A CLUSTER BASED MODEL FOR RESOURCE MANAGEMENT AND FAULT TOLERANCE IN DISTRIBUTED REAL-TIME SYSTEMS

Kayhan Erciyeş, Oznur Ozkasap, Turhan Tunali
e-mail:erciyes@ube.ege.edu.tr, ozkasap@bornova.ege.edu.tr, tunali@ube.ege.edu.tr

Abstract

We propose a distributed system model based on clusters of processors and processes to manage the resources and provide fault tolerance in a distributed real-time system. At the lowest level, time is managed in a distributed clock synchronization module by a leader process that provides agreement on a common clock value among the representatives of the clusters which then dictate this value to the individual members of each cluster. For fault tolerance in such a system, process groups are used to manage replicates of processes using a similar concept where the ordering of the events is managed by the leader and representatives of process groups of each cluster which then dictate this ordering to every process group member in each cluster. Finally, the static load balancing is performed in two phases; first the real-time tasks are allocated to clusters of processors and then scheduled within individual clusters to meet deadline constraints

Keywords: cluster based model, distributed clock synchronization, process groups, fault tolerance. load balancing

1 Introduction

We propose a cluster based model to manage resources and provide fault tolerance in a distributed real-time system. The clusters consist of computing nodes that are physically located close to each other such as workstations of a local area network.
Firstly, time is managed by a distributed clock synchronization algorithm where *representative nodes* of clusters communicate periodically to decide on a common clock value under the supervision of a central synchronizer called the *leader*. Then, these values are distributed by the representative nodes to all of the nodes in clusters. We provide fault management facilities for *representative*, *leader* or *node* process crashes. The second level of the proposed architecture aims at fault tolerance by the employment of process groups. The threads of a periodic task are included in the *task thread group* and the replicates of a task thread are the members of the *replicate thread group*. The ordering of events is accomplished at each group whether this is thread or replica group using the group cluster representatives and the central synchronizer as in the distributed clock synchronization above.

The third level of the system provides load balancing for the static tasks of the real-time system using the same model where a task set is partitioned by a suitable *Cluster Algorithm* to be allocated to processor clusters. In the second phase of the load balancing, the tasks of the set are allocated to individual processors of the cluster by the *Task-to-Processor Mapping Algorithm* as in [1]. Various heuristics are used to obtain suboptimal but less costly allocations as in [3].

A separate fault tolerance method is also provided to be able to survive node, representative and leader process failures. This scheme is independent of the layer of the distributed real-time system under consideration whether this is time management, process group management or load balancing. It provides generic state transition rules, elections for leader and representatives when they fail.

The background is given in Section 2 and important related work is presented in Section 3. The model is detailed in Section 4. The cluster fault model which is independent of the management layer in consideration is explained in Section 5 . Section 6 describes the distributed clock synchronization module. The process groups and the ordering of events in this model are discussed in Section 7. The implementation results for the clock synchronization module are given in section 8 and the current status of the project and future plans along with conclusions are given in Section 9.

2 Background

2.1 Clock Synchronization in Real-Time Systems

There are mainly two approaches for implementing a fault-tolerant time service in a distributed system. In the first one, local hardware clocks are synchronized at regular intervals in such a way that the clocks remain within some maximum distance of each other. In the second approach, a logical clock is constructed that causally orders events. Thus, the temporal order in which different events occur in the system is derived without direct association to a hardware clock value. The disadvantage of a logical clock is that, it does not provide a mapping from the timing of an event to real time, whereas synchronized clocks may provide this mapping by synchronizing with an external time source [19,20].

Each node in a distributed real-time system contains a local hardware clock of quartz accuracy. In such a system, it is a necessity to provide a common time reference of specified accuracy for an application. This common time reference which is called approximate global time, is generated by the application of a clock synchronization algorithm [20]. This provides a way to measure the time of occurrence of events and the duration of intervals between events in the system.

There are two levels of clock synchronization in distributed systems, namely, internal and external. Internal clock synchronization is the construction of an approximate global time base among the ensemble of nodes of the distributed system. That is, local clocks are always kept within a certain maximum drift of one another. In external clock synchronization, the local clocks are always kept within certain maximum deviation from an external time reference [19].

During the design of a clock synchronization algorithm, certain assumptions about propagation delays, processing delays and failure modes should be made. The algorithms proposed in literature differ from each other in their assumptions about the clocks, network topology and failure hypothesis. Generally, the mechanism involves periodically exchanging information about local clock values and then computing a correction term and applying it to the local clock.

