A HIERARCHICAL FAULT-TOLERANT RING PROTOCOL FOR DISTRIBUTED REAL-TIME SYSTEMS

TURHAN TUNALI*, KAYHAN ERCIYEŞ†, AND ZEHRA SOYSERT‡

Abstract. A synchronous communication protocol is designed and implemented for a distributed real-time system. The protocol operates on hierarchical rings and forms the communication backbone of a cluster based distributed system model that is designed to provide various distributed system functions such as clock synchronization, total event ordering, process group management, multicasting and fault tolerance. The multi-level hierarchy of the cluster based system model allows intracluster communication to be carried out independent of other clusters and hence provides parallelism in communication of cluster nodes. The intercluster communication among cluster *representatives* allows the *leader* to uniquely dictate various decisions to the whole system. The semantics of fault management for the cluster based model is also given and the interrelation of this fault mode with various layers of the system architecture is given by state transition diagrams. Finally, a distributed clock synchronization algorithm is implemented using the protocol and its performance is discussed.

Key words. clusters of processing nodes, fault tolerance, synchronous protocol, real-time systems

1. Introduction.

In distributed system applications, the implementation of fault tolerant techniques such as data replication increases the network traffic considerably, causing congestion and other complications. Various techniques have been developed to decrease the message complexity while providing high availability [1], [2]. The key concepts used in these techniques are the group communication protocols and multicasting. Among the important contributions to the field are Isis [3], Horus [4], Totem [5] and Transis [6] projects.These are all event-driven asynchronous distributed systems. Although these systems can be successfully used in applications such as transaction processing, banking and stock market trading and replicated database systems, their real-time capabilities are limited. Only Totem has features for real-time, however, the extended virtual synchrony model used is still based on an asynchronous model. In [7], based on asynchronous model, checkpoints are used to develop a fault-tolerant algorithm for distributed real-time systems.

On the other hand, synchronous models may yield better performance for real-time applications. For example, Delta-4 project uses synchronized clocks for the applications and logical clocks for multicast message ordering [8]. AAS is developed for air traffic control and uses a clock-driven and event-triggered approach [9]. MARS is developed for process control and uses a clock-driven and time-triggered approach [10], [11]. However, for geographically distributed systems, the situation is further complicated in synchronous models due to the requirement of a globally synchronized clock. Moreover, it is well known that, these models require high communication bandwidth to yield acceptable performance in real-time applications.

A distributed real-time system requires a real-time network which guarantees a bounded message delivery time, a distributed real-time operating system that manages resources of the whole system efficiently and timely as a single operating system

*International Computer Institute, Ege University, Bornova, İzmir 35100 Turkey (`tunali@ube.ege.edu.tr`).
†International Computer Institute, Ege University, Bornova, İzmir 35100 Turkey (`erciyes@ube.ege.edu.tr`).
‡International Computer Institute, Ege University, Bornova, İzmir 35100 Turkey (`soysert@ube.ege.edu.tr`).

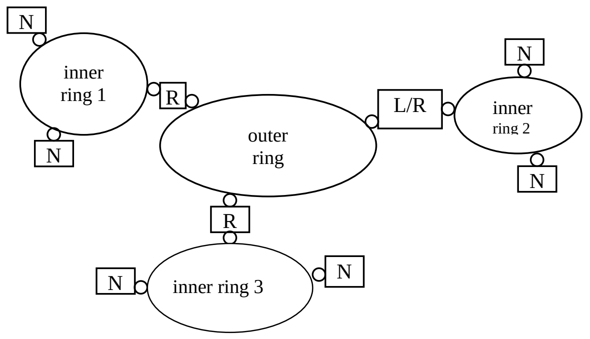

FIG. 2.1. *The two level communication hierarchy where L is the leader, R is a representative and N is an ordinary node*

and some "middleware" functionality to provide all this real-time platform to the distributed real-time applications. The aim of this study is to provide this "middleware" distributed real-time functionality with scalable and synchronous ring protocols embedded within each other. The reason for using such a protocol stems from the physical models of distributed real-time systems that are in fact clusters of processing nodes connected by some communication medium. As an example, a process control system can be considered where clusters may be mapped to processing cells. Providing fault-tolerant communication between a number of nodes running real-time applications is another aspect of our study.

