A Membership Protocol for Multi-Computer Clusters

Francesc D. Muñoz-Escoí   Vlada Matena   José M. Bernabéu-Aubán
Pablo Galdámez

Technical Report ITI-ITE-98/4

Abstract

Distributed applications need membership services to know which of their components are active or have failed. The Cluster Membership Monitor (CMM) provides membership services at the machine level. As its name suggests, the target environment for this membership service is a cluster of computing nodes, where the CMM checks the current state of all machines that have a CMM instance running on them. All monitors run a distributed membership protocol with an integrated timer-based failure detector. The aim of these monitors is to notify all membership changes to a set of software packages that previously requested their services. As a result, all these packages know about the current configuration of the cluster where they are running and may adapt its state to this configuration.

The protocol has been used as the basis for the development of a reliable communication layer and as an assistance service for the reconfiguration of an *object request broker* which provides the remote invocation support for the object-oriented applications running in the cluster.

1 Introduction

A membership service has to maintain the set of machines or processes that compose a group [9, 15]. When changes in the membership set arise, the service has to report these changes to the alive group members or to other system components which depend on the group being monitored.

Several types of groups exist in distributed systems. Group communication toolkits [1, 2, 4, 7, 10, 12, 14, 22] use the group concept to maintain a set of processes that are interested in the same sequence of incoming messages. Usually, each group of this kind can be seen as a replicated process. So, group communication toolkits provide a basis to develop highly available applications.

Other membership services maintain a set of machines as its members [5, 17, 19, 20]. The information provided by this kind of services can be used to build other services which depend on the set of machines that constitute the current distributed system.

Our membership service, the *cluster membership monitor* (CMM), also maintains the current set of machines that constitute the distributed system. In this case, the user of the membership services is a reliable transport layer. This transport relies on the information maintained by the membership service to guarantee that all messages that have been sent are either delivered at their target or an error notification is returned as soon as possible to the sender if the target node has crashed. Thus, the transport prevents a sender from waiting indefinitely for a response from a crashed node. Moreover, this transport is the lower layer of an *object request broker* (ORB) [16] which is also reported each time a membership change arises.

Since the protocol followed by the CMM is a distributed algorithm that needs a CMM replica in each of the system nodes, the CMM is also used to guide and synchronize the ORB software placed at each node when a reconfiguration has to be made. When some membership change is detected, the ORB must follow a series of steps to rebuild its state. The CMM controls that all the ORB's have completed a reconfiguration step before the next one is initiated.

The rest of the document is arranged as follows. Section 2 gives an overall description of the environment where the CMM is integrated and the properties that the protocol must provide. Section 3 explains the Cluster Membership Monitor protocol and its current implementation. Sections 4 and 5 give some performance values and some informal correctness arguments. Comparison with related work is given in Section 6. Finally, some conclusions about this work can be found in Section 7.

2 System Requirements

The *cluster membership monitor* provides membership services to an ORB which is used to intercommunicate the objects that run in a multi-computer cluster. The CMM has to maintain the set of nodes included in the cluster and must notify the ORB and other system components when some change happens in the membership set, either due to a new node joining the cluster or due to the failure or departure of a previously running one. To achieve these functions, the CMM requires that the following properties were satisfied by the system:

- *Non-reliable transport services.* The CMM uses this kind of services to send messages to other CMM modules placed in other nodes of the cluster. No additional attempt is made if a message cannot be delivered to its target node.
- *Halting failure assumption* [21]. Every node in the system works properly or crashes (stops) without causing any malfunction to the rest of the system.

  This is not so easy as it may seem. An isolated node[1] might answer its requests of service having a local state inconsistent with that of the rest of the system. Thus, an isolated node is providing an incorrect service to its clients. So, to prevent this situation, isolated nodes must be aborted as soon as possible.
- *Full connectivity.* Each node in the cluster is connected to any other node, either directly or via intermediate nodes. The underlying transport has to provide the image that any cluster node is directly accessible from any other node in the cluster.
- *Bound message transmission time.* An upper bound in the message transmission time[2] must be assumed. So, if an expected message is not received on time, its target CMM assumes that its sender has not been able to send it. If several attempts to receive a message from the same sender have failed, the target CMM assumes the sender has crashed.
- *Node addressing.* The CMM has to be integrated at a software level (inside the operating system kernel, if possible) where each node in the cluster must be distinguishable from other nodes. Each node has to use a different and unique identifier (e.g., its network address).
- *Invariable node and member identifiers.* Each node in the system has always the same node identifier which is used by the CMM protocol as an internal identifier when the node is alive and integrated in the membership set. So, a node does not change its identifier after being aborted or reinitiated.
- The latter property requires the *use of stable storage* to save a reconfiguration number associated to any node. Each time a node takes part in a cluster reconfiguration, it increases a *sequence number*. This number is required to distinguish obsolete messages sent by some member of the cluster.

Moreover the target distributed system is assumed asynchronous; i.e., all its nodes do not share a global clock and the message delivery time is not bound.

If the system provides an environment with the characteristics stated above, our protocol has to guarantee the following requirements:

R1 Each active node in the cluster must maintain the same membership set. When a change in the membership set is detected, all the active nodes must agree on this change and this agreement can not be deferred.

R2 Only preconfigured nodes can be members of the cluster. If a node was not registered as a possible member of the group, its requests to join the group are rejected.

