HMM: A Cluster Membership Service*

Francesc D. Muñoz-Escoí, Óscar Gomis, Pablo Galdámez, and José M. Bernabéu-Aubán

Instituto Tecnológico de Informática, Universidad Politécnica de Valencia
Camino de Vera, s/n, 46071 Valencia, Spain
{fmunyoz,ogomis,pgaldam,josep}@iti.upv.es

Abstract. The *Hidra Membership Monitor* (HMM) is a distributed service that maintains the current set of active nodes in a cluster of machines. This protocol allows the detection of multiple machine joins or failures in a unique reconfiguration, using a low amount of messages (with a cost that is linear on the number of nodes). These membership services are needed to detect cluster changes as soon as possible, initiating then the reconfiguration of the cluster state, where support for replicated objects has been included.
The HMM also manages and synchronises the reconfiguration steps needed by the kernel and Hidra components of each node, ensuring that all of them take the same steps at once. Thus, our system does not need an atomic multicast protocol to deliver the messages in these reconfiguration steps. All these services provide the basis to develop reliable intracluster transport protocols and to reduce the reconfiguration time of replicated objects and services.

1 Introduction

Hidra [3,8] is an architecture that provides high availability support in a distributed environment based on a cluster of machines interconnected by a private network. This architecture uses a low-level ORB [9] placed in the kernel of each node which includes the support needed for replicated objects. Thus, replicated objects can be used either in user-level applications or in kernel components needed to provide a single-system image.

Our ORB is not completely CORBA-compliant, since it extends some of the support that such an ORB must provide. Particularly, support for replicated objects and reference counting has not been considered in CORBA as a service of the ORB core, but they have been included in ours [6,4]. Both reference counting and replication support need a membership service to know which are the current active machines in the cluster and to reconfigure their state when a new machine joins or a previously active node fails.

In general, a distributed system with high availability support needs a membership service. This service must be placed in the lowest layers of the system

* This work was partially supported by the CICYT (Comisión Interministerial de Ciencia y Tecnología) under project TIC99-0280-C02.

R. Sakellariou et al. (Eds.): Euro-Par 2001, LNCS 2150, pp. 773–782, 2001.

architecture to assist in the development of other fault tolerant services, and to drive the reconfiguration protocols of these components, either in case of failures or in case of new machines joining the system.

In our Hidra architecture, the reconfiguration protocols that must be executed when a node fails or joins the cluster, need be synchronised, i.e., a reconfiguration step has to be started only when all the active nodes have concluded its previous step. Thus, the reconfiguration protocols know that the results of other previous protocols are ready in all system nodes. This characteristic has been used to recompute the reference counts of the ORB, which needs several reconfiguration steps, and the state of the main serialiser object [7], whose reconfiguration must be done afterwards.

HMM provides this kind of synchronisation in its *Steps* stage. Thus, our protocol manages in parallel the notification steps, their sinchronisation and the periodic interchange of messages among live nodes. As a result, only a few messages are needed to develop all these tasks.

The rest of the paper is structured as follows. Section 2 describes the system model. Section 3 describes the entire protocol. Section 4 compares HMM with other membership protocols and, finally, Sect. 5 provides the conclusions.

2 System Model

HMM is aimed to provide membership monitoring in a dynamic cluster with a preconfigured maximum number of machines interconnected by a private network. This scope determines the behaviour of the system and the assumptions made by our protocol. Thus, cluster nodes have static identifiers, but these identifiers are complemented with a node sequence number that is increased each time the node is restarted and allows us to use a fail-stop failure mode (since each time a node is restarted, it joins the cluster with a different identifier).

No clock synchronisation is needed, but a maximum message delivery time must exist. This allows us to decide when a message is lost, either due to a network failure, a sender failure or lack of available buffers to receive it.

We also assume that the private network being used presents a kind of interconnection that prevents the appearance of network partitions. This assumption is valid for a small cluster and some network topologies.

HMM uses the services of an unreliable transport layer which is assumed to provide broadcast, multicast and point-to-point transmissions. It does not check the delivery of such messages nor waits for any acknowledgement.