In this work, a cluster based approach is applied to a synchronous system model with the expectation of reducing the network overheads while providing a suitable communication medium for real-time applications. This approach needs the development of a fault tolerance model that should be glued to the underlying communication protocol within the clusters and between the clusters. This fault tolerance model should not be confused with the fault tolerance concept at the application layer. The latter is entirely independent of fault tolerance at the communication layer and any popular scheme such as replication can be used at that layer. At the communication level, the difficulty arises due to the crash failure model accepted. To maintain the *intracluster* and *intercluster* communication, a model that contains *ordinary cluster node*, *cluster representative* and *leader* concepts is developed. The communication protocol is designed to run these three processes at appropriate levels. In case of failures, the protocol rebuilds the system and starts functioning again. Even though a two-level approach will be given here, the protocol can be implemented for more than two levels without loss of generality.

The paper is organized as follows: In Section 2, the distributed system model is given and its interrelation with communication is stressed. In Section 3, the ring based protocol is described together with state diagrams and algorithms. In Section 4, the implementation is explained and results on regular operation, fault scenarios and clock synchronization are discussed. Concluding remarks are given in Section 5.

2. The system model. In a sense, our protocol provides a communication infrastructure on top of which various layers can be built as in [12]. However, our protocol assumes a synchronous model. Our distributed real-time system model is formed from clusters of computing nodes. Depending on the represented layer in the architecture, the word "node" will be used both for processors and processes. The nodes within a cluster are assumed to be physically located close to each other as workstations in a local area network. Each cluster has a *representative* that communicates with the *leader*. The leader requests information from cluster representatives via the ring protocol operating at a higher level that only consists of the cluster representatives and the leader. Each representative, receiving this request, passes it to the nodes within their respective clusters. Within each cluster, the communication is also carried out by using the ring protocol, but this time, the protocol is used at a lower level consisting of the nodes of a cluster including the cluster representative. The main idea here is that, exactly the same protocol is used in these two different levels of communication. It should also be noted that clusters can easily join the system and this provides scalability. Figure 2.1 illustrates the communication hierarchy.

The application considered in this study that will use this model consists of various layers: The lowest level is the distributed clock synchronization. This is due to the assumption that the system model is synchronous. At this level, the cluster-based model is implemented with nodes being the processors. In clock synchronization algorithm, the clock leader requests clock information from the nodes and upon receipt of the clock information, the clock leader decides on the new clock value and dictates this to the whole system via the cluster representatives. This operation is repeated at each period and the clocks are synchronized. The clock synchronization module is implemented and the results are given in Section 4.

Above the clock level, there is the distributed scheduler whose operation is similar to that of the clock level as far as the cluster model is concerned. The nodes are again the processors, the clusters are the same clusters, however, the representatives and leader of this level are not necessarily the same nodes as that of the clock level. In other words, the cluster topology is considered to be fixed for all layers of the architecture, however, the formation of the representatives and the leader of this level is entirely independent of that of the clock level.

Above the scheduler is the group membership layer. The cluster-based model is again applied to this layer. The clusters are again defined by the original topology, but this time the nodes are the processes running on the processors of the respective clusters. The processes may belong to a particular group that is defined independent of the cluster topology. That is, for each process group, the cluster-based model is applied once to determine the group leader, cluster representatives of the group and the nodes in clusters belonging to the group.

The above outlined structure stresses that the cluster based model is repeatedly applied to the various levels of the distributed system architecture. In a sense, these architectural layers of clock synchronization, distributed scheduler and group management form one dimension of the overall model. The other dimension is the two level hierarchy of the cluster-based model that is repeatedly applied to the architectural layers. It is important to see that the two dimensions are entirely independent of each other. Figure 2.2 illustrates the two dimensions of the system model.

3. Protocol for cluster based model. The cluster based model needs special semantics to solve the fault tolerance problem at the communication layer. This section describes the protocol that contains fault tolerance semantics particularly developed for the cluster-based model. It is an essential feature of the whole system.

The general operation of the protocol is as follows: Every representative controls

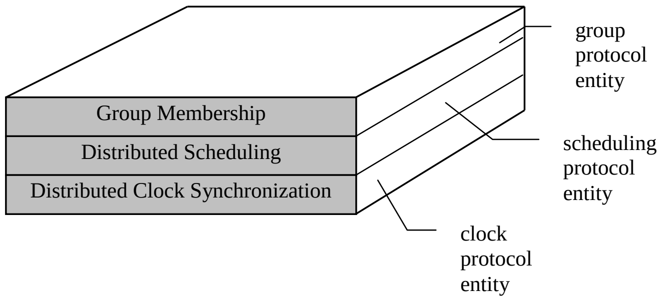

FIG. 2.2. *Two dimensions of the system model*

Control	Node R/O Info	Node Write Info

FIG. 3.1. *Structure of inframe*

the circulation of a message frame within its cluster. This message frame will be called *inframe* and the intracluster ring will be called *inner ring*. The inframe contains two main parts. The first part, called *global information*, contains the information that is sent by the leader such as the global clock value or ordered event list. This part is *read only* for the nodes and it dictates the leader's decision about the previous cycle. The second part is filled by the nodes. It contains node information such as the local clock value or local event list with time stamps, to be sent to the leader. The structure of inframe is shown in Figure 3.1.