R3 A node can be integrated in the membership set at any time. When it requests its integration, the joining node gets the current membership set and joins the group as soon as possible.

This can be made even if the membership set was being reconfigured due to the joining or failure of other nodes; i.e., multiple-nodes joins are accepted.

[1]The term *isolated node* refers to a cluster node that is not able to communicate with any other node of its cluster, although it can send or receive messages from external client nodes or domains.

[2]The term *transmission time* refers here to the time needed to carry a message from its sender node to its receiver node when no failure occurs. It is not the same as *message delivery time*, which considers the time needed to deliver a message to its destination, including all transmission attempts made in case of communication failures.

R4 A node is considered faulty when its messages are not received by the rest of the cluster nodes. A faulty node is removed from the membership set immediately.

When a member node does not receive any message from other cluster members, it is automatically aborted. This prevents interaction between a "faulty" node and external machines.

R5 Partitions are not allowed. When some members of the current cluster can not communicate with others, the greater subgroup becomes the new cluster and the nodes in other subgroups must be shutdown. Moreover, a minimum quorum is required to allow the remaining group to survive. If this quorum is not achieved, all the nodes are stopped.

These requirements are needed to serve appropriately other distributed servers that run in our multi-computer cluster. As stated above, two of these servers are the reliable transport layer and the ORB. The former needs a fast notification of membership changes, which can be satisfied if the failure detection mechanism is based on requirement R4. R4 is needed because the target system is asynchronous, and according to [3] in such a system the *accuracy* and *liveness* properties can not be simultaneously satisfied if the failure detection mechanism reports failures before the faulty component is restored. In our case, the CMM is not *accurate* —so it can notify failures that never arose— but is *live*, because all failures are notified by the membership service.

Requirement R2 is needed to guarantee that only the cluster nodes belong to the group. Similarly, requirement R5 is needed to prevent inconsistencies in the shared state of the distributed applications that run in the cluster (mainly the ORB).

3 Cluster Membership Monitor

The CMM protocol can be seen as a distributed algorithm that requires a monitor in each one of the machines that run the algorithm and are included in the membership set. Each one of these monitors runs the same algorithm, which is divided in three procedures according to the task being done by the membership service. These three procedures require different intercommunication approaches to deal with their purposes.

The first procedure corresponds to the search of the group of machines that are currently running and is known as the *agreement procedure*. All monitors enter this phase when a failure is suspected or a new node is trying to join the group. In the agreement procedure, all the monitors exchange information about the group of nodes that are accessible from their local nodes. Eventually, all nodes reach agreement on the set of nodes that are up and running. So, as a result of the agreement procedure, an stable membership set is found.

The second procedure is needed to notify other cluster-wide services about the new membership set found in the agreement phase. This is the *notification procedure* and can be divided in a sequence of steps. In each step, the membership monitor notifies the new membership set to a different system component. Once the local component has been notified, all monitors initiate a round of messages to agree on the termination of the current notification and, if other steps are required, the following one is initiated. At the end of the notification procedure, all system services that need to know the current membership set have received that information. The arrangement in steps guarantees that all nodes have done the notifications in the same order and at the same time.

The third procedure is also started when the agreement one is terminated. This is the *polling procedure*. In this procedure periodical messages are sent by the cluster nodes to check the state of the other cluster members. If a node does not receive some messages of this kind for a given sender, it suspects this sender is faulty and leads all cluster monitors to the *agreement procedure*.

This decomposition of the membership protocol in three procedures allows an efficient and robust implementation of the protocol, integrating failure detection in the resulting algorithm —several membership algorithms require external failure detectors [12, 13, 19, 18, 6], so the whole cost of their membership solutions depends on this external service. Also, the *notification procedure* of our membership service provides valuable help to other system services when a change in the membership set has happened. So, the approach followed in this protocol seems to be practical to satisfy all the membership requirements found in a multi-computer cluster.

The following subsections describe the data structures maintained by the membership monitors, the information being transferred in membership messages and the algorithms followed in each protocol procedure.

3.1 Data Structures

Each membership monitor runs an algorithm based on an automaton. Each automaton state represents a different protocol phase or a transitory step that modifies part of the data maintained by the monitor. Transitions between states occur when the tasks associated to the current state have been finished or some external event has been detected; for instance, a message from a joining member, or a timeout pointing out the failure of another member.

The completion of the tasks needed in each automaton state always requires an agreement among all the current members on their successful termination. To control the achievement of the agreement, a state matrix is used. This matrix maintains the current states of all preconfigured cluster nodes as they are known by the local monitor. The matrix has as many columns and rows as preconfigured members exist, independently of their current state. Each cell of the matrix holds an integer number that identifies the state of the node represented by that cell of the matrix. The possible states are:

START The node is still being initialized. No message has been broadcast by its monitor.

BEGIN The node is executing the *agreement procedure*. All nodes in this state are exchanging messages to find out which is the current membership set.

STEP(i) The node is notifying a system component in the i-th step of its *notification procedure*. It has also started the *polling procedure* to check the stability of the new membership set.

END The node has terminated the notification procedure and it is only checking the stability of the current membership set.

RETURN The node has detected a membership change. It has to notify this situation to the rest of membership monitors and has to return to the *agreement procedure*; i.e., it is returning to the BEGIN state.

UNKNOWN The state of this node is uncertain. Some expected messages from this node have not arrived on time, but some additional time is needed to declare that the node has failed.

