 
        then if  $backup(s) = myid$ 
          then stop backup server for  $s$ ;
        fi;
         $backup(s) \leftarrow \perp$ ;
      fi;
      if  $primary(s) \neq \perp$ 
      then if  $primary(s) = myid$ 
        then stop primary server for  $s$ ;
      fi;
       $primary(s) \leftarrow \perp$ ;
    fi;
    od;
  else  $on \leftarrow pag\_state.on$ ;
     $\mathcal{H} \leftarrow pag\_state.\mathcal{H}$ ;
    if  $\exists n \in \mathcal{M} : pg(n) = pag$ 
    then  $primary \leftarrow pag\_state.primary$ ;
       $backup \leftarrow pag\_state.backup$ ;
    else  $backup \leftarrow \lambda.\perp$ ;
       $primary \leftarrow \lambda.\perp$ ;
    fi;
    for all  $s \in \mathcal{S}$ 
    do if ( $backup(s) \notin \mathcal{M}$ ) and
      ( $primary(s) \notin \mathcal{M}$ ) and ( $on(s)$ )
      then  $primary(s) \leftarrow \perp$ ;
         $backup(s) \leftarrow \perp$ ;
        start-primary( $s, \mathcal{H}(s) \cap \mathcal{M}$ );
        start-backup( $s, \mathcal{H}(s) \cap \mathcal{M} - primary(s)$ );
      fi;
      if ( $backup(s) \notin \mathcal{M}$ ) and
        ( $primary(s) \in \mathcal{M}$ ) and ( $on(s)$ )
      then  $backup(s) \leftarrow \perp$ ;
        start-backup( $s, \mathcal{H}(s) \cap \mathcal{M} - primary(s)$ );
      fi;
      if ( $primary(s) \notin \mathcal{M}$ ) and
        ( $backup(s) \in \mathcal{M}$ ) and ( $on(s)$ )
      then  $primary(s) \leftarrow \perp$ ;
        promote-backup( $s, \mathcal{H}(s) \cap \mathcal{M}$ );
      fi;
    fi;
  od;
endcase;

```

endloop;

The procedures `start-primary`, `start-backup`, `promote-back`, and `start-servers` are as follows:

```

procedure start-primary( $s:\mathcal{S}, \mathcal{N}:\text{Set-of-A}$ )  $\equiv$ 
  if  $primary(s) = \perp$  and  $\mathcal{N} \neq \{\}$ 
  then  $primary(s) \leftarrow \text{select-host}(s, \mathcal{N})$ ;
    if  $primary(s) = myid$ 
    then locally start a primary server for  $s$ ;
    fi;
  fi;

```

```

procedure start-backup( $s:\mathcal{S}, \mathcal{N}:\text{Set-of-A}$ )  $\equiv$ 
  if  $backup(s) = \perp$  and  $\mathcal{N} \neq \{\}$ 
  then  $backup(s) \leftarrow \text{select-host}(s, \mathcal{N})$ ;
    if  $backup(s) = myid$ 
    then locally start a backup server for  $s$ ;
    fi;
  fi;

```

```

procedure promote-backup( $s:\mathcal{S}, \mathcal{N}:\text{Set-of-A}$ )  $\equiv$ 
  if  $backup(s) \neq \perp$ 
  then if  $backup(s) = myid$ 
    then locally promote backup server for  $s$ ;
    fi;
     $primary(s) \leftarrow backup(s)$ ;
     $backup(s) \leftarrow \perp$ ;
    start-backup( $s, \mathcal{N} - \{primary(s)\}$ );
  fi;

```

```

procedure start-servers( $s:\mathcal{S}, \mathcal{N}:\text{Set-of-A}$ );
  if  $active = true$ 
  start-primary( $s, \mathcal{N}$ );
  if  $primary(s) \neq \perp$ 
  then start-backup( $s, \mathcal{N} - \{primary(s)\}$ );
  fi;
fi;

```

6 Conclusions

In this paper, we have presented the key ideas in implementing an asynchronous Availability Management service that automatically reconfigures the system in the presence of communication and node failures, such that the computing services remain available, and the system reconfiguration is transparent to the users. The design, as sketched in the paper, considers only one type of availability policy, and restricts the availability of all services to only one group (active group) in the system. An extension of this design that would address the issue of enforcing more than one availability policy, and allow different active

groups for different services should pose no difficulty at this point. To enforce more than one availability policy, it is sufficient, for each service s , to keep track of its availability policy and ensure that it is automatically enforced along the lines of the previously sketched availability management service design. To allow different active groups for different services, it is sufficient, for the function `is_active`, to have an additional parameter (service s) and evaluate that function based on s .

References

- [1] A. Abbadi, D. Skeen, and F. Cristian. An efficient fault-tolerant protocol for replicated data management. In *SIGACT/SIGMOD*, 1985.
- [2] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, Aug 1991.
- [3] R. Carr. The Tandem global update protocol. *Tandem Systems Review*, Jun 1985.
- [4] J. Chang and N. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, Aug 1984.
- [5] F. Cristian. A rigorous approach to fault-tolerant programming. *IEEE Transactions on Software Engineering*, SE-11(1):23–31, Jan 1985.
- [6] F. Cristian. Asynchronous atomic broadcast. *IBM Technical Disclosure Bulletin*, 33(9):115–116, Feb 1991. Also presented at the First IEEE Workshop on Management of Replicated Data, Houston, TX, (Nov 1990).
- [7] F. Cristian. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing*, 4:175–187, 1991.
- [8] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of ACM*, 34(2):56–78, Feb 1991.
- [9] F. Cristian. Automatic reconfiguration in the presence of failures. *Software Engineering Journal*, pages 53–60, Mar 1993.
- [10] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. In *Proceedings of*

the Fifteenth International Symposium on Fault-Tolerant Computing, pages 200–206, Ann Arbor, MI, Jun 1985.

- [11] J. Gray. An approach to decentralized computer systems. *IEEE Transactions on Software Engineering*, SE-12(6):684–692, Jun 1986.
- [12] M. F. Kaashoek, A. Tanenbaum, S. F. Hummel, and H. Bal. An efficient reliable broadcast protocol. *Operating Systems Review*, 23(4):5–19, Oct 1989.
- [13] P. M. Melliar-Smith, L. E. Moser, and V. Agrawala. Broadcast protocols for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):17–25, Jan 1990.
- [14] S. Mishra, L. Peterson, and R. Schlichting. A membership protocol based on partial order. In *Proceedings of the Second Working Conference on Dependable Computing for Critical Applications*, pages 137–145, Tucson, AZ, Feb 1991.
- [15] S. Mishra, L. Peterson, and R. Schlichting. Consul: A communication substrate for fault-tolerant distributed programs. *Distributed Systems Engineering Journal*, 1993. *to appear*.
- [16] L. E. Moser, P. M. Melliar-Smith, and V. Agrawala. Membership algorithms for asynchronous distributed systems. In *Proceedings of the Eleventh International Conference on Distributed Computing Systems*, pages 480–488, Arlington, TX, May 1991.
- [17] D. L. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5), 1972.
- [18] A. Ricciardi and K. Birman. Using process groups to implement failure detection in asynchronous environments. In *Eleventh ACM Symposium on Principles of Distributed Computing*, Montreal, Canada, Aug 1991.