Every node in a cluster knows the addresses of its predecessor, successor and representative in the cluster ring. If the predecessor (successor) of a node is the representative, then, that node also knows the address of the predecessor (successor) of the representative. The reason for this requirement is that when the representative crashes the other nodes should be able to repair the ring. All the nodes in all of the clusters know the address of the leader.

The leader controls the circulation of the message frame in the outer ring. This frame will be called the *outframe*. As in inframe, the outframe also contains two main sections. The first section contains the information that is sent by the leader such as the global clock value or ordered event list. This section is *read only* for the representatives. The second section is filled by the representatives. It contains the forwarded node information such as the local clock value or local event list with time stamps. The structure of outframe is shown in Figure 3.2. As in inner ring, every representative knows the address of its predecessor and successor in the outer ring. If the predecessor (successor) of a representative is the leader, then the representative also knows the address of the predecessor (successor) of the leader.

The protocol can be examined in three different modes of operations: NODE, REPRESENTATIVE, and LEADER. Figure 3.3 shows the structure of the control messages used in the algorithms.

When a NODE process is activated, it is furnished with the address of the cluster representative that it intends to join and the identification number of the cluster. It

Control	Rep R/O Info	Rep Write Info

FIG. 3.2. *Structure of outframe*

Message_Type	Message_Info_Slots (Optional)	Source	Destination

FIG. 3.3. *Structure of control messages*

submits a *join request* to the representative. Then it waits for *new neighbor info* that contains the addresses of a predecessor, successor and the leader together with some sequencing information. When it receives the new neighbor information, it updates its configuration and starts waiting for the *ring check inframe* which is a final check circulated by the cluster representative. Acknowledging the ring check inframe, the node starts waiting for the data inframe to fill up the local information. As the inframe arrives, it reads the first section that contains the globally finalized information. It then fills up the appropriate fields to report its local information and starts waiting for the next inframe. If the inframe is late, it sends point to point *are you alive* messages to its predecessor and successor. If its predecessor and/or successor is dead, it informs the representative and waits for a new information. The representative bypassing the dead node provides new predecessor or successor address to the node and the node acknowledging this, starts waiting for the inframe.

If the representative is dead, this is noticed by its neighbors, namely by its predecessor and successor. The successor of the dead representative becomes the new representative. Having the address of the dead representative's predecessor, the new representative repairs the ring and informs this predecessor. The node algorithm is given in Appendix A. Figure 3.4 shows the state diagram of the node algorithm.

When a REPRESENTATIVE process is activated, it is furnished with a predecessor and successor address on the cluster ring. It issues a *gather info inframe* to collect the cluster addresses and let the cluster nodes know that it is the new representative. The node algorithm assures that every node in the cluster has its predecessor and successor address before the new representative process is activated. The gather info inframe contains the address of the predecessor of the new representative that is to be collected by the successor of the new representative. This precaution is for a possible failure of the representative in future. By this way, the neighbor will have chance to bypass the dead representative in case of a failure. As the gather info circulates, the last node before the representative, that is the predecessor of the representative collects the address of the successor of the representative, again for ring repair purposes.

Upon receiving the gather info inframe, the nodes update their records, fill in their addresses and acknowledge. After making sure of the status of its nodes, the new representative sends a *join request message* to the leader. The new representative starts waiting for outer ring neighbor information from the leader. The leader, forming the new outer ring, furnishes the new representative with a predecessor address and successor address. The leader then circulates a *ring check outframe*. After acknowledging the ring check, the new representative is ready for routine operations.