DOWN The node does not reply nor send any message. It has failed.

The cell state[i][j] maintains the state of the node j as known by the node i. Each time a monitor A detects a change in the state of another cluster node B, it modifies the B-th column of the row associated to its local node (the A-th row) and broadcasts this row to the rest of nodes. When a monitor C receives a message from a remote monitor A it replaces the A-th row of its local matrix to the one received in the incoming message. Also, the A-th column of the C-th row is changed to the current value of the state[A][A] cell.

As an example, figure 1 shows how the state matrix of the node 2 is changed when it receives a message from node 3. In this example the cluster has only 3 preconfigured nodes. Before arriving the message, the cluster consists of the nodes 2 and 3, which are both in the END state; node 1 has not started. When node 1 starts, its messages have been received only by node 3, which changes to the RETURN state and broadcasts its matrix row. When this row is received by node 2, this node changes its matrix contents to the ones shown in the second table of the figure. As a result, row 3 is completely replaced and the state[2][3] cell is also updated. Later, node 2 will initiate its transition to the RETURN state, too.

	1	2	3
1	UNK	UNK	UNK
2	DOWN	END	END
3	DOWN	END	END

(a)

	1	2	3
1	UNK	UNK	UNK
2	DOWN	END	RET
3	BEGIN	END	RET

(b)

Figure 1: State matrices of node 2 before (a) and after (b) receiving a message from node 3.

To reach agreement on a given state, all the cells associated to all active nodes[3] in the cluster must have the same state value. When this agreement is achieved, all cluster members transit to the following automaton state. All possible transitions are depicted in figure 2 and are explained in the following subsections.

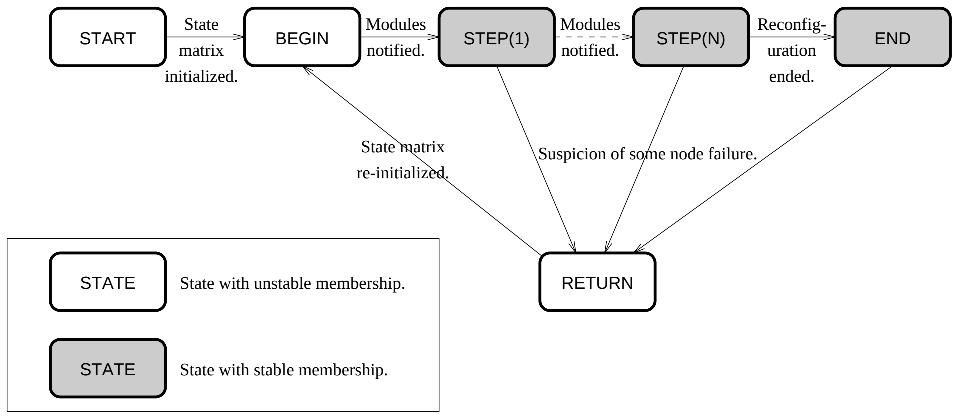
START
State matrix initialized.
BEGIN
Modules notified.
STEP(1)
Modules notified.
STEP(N)
Reconfig- uration ended.
END
State matrix re-initialized.
Suspicion of some node failure.
RETURN
STATE
State with unstable membership.
STATE
State with stable membership.

Figure 2: States and some causes of transition in the CMM automaton.

Other important variables in each monitor are seqnum and local_id. The seqnum variable maintains the number of reconfigurations seen by the local monitor; i.e., how many times the monitor automaton has arrived to the BEGIN state, either due to new nodes joining the cluster or to the failure of some nodes. A copy of this variable is saved in stable storage each time its value is modified. When the local machine is restarted, its value has to be read from stable storage and it is used to tag the initial outgoing messages sent by the local membership monitor.

The local_id variable maintains the identifier of the local node in the cluster membership set. This identifier is an integer number lower than the number of preconfigured members and greater than or equal to zero.

Finally, some other variables are needed to maintain the amount of time assigned to each one of the timers that control the reception or sending of messages. These variables are:

- heartbeat_time. Maximum amount of time that may be elapsed between the emission of two different messages. Once this time has passed, the local monitor must send a message to the other monitors.
- failure_time. Maximum amount of time the local monitor is waiting for a message from a remote monitor before that remote node is considered faulty. If no message has arrived from the intended sender in this time, that sender is registered as DOWN.
- shutdown_time. If the local monitor does not receive any message for shutdown_time microseconds, it shutdowns itself. The lack of messages for this period means that the node remains isolated.

Each monitor maintains a set of timers associated to these variables. There are a heart-beat and a shutdown timers associated to the local node. The *heart-beat timer* is re-instantiated each time the local monitor sends a message. The *shutdown timer* is reset each time the local node receives a message from any other monitor.

There is a pair of reception and failure timers for each one of the supposedly active remote monitors. The *reception timer* is associated to the expected heart-beat messages sent by all other monitors. It is reset each time a message is received from the expected sender. When this timer detects a reception timeout, the state of the associated sender is changed to UNKNOWN. The *failure timer* is also reset each time a message is received from its associated sender monitor.

3.2 Message Contents

The CMM protocol uses two types of messages. The first one is the *regular* message. It is used to transfer the row of the state matrix, either in the *agreement procedure* —to achieve consensus on the current membership set— or in the *polling procedure* to check the stability of the group. The other type is the *step* message, which is used to reach agreement on the termination of a notification step.