Most of the approaches for clock synchronization use a convergence function that guarantees the properties of monotonocity, precision and interval preservation. Different approaches for constructing clock synchronization algorithms work in the following ways [19,21,22,23]: One class of approaches consists of first exchanging local clock values and then applying a fault-tolerant averaging function to these values to compute a new clock value. These type of algorithms require a fully connected network, a known upper bound on message delay, and initial synchronization of the clocks. In another class of approaches, the clock values of various processors are first obtained through a protocol that guarantees an agreement among all correct processors on a vector of values, one from each clock. Then, each processor applies the same averaging function to compute a new clock value. The agreement process is used to tolerate failures. These type of algorithms require a bound on message delay and a limit on the maximum number of processes that may fail. Another class of approaches uses a synchronizer to synchronize clocks. The synchronizer process collects values from all local clocks and then propagates these values to all processes, which use them to compute a new clock value. To avoid the problem of single point failure, every process in the system periodically attempts to become the synchronizer at preset time intervals. These type of algorithms require a bound on message delay and initial synchronization of the clocks. All of the approaches described above assume a synchronous network, that is message delays are bounded.

2.2 Synchrony and Failure Model

A component in a system is synchronous if it always performs its intended function within a finite and known time bound, and asynchronous otherwise. Synchrony can be defined for communication links, processors, networks and protocols. A more formal definition of the synchronous system is given below:

Definition: A system is defined to be synchronous if it satisfies the following properties:

There is a known upper bound a on message delay, which consists of the time it takes for sending, transporting and receiving a message over a link. Every process p has a local clock Cp with known bounded rate of drift $\beta \geq 0$ with respect to real-time. That is, for all p and all $t > t'$,

$$(1+\beta)^{-1} \leq (C_p(t) - C_p(t'))/(t-t') \leq (1+\beta)$$

where $C_p(t)$ is the value of C_p at real-time t. There are known upper bounds on the time required by a process to execute a step.

A protocol is defined to be synchronous if the time to perform the sequence of events required by the protocol is known and bounded. Two different approaches have been taken during the design of synchronous protocols, namely, clock-driven and clockless. A clock-driven protocol relies on a common global time base constructed using a clock synchronization technique. Most of these type of protocols are periodic in nature, taking certain actions based on the time read from a common clock.

Clock-driven protocols are based on the assumption of synchronous processors and communication links. A clockless protocol, on the other hand, does not rely on the existence of synchronized clocks. Also, this type of protocol does not assume synchronous processors or communication. This leads to a different system style, in which protocols approximate processor and communication synchrony by using techniques such as timeouts and acknowledgements to put a bound on delays [19]. This is an inefficient way for realizing protocols for real-time applications, since it increases the number of messages used by the protocol.

Another important issue is a failure model which is necessary for precisely specifying assumptions on how a component in a distributed system behaves when it fails. When the specification of a component's acceptable behavior is available, it provides a standard against which the behavior can be judged. Failure model is both considered for processes and communication links. Several types of failure models have been studied in literature [19,26], such as crash, omission, timing and arbitrary (Byzantine) failures.

3 Related Work

3.1 Systems Providing High-Level Reliable Communication Protocols

There are several systems providing high-level communication protocols for distributed applications. In this section, properties of some of the systems are discussed.

Isis is a system for building fault-tolerant applications consisting of cooperating, distributed processes. The system provides two basic building blocks, namely group management and group communication. A design strategy such as that used in microkernel based operating systems is applicable, where a core system provides the basic functionality, on which more complicated services can be built [4].

Isis system is multi-threaded. For each arriving message, a new thread is created. The system was one of the first available system under the UNIX system that provided multi-threading. Furthermore, virtual synchrony model was introduced by Isis system.

Isis process groups have the following properties: Processes can subscribe to groups and groups may be created dynamically. Processes may send messages to groups, which will be reliably delivered to all subscribing processes. Causality among messages are guaranteed. Isis supports a way to track membership in groups, that is, all subscribers see exactly the same history of membership changes. In addition, it is possible to preclude processes from subscribing or posting to groups. This simplifies the development of secure, fault-tolerant tools and applications.

Available protocols used by the Isis system are IP, UDP, Deering multicast and raw Ethernet. MUTS (Multicast Transport Service) layer provides a simple, reliable multicast service that runs on one or more lower level protocols, and applies a simple sliding window protocol which is extended to handle multicast communication. CBCAST (Causal Broadcast) layer implements group membership protocols and guarantees causal ordering of messages.

Isis provides a set of tools such as RPC, atomic transactions, primary-backup replication, monitoring and control facilities, and resource management [16,6].