On top of the HMM, our ORB core uses the HMM notification services. To this end, the HMM object provides a set of operations in its interface that allow the registration of any Hidra component that wants to be notified when a given reconfiguration sequence step number has been initiated. These steps constitute the core of the ORB and associated components reconfiguration tasks and are driven by the HMM.

3 Protocol

HMM has to deal with either unique or multiple joins and failures in a single reconfiguration of the cluster membership. This objective has been attained with a sequence of stages (see Fig. 1) that manage the detection of a membership change and its reconfiguration steps. Moreover, cluster nodes have a role that can change in each stage.

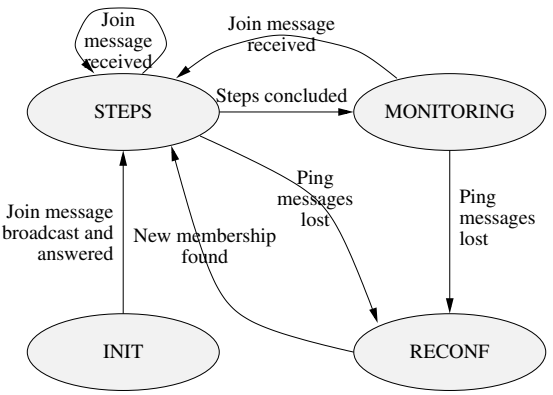

Fig. 1. HMM stages and transitions

3.1 Member Roles

A node that uses the HMM may play one of the following roles:

- *Master*. The basic role of the *master* node is to drive the reconfiguration steps and to accept `join` messages when new members arrive.
- *Slave*. Once a stable membership set has been found and the first step in the *Steps* stage has been started, the *slaves* must receive the *master*'s messages, notify the registered packages about them and reply to the master.
- *Beginner*. This role applies to the node that detects a given number of `ping` message losses. Once this situation arises, the sender of these messages is assumed faulty, and the *beginner* node enters the *Reconf* stage.
- *Unknown*. When the *Reconf* stage is entered, all members that are not *beginners* are marked as *unknown*.

3.2 Algorithm

The main algorithm is implemented as an automaton that uses the global variables shown in lines 8 to 14 of Fig. 2. There are several timers not shown in the algorithm. They detect missed `step`, `ends` and `endp` messages. Thus, the sender of the missed message can send it again.

```
 1: algorithm hmm;
 2: type
 3:    stage_t = ( INIT, STEPS,
 4:              MONITORING, RECONF );
 5:    role_t = ( MASTER, SLAVE,
 6:              BEGINNER, UNKNOWN );
 7: var
 8:    stage : stage_t;  (* Current stage. *)
 9:    thisid : node_t;  (* Local node ID. *)
10:    step : integer;   (* Step number. *)
11:    masterid : node_t;  (* Current master ID. *)
12:    set : nodeset_t;  (* Membership. *)
13:    seqnum : seqnum_t;  (* Config number. *)
14:    role : role_t;   (* Role of this node. *)
15: begin
16:    stage := INIT;
17:    masterid := thisid;
18:    seqnum := -1;
19:    step := 0;
20:    while true do
21:      case stage of
22:      INIT:  set := emptySet;
23:             addMember( set, thisid );
24:             role := MASTER;
25:             stInit;
26:      STEPS:              stSteps;
27:      MONITORING:         stMonitoring;
28:      RECONF:             stReconf;
29:      esac;
30: end;
```

Fig. 2. Main automaton of HMM

As we can see in lines 16 to 25 of Fig. 2, each node starts in the *Init* stage and plays the *master* role. The sequence number is initialised to -1 to ensure that it will start with the zero value or the value proposed by the elected *master* node. The membership set is initialised with only the local node. Once this node joins the cluster, it receives the complete membership set.

```
 1: algorithm stInit;
 2: var
 3:    theMsg : msg; elapsed : boolean;
 4: begin
 5:    elapsed := FALSE;
 6:    theMsg.kind = JOIN;
 7:    bcast( theMsg );
 8:    installTimer( joinTime );
 9:    while not elapsed do begin
10:      waitFor event;
11:      case event of
12:      recv( theMsg ):
13:        if theMsg.kind = NEWMEM
14:        then begin
15:          role := SLAVE;
16:          masterid := theMsg.sender;
17:          stage := STEPS;
18:          requeueMessage;
19:          elapsed := TRUE;
20:        end else if theMsg.kind = JOIN
21:        then if theMsg.sender     thisid
22:          then begin
23:            stage := STEPS;
24:            addMember(set, theMsg.sender);
25:          end else begin
26:            stage := INIT;
27:            elapsed := TRUE;
28:          end;
29:      joinTimeout: elapsed := TRUE;
30:      esac;
31:    end;
32: end;
```

Fig. 3. Algorithm of the *Init* stage

All the protocol stages are described in the following paragraphs.