[3]A node is *active* when no other cluster node suspects it faulty; i.e., it does not appear in the DOWN state in any cell of its associated column.

```
typedef struct {
        node_t          msg_sender;
        usectime_t      msg_sendertime;
        seqnum_t        msg_seqnum;
        node_state_t    msg_statevec[MAXNODES];
} message_t;
```

Figure 3: Contents of a regular message.

The contents of a regular message are depicted in figure 3 and are explained below:

- *Sender identifier* (msg_sender). Identifier of the node that sent the message. This identifier is set by the local monitor when the message was received, translating the network address of the sender to its identifier. This is done using a table of network addresses for all the preconfigured cluster members. So, machines that have not been preconfigured as possible cluster members can not be included in the membership set.
- *Sender time* (msg_sendertime). Timestamp set by the sender when the message was sent. This allows an accurate control of periodical message timeouts at the receiver side.
- *Reconfiguration sequence number* (msg_seqnum). This is the number of reconfigurations made by the sender. It is needed to check if the message is stale or if the receiver node has lost any messages in the BEGIN state.
- *Sender state row* (msg_statevec). Row of the state matrix where the sender maintains the states of other cluster nodes.

```
typedef struct {
        node_t          msg_sender;
        boolean_t       msg_is_master;
        boolean_t       msg_is_ended;
        unsigned        msg_step_number;
        seqnum_t        msg_seqnum;
} messtep_t;
```

Figure 4: Contents of a step message.

Step messages maintain the reconfiguration sequence number and the identifier of the sender fields of regular messages and replace all other fields by a boolean (msg_is_master) that tells the role its sender is playing in the synchronization needed at the end of the step, another boolean (msg_is_ended) which points out if the local node does not have additional programs to be notified and an integer (msg_step_number) that holds the step number. The contents of a message of this type are shown in figure 4.

3.3 Initialization Procedures

In the START and RETURN states described in section 3.1, the data structures maintained by the monitor have to be initialized. This initialization procedure sets the states of all other nodes to the UNKNOWN value in the state matrix. Also, in the START state, the value of the seqnum variable is read from stable storage. Finally, a set of timers are installed to check the current state of all preconfigured nodes in the cluster. For each remote node there are a pair of timers, the *reception* and the *failure* ones. The *heart-beat* and the *shutdown* timers have to be set for the local node, too. The function associated to each timer has already been described in section 3.1. The steps followed in this procedure are shown in figure 5.

```
procedure initialize;
var
    i, j : node_t;
begin
    for i := 0 to MAXNODES - 1 do
    begin
        for j := 0 to MAXNODES - 1 do
            state[i][j] := UNKNOWN;
        if i <> local_id
        then begin
            reinstall_timer( i, reception_timer );
            reinstall_timer( i, failure_timer );
        end;
    end;
    reinstall_timer( local_id, heart-beat_timer );
    reinstall_timer( local_id, shutdown_timer );
end;
```

Figure 5: Initialization procedure run in the START state.

The reinstall_timer procedure sets the timer received as the second argument for the node received as the first one. Each timer class needs to be set to a different time in the future. For instance, the reception and heart-beat timers are set to the current time plus the value stored in the heartbeat_time variable. The failure timer uses the failure_time variable and the shutdown timer the shutdown_time one.

3.4 Agreement Procedure

In the *agreement procedure* all membership monitors cooperate to find a new membership set when new cluster machines have started or some of the previously running ones have been stopped or have failed. Also, a new configuration *sequence number*, usually the following one in ascending order, is chosen and all messages sent after the agreement procedure will carry it. This sequence number is needed to avoid the processing of stale messages in the membership monitors.

The tasks done in the agreement procedure are outlined in figure 6. The check_agreement function checks if all non-faulty nodes in the local state matrix have the same contents in their associated state rows. If so, the agreement variable is set to true. Moreover, if some cell of the row associated to the local node in the state matrix has changed, the updated_state variable is also set to true. If the agreement has been achieved all membership monitors do a transition to the following automaton state; i.e., to the STEP1 state. This is achieved when the state of the local node is changed in the state matrix.

The bcast_state function broadcasts a regular message to all preconfigured cluster members. It also reinstalls the *heart-beat timer* for the local node.

Finally, the shutdown function initiates a shutdown of the local node, reporting appropriately all system components that have requested this notification. This function is only used when the local node has not received any message from any other preconfigured cluster member for a given time and this may only happen when this node becomes isolated.

The main loop of this procedure waits for external events. These events may be the arrival of a message sent by another monitor or the triggering of a local timer. When a message arrives and its sequence number is equal to the local one, the reception and failure timers are reinstalled for the message sender. The local shutdown timer is reprogrammed, too. Later, the state matrix is updated accordingly to the contents of the incoming message and the resulting state is checked to find out if an agreement has already been achieved. If so, the local automaton goes to the STEP1 state, initiating the *notification procedure*.