The representative issues an inframe and waits. When it receives the Inframe, it waits for the outframe. As the outframe is received, it reads the global information, writes the local information and forwards the ouframe to its successor. It then issues the next inframe with the most recent global information to be forwarded. If the

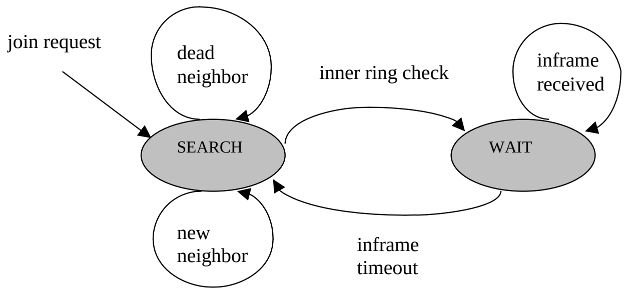

FIG. 3.4. *Node state diagram*

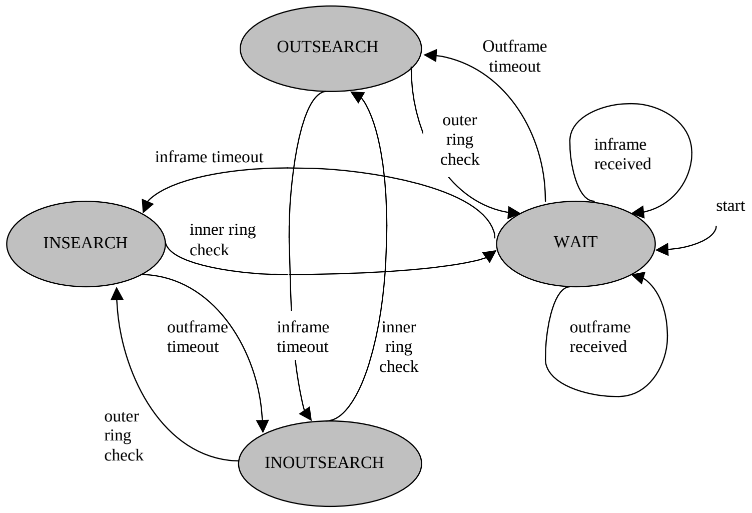

FIG. 3.5. *Representative state diagram*

outframe arrives before the inframe, the representative holds the outframe for a limited time. If inframe is still late, it releases the outframe with empty inner ring info information and *inner ring trouble* marked. If the inframe arrives before the outframe, the representative holds the new inframe to be issued for a limited time. If the outframe is still not in, the representative issues the new inframe with *outer ring trouble* marked.

If the inframe is late, it starts an *inner ring search* timer and sends *are you alive* to its inner ring neighboring nodes. If everyone is alive and the timer goes off, it issues a ring check inframe and upon the completion of the circulation of ring check, the representative issues inframe. If a node is dead, the representative discards the node from the ring and furnishes new addresses to the predecessor and successor of

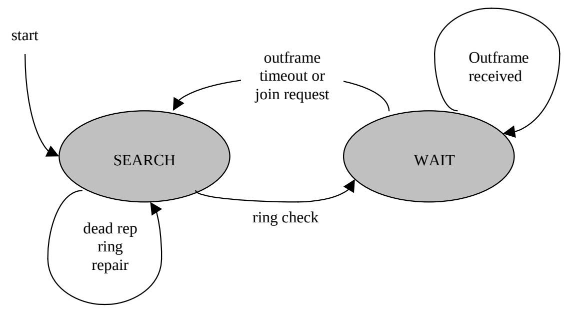

FIG. 3.6. *Leader state diagram*

the dead node and then circulates a ring check inframe. Then it retransmits the last inframe. If there is a join request from a node, the representative honors this only in its wait sate. It modifies the ring, informs the related nodes and retransmits the last inframe. The representative algorithm is given in Appendix B. The state diagram of the representative is given in Figure 3.5.

If the outframe is late, the representative sends point to point *are you alive* messages to its outer ring predecessor and successor. If either of its predecessor or successor is dead, it informs the leader and waits for a new outer ring neighbor information. The leader bypassing the dead representative provides new predecessor or successor address to the representative and the representative acknowledging this, starts waiting for the outframe.

If the leader is dead, its successor takes over. Having the address of the dead leader's predecessor, the new leader repairs the outer ring and informs this predecessor. It should be noted that the representative acts similar to a leader in the inner ring and similar to a node in the outer ring.

When the leader process is activated, it issues a *gather info outframe*. Upon receiving the gather info outframe, the representatives update their records, fill in their addresses and acknowledge. As in inner ring, necessary measures are taken to bypass the leader in case of failure. They also update the leader address field in their inframes. After making sure of the status of its representatives, the new leader is ready for routine operations. It sorts the collected local information and issues an outframe containing the global information.