Init Stage: Figure 3 shows the *Init* stage. The local node builds a `join` message and broadcasts it (lines 6 and 7). Later, it installs a timer (line 8) that raises a timeout when `joinTime` has elapsed.

An event is later expected. It can be the reception of a message (line 12) or the timeout signal (line 29). In the first case, HMM checks if the `join` message has been answered by a *master* node using a `newmem` message. If so, the role is changed to *slave* and the stage is changed to *Steps* where this message will be processed (lines 15 to 19).

If a `join` message is received in this stage, the sender static identifier is checked. If it is greater than the local identifier, the sender is added to the current membership set (lines 23 to 24) and the stage is marked as *Steps*, but no transition is made until the join timeout is signaled. Thus, multiple nodes can be added simultaneously when all nodes are powered on at once. If the sender identifier is lower than the local one, this stage is reinitiated immediately.

Steps Stage: The algorithm of the *Steps* stage appears in Fig. 4. Once a node has entered this stage, it enables the sending and reception of `ping` messages in line 5. In this and the *Monitoring* stages the nodes are arranged in a logical ring, using an increasing order of their static node identifiers. Thus, each node periodically sends a `ping` message to its neighbour with greater static identifier, except for the last node of the ring, which sends it to the first node in this order.

Lines 7 to 39 contain the code that has to be executed by the *master* node. In each one of the steps, the master has to send a message to all current members. This task is done using the multicast procedure provided by the unreliable transport. Lines 8 to 18 give the message contents, depending on the step being processed. When step 0 is initiated, the cluster sequence number is increased.

The notifyStep in line 19 is a procedure that invokes all registered Hidra components that are interested in the current step number.

Once all these tasks have been completed, the *master* waits for an event (line 20). If all members have answered with `ends` or `endp` messages, the allAnswersReceived case is taken. If so, the step number is increased and the *Steps* stage is entered again. When all members have sent their final `endp` message (line 24), the *Steps* stage is left and the *Monitoring* stage is entered. If a `join` message is received, its sender is added to the membership set, and the step number is reset to zero. As a result, the *Steps* stage is reinitiated. When some `ping` messages have been lost, a pingTimeout arises (lines 31 to 34), the role is changed to *beginner* and the stage becomes a *Reconf*. Finally, when some `change` message arrives, the role is changed to *unknown* and the stage to *Reconf* (lines 35 to 38).

On the other hand, if the node has a *slave* role, its tasks are shown in lines 41 to 67 of Fig. 4. To begin with, once it has entered this stage it waits immediately for an event. The usual case is the reception of a message, either a `newmem` (lines 43 to 48) or a `step` one (lines 49 to 52). In both cases, the registered clients are informed about the beginning of the step, using the notifyStep procedure described above. Once this has concluded, an answer is returned to the *master* node. Finally, the step is increased in both cases. If the received message was a

```
 1: algorithm stSteps;
 2: var
 3:    theMsg : msg;
 4: begin
 5:    enablePings;
 6:    if role = MASTER
 7:    then begin
 8:       if step = 0
 9:       then begin
10:          theMsg.kind = NEWMEM;
11:          theMsg.contents = set;
12:          seqnum := seqnum + 1;
13:       end else begin
14:          theMsg.kind = STEP;
15:          theMsg.contents = step;
16:       end;
17:       theMsg.seqnum = seqnum;
18:       multicast( theMsg );
19:       notifyStep;
20:       waitFor event;
21:       case event of
22:       allAnswersReceived:
23:          step := step + 1;
24:       allStepsConcluded:
25:          step := 0;
26:          stage := MONITORING;
27:       joinReceived:
28:          addMember( set,
29:             sender );
30:          step := 0;
31:       pingTimeout:
32:          step := 0;
33:          role := BEGINNER;
34:          stage := RECONF;
35:       changeReceived:
36:          step := 0;
37:          role := UNKNOWN;
38:          stage := RECONF;
39:       end;
40:    end else begin (* Role is SLAVE. *)
41:       waitFor event;
42:       case event of
43:       newmemReceived:
44:          setMembership( receivedMsg,
45:             set, seqnum );
46:          notifyStep;
47:          sendEndsOrEndp;
48:          step := step + 1;
49:       stepReceived:
50:          notifyStep;
51:          sendEndsOrEndp;
52:          step := step + 1;
53:       joinReceived:(* Ignore JOINs *);
54:       pingTimeout:
55:          step := 0;
56:          role := BEGINNER;
57:          stage := RECONF;
58:       changeReceived:
59:          step := 0;
60:          role := UNKNOWN;
61:          stage := RECONF;
62:       end;
63:       if stepsConcluded then begin
64:          step := 0;
65:          stage := MONITORING;
66:       end;
67:    end;
68: end;
```