If the sequence number of the incoming message is lower than the one maintained by the local monitor, the message is discarded. Otherwise, if it is greater than the local one, the monitor has to update its seqnum variable

```
procedure agreement;
var
  m : message_t;
  n : node_t;
begin
  state[local_id][local_id] := BEGIN;
  while state[local_id][local_id] = BEGIN do
      wait-for event;
      case event of
      recv( m ):
          if m.msg_seqnum = seqnum
          then begin
              reinstall_timer( m.msg_sender, reception_timer );
              reinstall_timer( m.msg_sender, failure_timer );
              reinstall_timer( local_id, shutdown_timer );
              state[m.msg_sender] := m.msg_statevec;
              state[local_id][m.msg_sender] := m.msg_statevec[m.msg_sender];
              check_agreement;
              if agreement
              then begin
                  state[local_id][local_id] := STEP1;
                  seqnum := seqnum + 1;
              end;
              if updated_state
              then begin
                  bcast_state;
                  updated_state := FALSE;
              end;
          else if m.msg_seqnum > seqnum
          then begin
              seqnum := m.msg_seqnum;
              state[local_id][local_id] := RETURN;
              bcast_state;
          end;
      reception_timeout( n ):
          state[local_id][n] := UNKNOWN;
          bcast_state;
      failure_timeout( n ):
          state[local_id][n] := DOWN;
          bcast_state;
      heart-beat_timeout:
          bcast_state;
      shutdown_timeout:
          shutdown;
      esac;
    done;
end;
```

Figure 6: Agreement procedure run in the BEGIN state.

and go to the RETURN state to reinitialize its variables.

The timers set for the remote monitors are used to find out which remote nodes are not active. So, when these timers trigger a timeout event, the state associated to that remote node is updated in the local state matrix. To reduce the time needed to rebuild the membership set when a new member is being added or an old member has failed, the re-initialization made in the RETURN state is slightly different to the one shown in figure 5. In that case, the assignment done in line 8, is only made if the current contents of that cell in the matrix are different to the DOWN value; i.e., faulty or stopped nodes are suspected to remain in that state. Moreover, the timer re-installation made at lines 12 and 13 is only made for nodes that are not in the DOWN state. If some of these faulty nodes has been started, when its first messages are received by the other monitors, the appropriate timers are reinstalled. As a result, the agreement time can be reduced if no timer is initially associated to these nodes.

3.5 Notification Procedure

The *notification procedure* shown in figure 7 is used to report the new membership set to different software components in the local node that depend on the current set of cluster machines. Samples of applications of this kind are the reliable communications layer, the object request broker and several ORB extensions required to provide high availability support [8].

The programs or objects to be reported about the membership changes have to register themselves when the CMM is being initialized or when it is already running. Each node may have a different set of applications registered to be reported in the notification procedure. Each application has to request a different step stage.

At the beginning of the notification procedure, each monitor finds out if it is the current master of this stage. To do so, the is_master function is used (line 8). This function returns TRUE if the local monitor has the lowest identifier among all the current active nodes in the cluster. Once this is done, the main loop of the procedure starts (lines 11 through 34 in figure 7). Each iteration in the loop is used to advance an step, notifying all instances of the program registered for this step in each cluster node. The function registered checks if there is a program registered for the current notification step in the local node. If so, the notify_application function is used to synchronously notify that component about the current membership.

Later, all cluster nodes have to send a step message to the chosen master. When the master has received the appropriate message from each other monitor, it broadcasts an acknowledgment to all them. When the acknowledgment is received, the monitor updates its state matrix accordingly and initiates a transition to the following automaton state, which can be another STEP message or the final END state. In the END state, the notification stage terminates and the local monitor only executes the *polling procedure*.

Since each monitor may have a different number of applications interested in the membership status and additional applications may be registered at any time, the number of required steps have to be decided at runtime. To this end, the step messages carry a msg_is_ended field. This field, when the message is sent by a non-master node, points out whether the sender monitor has no other program to be notified. On the other hand, for the acknowledgment messages sent by the master, a TRUE value means that all monitors have notified all registered programs and the current one was the last STEP needed. Two functions are used to find out the value to be assigned to this message field. In the master side, the check_if_ended function (in line 17) returns TRUE when the master node has notified all its registered programs and it has received step messages from all other monitors pointing out that they also have notified all their programs. In the non-master nodes, the check_if_all_notified function (in line 23) checks if all programs locally registered have been already notified.

3.6 Polling Procedure

The *polling procedure* is started as soon as the *agreement procedure* is terminated. It checks the stability of the membership set found in the agreement stage. To do this, at the start of the polling procedure each monitor chooses two neighbors from the whole membership set, building in this way a logical ring among all current members. Periodically, each monitor checks the state of its two neighbors, sending a message to them and expecting also a message from each one of them. This behavior is controlled by the heart-beat, shutdown, reception and failure timers, as it was done in section 3.4.

The algorithm followed in this procedure is outlined in figure 8. The first steps (lines 6 through 16) are used to cancel the timers set in the BEGIN state and to reinstall them for the neighbor nodes in the logical ring being established.

```
procedure notification;
var
    step : integer := 1;
    ended : boolean := FALSE;
    master : boolean;
    m, n : messtep_t;
begin
    master := is_master( local_id );
    m.msg_seqnum := seqnum;
    m.msg_is_master := master;
    while not ended do
        if registered( step )
        then notify_application( step );
        if master
        then begin
            wait-for reception-of-step-messages;
            ended := check_if_ended;
            m.msg_step_number := step;
            m.msg_is_ended := ended;
            bcast( m );
        end else begin
            m.msg_step_number := step;
            m.msg_is_ended := check_if_all_notified;
            send_to_master( m );
            recv_from_master( n );
            ended := n.msg_is_ended;
        end;
        step := step + 1;
        for i in live_nodes do
            for j in live_nodes do
                if ended
                then state[i][j] := END;
                else state[i][j] := STEP(step);
    done;
end;
```