If the outframe is late, the leader starts an *outer ring search* timer and sends *are you alive* to its predecessor and successor. If everyone is alive and the timer goes off, it issues a ring check. If a representative is dead, it discards the representative from the outer ring and furnishes new addresses to the predecessor and successor of the dead representative. Then it retransmits the last outframe. If there is a new representative message, the leader modifies the ring, informs the related representatives and retransmits the last outframe. The leader algorithm is given in Appendix C. Figure 3.6 shows the state diagram for the leader process.

The protocol assumes that, if a node crashes, the representative does not crash

TABLE 4.1

Average ring cycle completion times of the first topology in msec.

Message size	Cluster1	Cluster2	Cluster3	Outer Ring
1K	5.254	13.696	3.003	19.456
2K	9.821	19.195	6.803	28.225
4K	14.179	25.132	10.872	43.421
8K	24.725	39.341	20.930	62.665

TABLE 4.2

Average ring cycle completion times of the second topology in msec.

Message size	Cluster1	Cluster2	Cluster3	Outer Ring
464 bytes	3.906	4.880	5.856	9.760

during the ring repair period which is measured to take less than 50 milliseconds in our implementation. Multiple node crashes are allowed. However, the protocol again assumes that, if a representative crashes, no other node crashes during the new representative startup process. This means that, if an event violating the above assumptions happens within a cluster, then that cluster will be discarded from the whole system and the rest of the system will continue functioning.

The above assumptions are also valid for the operation of the outer ring. In case of leader and a neighbor representative failing at the same time, the whole system collapses and has to be restarted. However, the probability of such an event is quite low since different clusters usually reside in different LAN segments.

Note that the deterministic nature of the protocol is suitable for supporting real-time applications. The synchronous behavior can easily be exploited to provide necessary services for applications. In the following section, implementation of clock synchronization will be reported.

4. Implementation. The protocol is implemented on a network of workstations. In the first part of our experiments, the network consisted of three clusters, each having three nodes with one of the clusters having two of the nodes on the same machine. All nodes were DEC Alpha workstations that are located in Ege University campus and running OSF1/Mach operating system. The clusters were all on different 10 Mbps Ethernet segments. The outer ring was 100 Mbps FDDI. Initially, the implementation was limited to the routine operation of the synchronization of inframe and outframe. The frame synchronization rate achieved is an essential measure for the performance of the protocol when no crash occurs. Therefore, the maximal rate achieved at this level gives a good idea about the cycle period both for the inner ring and outer ring. Four different sizes are used for the messages: 1K, 2K, 4K and 8K. The outframe issue period was taken as 100 msec. Table 4.1 shows the average completion times of ring cycles in milliseconds based on 30 observations. As it can be seen from the table, outframe issue period is well above the cluster periods to guarantee that inframes are ready waiting for the outframe. Representatives issued their inframes as they received outframes.

In the second part, we have implemented the system on a similar topology as above, but this time two of the clusters contained four nodes each and the third

TABLE 4.3

Average crash recovery times in msec.

Crash scenario	Recovery	Average crash recovery time
Single node crash in a cluster	Recovery of cluster	22.590
Simultaneous two node crashes in a cluster	Recovery of cluster	25.725
Total collapse of a cluster	Recovery of the outer ring	31.784
Single representative crash	Recovery of the whole system	55.867

TABLE 4.4

Average clock drifts measured at nodes in msec.

Clock scenario	Cluster1	Cluster2	Cluster3
Uncorrected Clock Drifts (5 sec.)	5.963 7.940 7.219 6.832	6.242 5.672 8.450 8.324	5.231 5.734 6.675
Uncorrected Clock Drifts (10 sec.)	8.439 8.963 8.231 7.171	7.903 6.356 8.909 8.654	6.434 7.423 8.284
Uncorrected Clock Drifts (20 sec.)	10.524 9.520 10.359 9.036	10.939 9.632 9.158 9.236	7.564 10.345 9.453
Corrected Clock Drifts (frame issue period=200msec.	2.089 1.902 2.187 1.980	2.172 1.883 1.658 2.012	1.765 1.806 2.196

cluster contained three nodes. All the nodes were on different machines. With a fixed message payload of 464 bytes, the average ring cycle completion times are given in Table 4.2. The measurements are based on 20 observations. Even though the second topology contains more nodes, when compared with Table 4.1, Table 4.2 figures are much more improved. We have continued our experiments with the second topology in the following.

For the crash scenarios, *are you alive* timer is set to 5 msec. *Issue Inframe* and *release outframe* timers are set to 5 msec. *Inframe* and *outframe* timers are set to 200 msec. Table 4.3 lists average recovery times of various scenarios in msec. The averaging is done on 20 observations. The crash figures show that, with 100msec frame issue period, the system solves the problem within the issue period and the normal operation is not interrupted.