Fig. 4. Algorithm of the *Steps* stage

`newmem` one, the membership set and the configuration number of the node are updated according to the message contents.

Another possibility is the reception of a `join` message, but this type of message cannot be managed by a *slave* node and it is ignored. The last two cases are managed the same way as the *master* node did.

Monitoring Stage: This stage is shown in Fig. 5. Take into account that the procedure enablePings shown at the *Steps* stage, created a thread that periodically sends `ping` messages to its neighbour. This thread still runs in this stage.

In lines 3 to 24 there is a loop that only terminates when the stage is changed. In this loop an event is waited for. Lines 7 to 9 deal with a `newmem` message, leading to the *Steps* stage where this message will be processed. Lines 10 to 12 process a `ping` timeout, i.e., when several `ping` messages have been lost. In

```
 1: algorithm stMonitoring;
 2: begin
 3:    while stage = MONITORING do
 4:    begin
 5:       waitFor event;
 6:       case event of
 7:       newmemReceived:
 8:          stage := STEPS;
 9:          requeueMessage;
10:       pingTimeout:
11:          role := BEGINNER;
12:          stage := RECONF;
13:       joinReceived:
14:          if role = MASTER
15:          then begin
16:             addMember( set,
17:                sender );
18:             stage := STEPS;
19:          end;
20:       changeReceived:
21:          step := 0;
22:          role := UNKNOWN;
23:          stage := RECONF;
24:       esac;
25:    end;
26: end;
```

Fig. 5. Algorithm of the *Monitoring* stage

that case, the stage is changed to *Reconf* and the role of the detector becomes *beginner*. On the other hand, lines 13 to 19 maintain the actions to be taken when a `join` message is received. If the receiver node is the *master*, it adds the new node to the membership set, and changes the stage to *Steps*, otherwise the message has to be ignored. Finally, when a `change` message is received (lines 20 to 23), the behaviour is the same shown in the previous stage.

Reconf Stage: Figure 6 shows the *Reconf* stage. While the nodes are in this stage, `ping` messages are not sent nor expected. To this end, the disablePings procedure is used in line 3.

The rest of the code depends on the role the local node plays. If it is a *beginner*, it executes the instructions in lines 6 to 29, otherwise, it executes that contained in lines 31 to 47. Note that several simultaneous failures may arise, and several *beginner*s may exist.