Figure 7: Notification procedure run in all STEP states.

```
procedure polling;
var
   m : message_t;
   n : node_t;
begin
   for n in live_nodes do begin
        stop_timer( n, reception_timer );
        stop_timer( n, failure_timer );
   end;
   neighbors := find_neighbors;
   for n in neighbors do begin
        reinstall_timer( n, reception_timer );
        reinstall_timer( n, failure_timer );
   end;
   reinstall_timer( local_id, heart-beat_timer );
   reinstall_timer( local_id, shutdown_timer );
   send_state_to_neighbors;
   while state[local_id][local_id] <> RETURN do
        wait-for event;
        case event of
        recv( m ):
             if m.msg_seqnum = seqnum
             then begin
                  reinstall_timer( m.msg_sender, reception_timer );
                  reinstall_timer( m.msg_sender, failure_timer );
                  reinstall_timer( local_id, shutdown_timer );
                  if m.msg_statevec[m.msg_sender] = RETURN
                  then begin
                       state[local_id][local_id] := RETURN;
                       bcast_state;
                  end;
             end else if m.msg_sender not in neighbors
             then begin
                  state[local_id][local_id] := RETURN;
                  bcast_state;
             end;
        reception_timeout( n ):
             state[local_id][n] := UNKNOWN;
        failure_timeout( n ):
             state[local_id][n] := DOWN;
             state[local_id][local_id] := RETURN;
             bcast_state;
        heart-beat_timeout:
             send_state_to_neighbors;
        shutdown_timeout:
             shutdown;
        esac;
     done;
end;
```

Figure 8: Polling procedure run in the STEP and END states.

Later, the local monitor starts the main loop, sending periodically a message to its both neighbors and expecting the arrival of their messages. If some neighbor does not send any message for a failure_time interval, it is considered faulty and the local monitor changes to the RETURN state, broadcasting its associated row to all other cluster nodes. As a result, once a monitor has detected the failure of another one, all monitors eventually arrive to the BEGIN state, initiating another *agreement procedure*. Note also that the polling and the notification procedures may be executed simultaneously. If a failure is detected when both are being executed, the decision taken by the polling procedure has the highest priority, and the local automaton goes to the RETURN state.

Finally, the lines 32 through 36 are needed to detect the attempts of a new node to join the cluster and they also lead the local automaton to the RETURN state.

3.7 Isolated Nodes

Section 3.4 outlines a mechanism to halt isolated nodes based on the local *shutdown timer* that is reset each time the local node receives a message from another cluster node. But this timer is only useful when only one node is isolated from the rest of the cluster, preventing its interaction with cluster clients.

A source of isolated nodes is a cluster partition. A partition arises when some of the active cluster nodes are not able to communicate with other active nodes in the same cluster. As a result of a partition some subgroups of machines can be identified. This situation can be considered a problem, since the members of each subgroup can receive messages from other cluster nodes, but not from all active nodes. As a result, the distributed applications or services running on the remaining subgroups do not have a state known by all their active components. So, in a multi-computer cluster environment only one of the resulting subgroups can be allowed to proceed. All other ones have to be shutdown, but in this case the local shutdown timer becomes completely unuseful.

For managing partitions, the protocol provides two mechanisms:

- Require a minimum number of nodes in the current cluster. If this number can not be achieved, the nodes are shutdown. So, if a cluster becomes partitioned, only the subgroup which exceeds the required population remains. Obviously, the required minimum must be high enough to avoid two remaining subgroups.
- The local software modules notified in the STEP states are able to request a shutdown of the local or some remote nodes. These software modules are at kernel level and they are assumed trustful.

  When one of these modules notes that the current membership set is insufficient and the service it is carrying on is essential for the cluster, it can request the shutdown of its subgroup (which can be the whole cluster).

Anyway, the partition problem seldom arises in a multi-computer cluster. Usually, the cluster nodes are physically close each other and the interconnecting network can be easily replicated. With a system of this kind, communication problems may only arise due to an excessive load which slows down the message delivery time. So, physical partitions are not frequent.

4 Performance

The cost in messages and elapsed time of a reconfiguration (either a join or a failure) depends mainly on the number of machines in the resulting cluster (N). Besides this, a joining reconfiguration can be initiated at any time but a reconfiguration to drop some node is delayed a bit. This sort of reconfiguration is triggered when a period of failure_time microseconds has been elapsed since the last message from the faulty node was received by the other monitors.

Theoretically, to get the next membership set the following point-to-point messages are needed:

- The node which detects the membership change sends N messages to the other members. This sending leads all member nodes to the RETURN state.
- On reception of this message, each other member notifies its local software modules about what is happening, and eventually arrives to the BEGIN state.
- Once in the BEGIN state, each node broadcasts its local row. That implies N^2 messages.

In the best case a node can receive a message from any live node before replying to any of them. In the worst case, the node receives a message from any other node which implies a change in the row associated to the local node and broadcasts a message for each one of them; when the node receives a second or later message from a node, it does not reply to it. Consequently, the BEGIN state ends with an upper bound of N^3 messages.

- Later, in the STEP and END states only $2N$ messages are needed at the end of each step or every heart-beat_time microseconds; i.e., for each round of messages.

So, the cost in messages is $O(N^3)$ in a cluster of N nodes.