Using the second topology, a simple clock synchronization algorithm is implemented. The local clock values are collected from nodes and forwarded to the leader. During the forwarding process, a fixed correction term per hop is used to account for the time spent during the communication. The leader simply takes the average and dictates it as the global clock value. The representatives forward this global clock value to their nodes. Both the representatives and nodes normalize the value of clock for the time passed during communication. The frame issue period used in this implementation is 200 msec. The drifts in the local clock values are measured and the results are given in Table 4.4. The results are averaged on 20 observations. The minimum drift is measured as 0.048 msec. and the maximum drift is measured as 5.760 msec. Table 4.4 also contains clock drifts in 5,10 and 20 sec. ranges when no correction is done. As can be seen from the table, the clock values drastically improve when the algorithm is implemented.

5. Conclusion. A cluster-based distributed system model is considered for a ring-based hierarchical communication protocol developed together with its imbedded fault-tolerant cluster membership algorithm. The cluster concept is considered on a hardware basis such as the computers on the same Ethernet segment. The fault-tolerance algorithm developed allows the protocol to maintain communication in case of crash failures. The synchronous nature of the communication algorithm can easily be exploited for real-time applications. As an application, a clock synchronization algorithm is implemented using our protocol. Crash scenarios are tested and performance is reported.

The current work is progressing in two steps: Firstly, managing processing power using the ring protocol is being implemented. In this case, each representative periodically monitors availability of the processors in its cluster. When a node receives an asynchronous real-time processing request, it checks whether this can be fulfilled locally. If not, the representative is informed which may or may not find a potential receiver in the cluster to process the request. If the request can not be met within the cluster, it is forwarded to the outer ring to find a possible receiver in another cluster. The shadow process concept is used in this case to eliminate the need for process migration. The second implementation involves using process groups and active replication to provide fault tolerance to the application. To provide such a feature, the order of messages sent to copies of a process must be preserved. The inner and outer ring protocols function similarly in this case where events are gathered locally in each cluster, delivered to the leader, which determines the order and dictates the result to the nodes which contain the replicated process.

REFERENCES

[1] D. Malki, *Multicast Communication for High Availability*, Ph. D. Dissertation, Hebrew University in Jerulsalem (1994).
[2] Y. Amir, *Replication Using Group Communication Over a Partitioned Network*, Ph. D. Dissertation, Hebrew University in Jerusalem (1995).
[3] K.P. Birman and R. Cooper, *The ISIS Project: Real Experience with Fault-Tolerant Programming System*, Technical Report, Department of Computer Science, Cornell University (1993).
[4] R. Van Renesse, K.P. Birman and S. Maffeis, *Horus: A Flexible Group Communication System*, Communications of the ACM, Special Section on Group Communication, 39-4, (1996).
[5] L.E. Moser et.al., *Totem: A Fault Tolerant Multicast Group Communication System*, Communications of the ACM, Special Section on Group Communication, 39-4 (1996).
[6] D. Malki and D. Dolev, *The Transis Approach to High Availability Cluster Communication*, Communications of the ACM, Special Section on Group Communication, 39-4 (1996).

[7] Z.M. Wojcik and B.E. Wojcik, *Optimal Algorithm for Real-Time Fault Tolerant Distributed Processing Using Checkpoints*, Informatica, Vol.19, (1995), pp. 111–122.
[8] S. Mishra and R. Schlichting, *Abstractions for Constructing Dependable Distributed Systems*, Technical Report, TR92-19, Department of Computer Science, University of Arizona (1992).
[9] F. Cristian, *Understanding Fault-Tolerant Distributed Systems*, Technical Report, Department of Computer Science, University of California, San Diego (1993).
[10] H. Kopetz, et. al., *Distributed Fault-Tolerant Real-Time System: The MARS Approach*, IEEE Micro, Vol.9, (1989), pp. 25–40.
[11] S. Mullender, *Distributed Systems*, ACM Press, Second Edition.
[12] O. Babaoglu, et. al., *RELACS: A Communications Infrastructure for Constructing Reliable Applications in Large-Scale Distributed Systems*,Technical Report-UBLCS-94-15, Laboratory for Computer Science, University of Bologna, Italy (1995).