In case of a *beginner* node, it starts broadcasting a `change` message (line 7). Next, the local node is added to the membership set and a reconfiguration timer is set in line 9. When this time expires, all nodes that have replied with an `alive` message form the next membership set. Later, an event is waited for. Thus, if an `alive` message is received, its sender is included in the membership set being built; if a `change` message is received, an `alive` message is replied to its sender. This answer is needed because several simultaneous failures may arise, and all *beginner*s must behave as the rest of the nodes when a `change` message arrives. When the reconfiguration timer is out, a new master is chosen among all live nodes using the getMaster function, (line 18) and a `setmem` message is sent to it. Finally, if a `setmem` message is received the node becomes *master* or if the message is a `newmem` one, it becomes *slave* and in both cases a transition is initiated to the *Steps* stage. These messages are generated by other *beginner* nodes that have concluded this stage of the protocol before the local one. Additionally, if a `setmem` message has been received, its message contents are used to build the new membership set.

```
 1: algorithm stReconf;
 2: begin
 3:    disablePings;
 4:    set := emptySet;
 5:    if role = BEGINNER
 6:    then begin
 7:       broadcastChange;
 8:       addMember( set, thisid );
 9:       installTimer( reconfTime );
10:       repeat
11:          waitFor event;
12:          case event of
13:          aliveReceived:
14:             addMember( set, sender );
15:          changeReceived:
16:             replyAlive;
17:          reconfTimeout:
18:             masterid := getMaster( set );
19:             send( masterid, setmemMsg );
20:          setmemReceived:
21:             role := MASTER;
22:             stage := STEPS;
23:             setMembers(set, msgContents);
24:          newmemReceived:
25:             role := SLAVE;
26:             stage := STEPS;
27:             requeueMessage;
28:          esac;
29:       until stage       RECONF;
30:    end else begin
31:       replyAlive;
32:       repeat
33:          waitFor event;
34:          case event of
35:          changeReceived:
36:             replyAlive;
37:          setmemReceived:
38:             role := MASTER;
39:             stage := STEPS;
40:             setMembers(set, msgContents);
41:          newmemReceived:
42:             role := SLAVE;
43:             stage := STEPS;
44:             requeueMessage;
45:          esac;
46:       until stage       RECONF;
47:    end;
48: end;
```

Fig. 6. Algorithm of the *Reconf* stage

On the other hand, if the node plays the *unknown* role, it replies immediately to its known *beginner* and later it waits for a message. Several messages are accepted in this case, but they are treated as they had been in the other role.

4 Related Work

In [2] three membership algorithms are described and an environment with bound delivery time is assumed. Its members are machines and no network partition management is considered, as in our algorithm. However, it assumes atomic multicasts that require an additional –and costly– protocol to implement them. Its first algorithm is called *periodic broadcast*. In it, each member broadcasts periodically a message indicating that it is present and that it belongs to the group. To join a group, the new node has to broadcast a different message. It is also able to detect multiple failures and joins, but requires a lot of messages to do so.

The second and third algorithms of [2] are quite similar and we only describe the third one (*neighbour surveillance*). It reduces the amount of messages needed to check the stability of the membership set. The members are arranged in a logical ring and they only send a message to one of their neighbours. However, when a failure is detected all live members have to initiate a round of atomic broadcasts to rebuild the set. Our algorithm requires a lower amount of messages

to do so. Like our algorithm, these protocols are able to detect multiple joins and failures.

In [5], a redundant broadcast channel interconnects all cluster nodes. It is in a real-time system, and a *time-division multiple-access* (TDMA) schema is used to gain access to the network. So, each node has an access period (or sending slot) to the network in each TDMA round. In this environment, a node is considered faulty if it has not sent anything in a given number of TDMA rounds, and all nodes are aware of that fault. When a node restarts, it joins the cluster simply sending a message in its sending slot for the current TDMA round (but it has to wait until that sending slot arrives). Its main disadvantage is that it requires that all nodes share the same clock or that they have highly synchronised clocks. This requirement is not needed by our algorithm and it is difficult to achieve in general-purpose systems. As an advantage, multiple failures and joins are detected immediately without any extra messages.

In the *strong group membership protocol* of [10], a solution quite close to ours is described. It uses a similar algorithm to join a new member and a logical ring with a master once the membership set is stable. Once the ring is built, a member sends heart-beat messages to its both neighbours. However, in case of failure, the master is searched for initiating the reconfiguration. Their solution does not work with multiple failures, since they depend on its master or in a submaster, but if both fail, all nodes have to be restarted and they have to initiate again all the protocol. They also use a two-phase commit protocol to commit the changes, driven by the master. This solution requires more messages than ours in case of join detection and in case of multiple failures or joins.

The following table summarises the worst-case costs (expressed in transmitted messages) of several membership algorithms, assuming that no broadcast service is available and that the number of nodes in the cluster is "N". The "join" and "failure" columns describe the amount of messages needed when a join or failure arises, respectively. The last column gives how many messages are needed in each of the monitoring rounds.