5 Correctness

An argument about the accomplishment of the requirements introduced in section 2 is given below. It is not a formal prove, we only describe the behavior of the algorithm in the situations required in that section.

R1 Each active node in the cluster must maintain the same membership set.

The membership set is built in the BEGIN state. Using the state matrix, we ensure that each node leaves the BEGIN state only if each member of the cluster shares the same matrix contents.

When a node monitor detects a variation in its membership set, it goes to the RETURN state and eventually all cluster members re-initiate a BEGIN state. So, either the membership set is stable and shared by all cluster members or it is unstable and the active nodes are in the RETURN or BEGIN state, building the new one.

If the cluster is partitioned, the cluster could be in a situation where more than one membership set has been built and different active nodes assume different membership sets. In section 3.7 we have shown how this situation is avoided.

R2 Only preconfigured nodes can be members of the cluster.

We use a configuration file where the network address of each possible cluster member is associated to a logical identifier. Only machines which appear in this file can be members of the cluster. So, external nodes can not receive protocol messages and can not take the place of a real cluster member.

R3 A node can be integrated in the membership set at any time.

When a new node tries to join the cluster, it broadcasts a regular message. These messages will lead the cluster to the RETURN state and in the subsequent BEGIN state, the new node is included in the cluster.

This procedure allows multiple node joins, too. If several nodes are awaken at the same time, they can be included in the cluster in the BEGIN state of the same reconfiguration.

Moreover, the delay of a (possibly multiple) node join is bounded. The maximum number of messages which can be exchanged due to a join is known and the time needed to do so is easily predictable.

R4 A node considered faulty is removed from the membership set in a bounded delay.

When a node is faulty, extremely slow or is attached to a faulty link, other nodes detect this situation and initiate a reconfiguration sequence. In the next BEGIN step, the node is excluded from the membership set.

The node failure is detected using periodical messages in the STEP or END states. Its two neighbors in the logical ring detect the problem. Then a reconfiguration is initiated and the membership set is rebuilt.

R5 Partitions are not allowed.

This is only a consequence of the first requirement.

6 Related Work

There has been a lot of work related to membership protocols. The case of a multi-computer cluster as the target system simplifies some of the problems to be solved. For instance, we can assume that the interconnecting network is private to the cluster members and their membership monitors are well-behaved. So, our protocol does not have to worry about malicious failure suspectors, as is the case of the one proposed in [18].

Another property that has to be required in a cluster membership protocol is to avoid partitions. A multi-computer cluster behaves as a single system and its components share their state. It can not tolerate its division in two or more subgroups which believe that they are the whole cluster, since this might lead to inconsistencies in the cluster data or in the responses provided to external clients. Some other systems, for instance [5, 11, 18, 19], neither tolerate partitions but their way to guarantee that minor partitions realize they have been excluded from the group are different. In their case, the dropped members detect that the current group size is not big enough and leave the group. In our case, this mechanism is aided by a timer which lead to the shutdown of the node in case of becoming isolated. This stronger guarantee is needed to avoid any communication among the isolated node and remote clients, as explained above.

Group members can have some degree of centralization. Some systems give any member the same role in the protocol, as in the algorithms described in [5, 11]. So, the protocol is fully distributed and does not depend on any centralized authority. On the other hand, [18] and [19] distinguish the figure of a manager node which controls a two-phase protocol each time a membership change occurs. The former alternative has the advantage that the nodes have not to agree in which is the current leader or manager of the group, and reconfigurations are not penalized when this most privileged node fails. However, if a manager node is used, the amount of messages to be exchanged when the membership set is reconfigured can be reduced extremely. Our protocol is a mixture of these two approaches. In the reconfiguration stage, it behaves as a decentralized algorithm, and each future member broadcasts a similar number of messages. No worries are needed to elect a special node at this stage. So, the protocol is quite sound when a reconfiguration is done. When the membership set has been computed, we define a subsequent stage to report to other system components about the new members. This task can be done in a set of sub-stages. So, we need some synchronization among the cluster members (otherwise, an individual node may become reconfigured before the others and this fact can be dangerous in a system like ours). To achieve this agreement on the completion of sub-stages, we devise a centralized sub-protocol where a manager is elected. But the leader election is trivial now. We already know which members belong to the cluster and each member manages exactly the same information. So, the election can be done locally in each node and no additional message is required. Finally, when the membership set is already stable, we follow a protocol similar to the third appeared in [5]. This protocol is also decentralized and requires that each member exchanges messages with another two members of the cluster. So, we have elected a decentralized algorithm which requires a low number of messages and guarantees a fast detection of failures. Other approaches, as the use of a token that is passed along a logical ring ([1], [11] and the second protocol of [5]), require a lower number of messages but the time needed to detect multiple failures is longer. In a multi-computer cluster, it is very important to detect failures as soon as possible.

The assignment of logical identifiers to the group members also has several flavors. Systems like [19] propose to assign a different identifier for a node in each of its incarnations. So, when a node is repaired and initiated again it is considered by the protocol as a different machine. In our case, the use of a dedicated interconnecting network among the cluster members ensures that the maximum number of nodes and their identity can be known a priori. To avoid confusion among successive reconfigurations of the same machine, a sequence number is used and attached to all messages. So, stale messages can be easily discarded.

The use of constant logical identifiers for all nodes simplifies the relationship between the cluster membership monitor and other components of the distributed operating system.