Appendix A. Node algorithm.

```
Node_process
{
send_message(join_request, my_node, repr);
wait_for_neighbor_info;
curstate=SEARCH;
cycle
  wait_for_event;
  switch (event_type);
    RING_CHECK_INFRAME_RECEIVED:
      mark_ack_slot;
      release_ring_check_inframe;
      start_timer(inframe);
      if (curstate=SEARCH)
        curstate=WAIT;
    INFRAME_RECEIVED :
      if (curstate=WAIT)
        if (not_outer_ring_trouble)
          read_global_info;
          write_local_info;
        release_inframe;
        start_timer(inframe);
    INFRAME_TIMEOUT:
      curstate=SEARCH;
      send_message(are_you_alive,my_node,pred);
      send_message(are_you_alive,my_node,successor);
      start_timer(are_you_alive);
    AREYOUALIVE_TIMEOUT:
      if (my_pred=rep and my_pred=dead)
        send_message (I_am_your_new_successor,my_node,pred_of_dead_rep);
        wakeup(Representative_process);
      else
        send_message (dead_neighbor,dead_neighbor_addr,my_node,repr);
    GATHER_RING_INFO_INFRAME_RECEIVED:
      if (curstate=WAIT)
        curstate=SEARCH;
        stop_timer(inframe);
      update (rep_info);
      mark_address_slot;
      release_inframe;
    NEW_NEIGHBOR_MESSAGE_RECEIVED:
      if (curstate=WAIT)
        stop_timer (inframe);
        curstate=SEARCH;
      update (neighbor_info);
    ARE_YOU_ALIVE_MESSAGE_RECEIVED:
      if (curstate=WAIT)
        curstate=SEARCH;
      send_message (I_am_alive,my_node,asking_neighbor);
      send_message(are_you_alive,my_node,other_neighbor);
endcycle
}
```

Appendix B. Representative algorithm.

```
Representative_process
{
issue_inframe(gather_ring_info,my_addr,my_successor,ring_addr_slots);
wait_for_gather_ring_info;
issue_inframe(ring_check);
wait_for_ring_check_ack;
send_message(cluster_join_request,my_node,leader);
wait_for_new_outer_neighbor_info;
wait_for_ring_check_outframe;
start_timer(outframe);
issue_inframe(data,global_info_slots,local_info_slots);
start_timer(inframe);
curstate=WAIT;
cycle
  wait_for_event;
  switch (event_type);
    INFRAME_RECEIVED :
      stop_timer(inframe);
      read_local_info;
      if (outframe_waiting)
        update(cluster_info_slot);
        release_outframe;
        start_timer(outframe);
        issue_inframe (data,glob_info_slots,local_info_slots);
        start_timer(inframe);
      start_timer(issue_inframe);
    ISSUE_INFRAME_TIMEOUT:
      issue_inframe (outer_ring_trouble);
    OUTFRAME_RECEIVED :
      stop_timer(outframe);
      read_global _info;
      if(curstate=WAIT)
        if (inframe_in)
          update(cluster_info_slot);
          release_outframe;
          start_timer(outframe);
          issue_inframe (data,glob_info,local_info_slots);
          start_timer(inframe);
        else
          start_timer(release_outframe);
      if(curstate=INSEARCH)
        mark (inner_ring_trouble);
        release_ouframe;
        start_timer(outframe);
    RELEASE_OUTFRAME_TIMEOUT:
      set (inner_ring_trouble);
      release_ouframe;
      start_timer(outframe);
    INFRAME_TIMEOUT:
      if(curstate=WAIT)
        curstate=INSEARCH;
      else if(curstate=OUTSEARCH)
        curstate=INOUTSEARCH;
      start_timer(insearch);
      send_message (are_you_alive,my_node,in_pred);
      send_message(are_you_alive,my_node,in_successor);
```

```
   start_timer(in_are_you_alive);
IN_ARE_YOU_ALIVE_TIMEOUT:
   stop_timer(in_search);
   if (pred=dead)
      send_message (new_succes,my_node_addr,my_succes_addr,my_node,dead_node's_pred);
   else
      send_message (new_pred,my_node_addr,my_pred_addr,my_node,dead_node's_succes);
   issue_inframe(ring_check);
INSEARCH_TIMEOUT:
   issue_inframe(ring_check);
INNER_RING_CHECK_RECEIVED:
   if(curstate=INSEARCH)
      curstate=WAIT;
      issue_inframe(data,glob_info,local_info_slots);
      start_timer(inframe);
   else
      curstate=OUTSEARCH;
      issue_inframe (outer_ring_trouble,glob_info_slots,local_info_slots);
      start_timer(inframe);
OUTER_RING_CHECK_RECEIVED:
   if(curstate=OUTSEARCH)
      curstate=WAIT;
      start_timer(outframe);
   else
      curstate=INSEARCH;
      start_timer(outframe);
OUTER_RING_CHECK_RECEIVED:
   if(curstate=OUTSEARCH)
      curstate=WAIT;
      start_timer(outframe);
   else if (curstate=INOUTSEARCH)
      curstate=INSEARCH;
      start_timer (outframe);
DEAD_NEIGHBOR_MESSAGE_RECEIVED:
   stop_timer(in_search);
   if (dead_node=my_successor)
      send_message (rep's_new_successor,dead_node's_successor,my_node,my_in_pred);
   send_message (new_pred,dead_node_pred_addr,my_node,dead_node's_successor);
   send_message (new_success,dead_node_succ_addr,my_node,dead_node's_pred);
   issue_ inframe(ring_check);
OUTFRAME_TIMEOUT:
   if(curstate=WAIT)
      curstate=OUTSEARCH;
   else if (curstate=INSEARCH)
      curstate=INOUTSEARCH;
   send_message(out_are_you_alive,my_node,out_pred);
   send_message(out_are_you_alive,my_node,out_successor);
   start_timer(out_are_you_alive);
OUT_ARE_YOU_ALIVE_TIMEOUT:
   if (my_out_pred=leader and my_out_pred=dead)
      send_message (I_am_your_new_out_successor,my_node,pred_of_dead_leader);
      wakeup (Leader_process);
   else
      send_message (dead_neighbor,dead_neighbor's_addr,my_node,leader);
GATHER_RING_INFO_OUTFRAME_RECEIVED:
   update (leader_info);
   if (curstate=WAIT)
      curstate=OUTSEARCH;
   else if (curstate=INSEARCH)
      curstate=INOUTSEARCH;
   stop_timer(outframe);
   mark_address_slot;
```

```
            release_outframe;
        NEW_NODE_JOIN_REQUEST_RECEIVED:
            curstate=INSEARCH;
            send_message (new_successor,new_node's_addr,my_node,in_pred);
            send_message (rep's_pred,new_node's_addr,my_node,in_successor);
            send_message (nghbor_info,pred_addr,succes_addr,rep's_succes,my_node,new_node);
            issue_inframe(ring_check);
endcycle
}
```