Protocol	**Amount of messages**		
	Join	**Failure**	**Monitoring**
Periodic broadcast [2]	N^2 - N	N^2 - N	N^2 - N
Neighbour surveillance [2]	N^2 - N	N^2 - N	N
Delta-4 [12]	4N - 4	4N - 4	2N - 2
Totem [1]	$2N^2$ + 2N	6N	N
Strong [10]	3N - 2	3N - 2	2N
Isis [11]	3N - 2	5N - 5	unknown
TTP [5]	N^2 - N	N^2 - N	N
HMM	2N - 1	4N - 2	N

5 Conclusions

We have presented a membership protocol for a multi-computer cluster with machine granularity, integrated failure detectors, management of step-synchronous

reconfiguration protocols and a low amount of messages needed to accomplish its tasks. It does not need clock synchronisation among the nodes that participate in the protocol, although the distributed system where it runs cannot be considered totally asynchronous.

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 *Proc. of the 13th International Conference on Distributed Computing Systems*, pages 551–560, Pittsburgh, PA, EE.UU., May 1993. IEEE-CS Press. 781
2. F. Cristian. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing*, 6(4):175–187, 1991. 780, 781
3. P. Galdámez, F. D. Muñoz-Escoí, and J. M. Bernabéu-Aubán. High availability support in CORBA environments. In F. Plášil and K. G. Jeffery, editors, *24th Seminar on Current Trends in Theory and Practice of Informatics, Milovy, Czech Republic*, volume 1338 of *LNCS*, pages 407–414. Springer Verlag, November 1997. 773
4. P. Galdámez, F. D. Muñoz-Escoí, and J. M. Bernabéu-Aubán. Garbage collection for mobile and replicated objects. In J. Pavelka, G. Tel, and M. Bartosek, editors, *26th Seminar on Current Trends in Theory and Practice of Informatics, Milovy, Czech Republic*, volume 1725 of *LNCS*, pages 373–380. Springer Verlag, November 1999. 773
5. H. Kopetz and G. Grünsteidl. TTP - A protocol for fault-tolerant real-time systems. *IEEE Computer*, pages 14–23, January 1994. 781
6. F. D. Muñoz-Escoí, P. Galdámez, and J. M. Bernabéu-Aubán. ROI: An invocation mechanism for replicated objects. In *Proc. of the 17th IEEE Symposium on Reliable Distributed Systems, Purdue Univ., West Lafayette, IN, USA*, pages 29–35, October 1998. 773
7. F. D. Muñoz-Escoí, P. Galdámez, and J. M. Bernabéu-Aubán. A synchronisation mechanism for replicated objects. In B. Rovan, editor, *Proc. of the 25th Conference on Current Trends in Theory and Practice of Informatics, Jasná, Slovakia*, volume 1521 of *LNCS*, pages 389–398. Springer Verlag, November 1998. 774
8. F. D. Muñoz-Escoí, P. Galdámez, and J. M. Bernabéu-Aubán. The NanOS cluster operating system. In R. Buyya, editor, *High Performance Cluster Computing*, volume 1, chapter 29, pages 682–702. Prentice-Hall PTR, Upper Saddle River, NJ, USA, 1999. 773
9. OMG. *The Common Object Request Broker: Architecture and Specification*. Object Management Group, July 1999. Revision 2.3. 773
10. R. Rajkumar, S. Fakhouri, and F. Jahanian. Processor group membership protocols: Specification, design and implementation. In *Proc. of the 12th IEEE Symposium on Reliable Distributed Systems, Princeton, NJ*, pages 2–11, October 1993. 781
11. A. Ricciardi and K. P. Birman. Consistent process membership in asynchronous environments. In K. P. Birman and R. van Renesse, editors, *Reliable Distributed Computing with the Isis Toolkit*, chapter 13, pages 237–262. IEEE Computer Society Press, Los Alamitos, CA, USA, 1994. 781
12. L. Rodrigues, P. Veríssimo, and J. Rufino. A low-level processor group membership protocol for LANs. In *Proc. of the 13th International Conference on Distributed Computing Systems*, pages 541–550, May 1993. 781