As we have seen above, the characteristics required to a membership protocol for a multi-computer cluster are not new. They can be found in some previous works. However, none of these have been appropriately tailored for a cluster system. In this kind of systems, the abortion of isolated nodes and the time spent to detect some failure are critical. We believe our algorithm is a reasonable solution to both problems.

7 Summary

We have outlined the requirements that have to be accomplished by a multi-computer cluster membership protocol and we have described a possible solution for a system of this kind. Our protocol combines some ideas of previous protocols to achieve its main goals: fast failure detection, avoidance of partitions, assurance of fail stop behavior and a sound reconfiguration procedure.

Besides solving the membership problem, our protocol provides mechanisms to link the membership monitor to other operating system's local components which depend on the membership service. Also, it ensures that the reconfiguration procedure is highly synchronized among the membership members when the new membership set has been built, avoiding inconsistencies.

It will be used to provide membership services to the reliable transport layer of the HIDRA architecture. It also assists in the reconfiguration of the ORB and all their components related to the high availability support being provided by this architecture.

So, the protocol described in this paper seems to be appropriate to a particular kind of system — multi-computer clusters —, such as the one introduced here.

References

[1] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. Fast message ordering and membership using a logical token-passing ring. In *Proceedings of the 13th International IEEE Conference on Distributed Computing Systems*, pages 551–560, May 1993.

[2] Ö. Babaoğlu, R. Davoli, L. A. Giachini, and M. Baker. Relacs: A communications infrastructure for constructing reliable applications in large-scale distributed systems. Technical report, UBLCS-94-15, Dept. of Computer Science, University of Bologna, Bologna, Italy, June 1994.

[3] K. P. Birman and B. B. Glade. Consistent failure reporting in reliable communication systems. Technical report, TR93-1349, Dept. of Computer Science, Cornell University, Ithaca, NY, May 1993.

[4] K. P. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, Los Alamitos, CA, 1994.

[5] F. Cristian. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing*, 6(4):175–187, 1991.

[6] D. Dolev, D. Malki, and R. Strong. An asynchronous membership protocol that tolerates partitions. Technical report, CS94-6, Institute of Computer Science, The Hebrew University of Jerusalem, Israel, March 1994.

[7] D. Dolev, D. Malki, and R. Strong. A framework for partitionable membership service. Technical report, CS95-4, Insitute of Computer Science, The Hebrew University of Jerusalem, Israel, 1995.

[8] P. Galdámez, F. D. Muñoz-Escoí, and J. M. Bernabéu-Aubán. HIDRA: Architecture and high availability support. Technical report, DSIC-II/14/97, Univ. Politècnica de València, Spain, May 1997.

[9] M. A. Hiltunen and R. D. Schlichting. Understanding membership. Technical report, 95-07, Dept. of Computer Science, The University of Arizona, Tucson, AZ, July 1995.

[10] M. A. Hiltunen and R. D. Schlichting. A configurable membership service. Technical report, 94-37A, Dept. of Computer Science, The University of Arizona, Tucson, AZ, January 1996.

[11] H. Kopetz and G. Grünsteidl. TTP - A protocol for fault-tolerant real-time systems. *IEEE Computer*, 27(1):14–23, January 1994.

[12] C. Malloth, P. Felber, A. Schiper, and U. Wilhelm. Phoenix: A toolkit for building fault-tolerant distributed applications in large scale. Technical report, Dépt. d'Informatique, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, July 1995.

[13] S. Mishra, L. L. Peterson, and R. D. Schlichting. A membership protocol based on partial order. In J. F. Meyer and R. D. Schlichting, editors, *Dependable Computing for Critical Applications*, pages 309–331. Springer-Verlag, Wien, 1992.

[14] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. Budhia, and C. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, April 1996.

[15] F. D. Muñoz-Escoí, J. M. Bernabéu-Aubán, and P. Galdámez. The group membership problem and its solutions. Technical report, DSIC-II/8/97, Univ. Politècnica de València, Spain, May 1997.

[16] OMG. *The Common Object Request Broker: Architecture and Specification*. Object Management Group, July 1995. Revision 2.0.

[17] R. Rajkumar, S. Fakhouri, and F. Jahanian. Processor group membership protocols: Specification, design and implementation. In *Proceedings of the 12th IEEE Symposium on Reliable Distributed Systems, Princeton, NJ*, pages 2–11, October 1993.

[18] M. K. Reiter. A secure group membership protocol. Technical report, AT&T Bell Labs., 1994.

[19] A. M. Ricciardi. The group membership problem in asynchronous systems. *Ph.D. dissertation (also available as TR92-1313), Dept. of Computer Science, Cornell University, Ithaca, NY*, page 198 pgs, January 1993.

[20] L. Rodrigues, P. Verı́ssimo, and J. Rufino. A low-level processor group membership protocol for LANs. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 541–50, May 1993.

[21] R. D. Schlichting and F. B. Schneider. Fail-stop processors: An approach to designing fault-tolerant systems. *ACM Trans. on Computer Sys.*, 1(3), August 1983.

[22] R. van Renesse, K. P. Birman, B. Glade, K. Guo, M. Hayden, T. M. Hickey, D. Malki, A. Vaysburd, and W. Vogels. Horus: A flexible group communications system. Technical report, TR95-1500, Dept. of Computer Science, Cornell University, Ithaca, NY, March 1995.