Appendix C. Leader algorithm.

```
Leader_process
{
issue_inframe(gather_ring_info,my_addr,my_successor,ring_addr_slots,);
wait_for_gather_ring_info;
issue_outframe(ring_check);
wait_for_ring_check_ack;
issue_outframe(data,last_glob_info,local_info_slots);
start_timer(outframe);
cycle
    wait_for_event;
    switch (event_type);
        OUTFRAME_RECEIVED :
            read_local_info_slots;
            form_global_info;
            issue_outframe(data,glob_info,local_info_slots);
            start_timer(outframe);
        OUTFRAME_TIMEOUT:
            curstate=SEARCH;
            start_timer (ring_search);
            send_message(are_you_alive,my_node,pred);
            send_message(are_you_alive,my_node,successor);
            start_timer(are_you_alive);
        ARE_YOU_ALIVE_TIMEOUT:
            stop_timer (ring_search);
            if (pred=dead)
                send_message (new_succesor,my_addr,my_successor's_addr,my_node,dead_rep's_pred);
            else
                send_message (new_pred,my_addr,my_pred's_addr,my_node,dead_rep's_successor);
            issue_ouframe(ring_check);
        DEAD_NEIGHBOR_MESSAGE_RECEIVED:
            stop_timer (ring_search);
            if (dead_neigh=my_successor)
                send_message (leader's_new_successor,dead_neighbor's_successor,my_node,my_pred);
            send_message (new_pred,dead_rep's_pred_addr,my_node,dead_rep's_successor);
            send_message (new_successor,dead_rep's_successor_addr,my_node,dead_rep's_pred);
            issue_ouframe(ring_check);
        RING_SEARCH_TIMEOUT:
            issue_ouframe(ring_check);
        RING_CHECK_RECEIVED:
            curstate=WAIT;
            issue_outframe(data);
            start_timer(outframe);
        NEW_REP_JOIN_REQUEST_RECEIVED:
            curstate=SEARCH;
            send_message (new_successor,new_rep's_addr,my_node,my_pred);
            send_message (leader's_pred,new_rep's_addr,my_node,my_successor);
            send_message (nghbor_info,pred's_addr,succes's_addr,leader's_succes,my_node,new_rep);
            issue_outframe(ring_check);
endcycle
}
```