The Horus project seeks to develop a communication system addressing the requirements of a wide variety of distributed applications. Compared to its parent system Isis, Horus is smaller, provides more flexibility, and performs better. The system implements the group communications model providing unreliable or reliable FIFO, causal or total group multicasts. It is extensively layered and highly reconfigurable allowing applications to only pay for services they use. Horus has few system dependencies, and can be incorporated in modern distributed operating systems as either a user-level service or kernel-level subsystem, or both. Horus also offers security features, and is able to deal with network partitioning.

Horus system provides the following properties, which are group and point-to-point communication layered communication semantics, support for multiple transport protocols, threads, security model and failure consistency [10,12,13,14,15].

Transis is a transport level group communication service that simplifies the development of fault-tolerant distributed applications. The system distinguishes itself in allowing partitionable operation, in which multiple network components that are temporarily disconnected from each other operate autonomously, and in supporting consistent merging upon recovery. It guarantees reliable message delivery coupled with consistent failure notification. Transis contains a reliable broadcast service that delivers messages to a group of processes, while preserving their relative orders.

Transis system was designed with the following goals: It tackles network partitions and provides tools for recovery from them. It meets the needs of a large network through a hierarchical communication structure, with gateways selectively filtering messages among domains. It exploits available network multicast within each local area network and provides fast cluster communication [8,18].

The Totem System provides reliable, totally ordered muticasting of messages over local area networks and exploits the hardware broadcasts of such networks to achieve high performance. The system uses extended virtual synchrony model. Totem is intended for complex applications in which both fault-tolerance and real-time performance are critical. Such applications are typically built as asynchronous event-driven distributed systems, such as air traffic control, industrial automation, transaction processing, banking and stock market trading, intelligent highways, medical monitoring and replicated database systems.

The main characteristics of the Totem system are as follows: high throughput and low predictable latency; rapid detection of recovery from faults; systemwide total ordering of messages, even for systems in which the network can partition and remerge, and for systems in which process groups can intersect; scalability to larger systems based on multiple local area networks interconnected by gateways, within the same geographical area [7].

Group communication and management aspects of our proposed model has some similarities with the Totem system. For example, both systems use a token-based approach to totally order multicast messages. Totem exploits tokens for both intra-LAN and inter-LAN message ordering. On the other hand, we use tokens for intercluster event ordering and for intracluster event ordering a representative process in each cluster uses a diffusion based approach. LANs connected by gateways resembles processor clusters in our model and, we do not consider

WANs since we intend to support distributed real-time applications. The main differences are as follows: Our model assumes a synchronous system, and we use our cluster-based distributed clock synchronization to synchronize nodes in the distributed system. Since real-time applications are considered, modules works in a periodic manner.

Rampart system is a research prototype which is a toolkit of protocols designed to simplify the task of coordinating correctly operating nodes in a dynamic environment characterized by simple and arbitrary node failures, node recoveries, and communication failures [9].

3.2 Real-Time Group Communication Systems

There are some systems studied in literature which have real-time group communication aspects. The system assumptions of these systems and their main characteristics are given in this section. All of the systems discussed are based on a synchronous system model.

The goal of Delta-4 project is to define a general system architecture for dependable distributed systems. The applications targeted by the project include process control, computer integrated manufacturing, distributed databases, transaction processing and scientific computation. XPA (Extra Performance Architecture) is one the architectures designed explicitly for constructing fault-tolerant distributed programs that also have critical real-time requirements. The failure model for nodes and communication links is crash failures. XAMP (Extended Atomic Multicast Protocol) used by XPA provide services ranging from unreliable datagrams to reliable multicast and full atomic multicast. Two time services are provided, one using synchronized clocks and the other using logical clocks. The former is intended primarily for use by applications, while the latter is used in the multicast message ordering. When considered its multicasting facility, Delta-4 is based on a clockless and event-triggered approach. Event-triggered protocols initiate a transmission at the moment they receive a send request [19,24].

AAS (Advanced Automation System) is fault-tolerant distributed system designed for air-traffic control. It uses abstractions such as a time service, multicast and membership. The time service is provided by synchronizing clocks using probabilistic scheme. The multicast service provides atomicity, consistent total order and termination. AAS is based on a clock-driven and event-triggered approach [2,24,25].

MARS (MAintainable Real-time System) is a system used for distributed process control applications. In these type of applications, the control system must meet stringent real-time guarantees, in addition to being functionally correct and fault-tolerant. The failure model corresponds to crash failures. The time service uses a hardware clock synchronization units. The multicast service is based on a TDMA (Time-Division Multiple Access) approach in which the bus is conceptually divided into slots along the time dimension and machines are allocated time slots using a round-robin scheduling algorithm. The membership service is implemented by logically including a machine's current membership set in every message. MARS is based on a clock-driven and time-triggered approach. Time-triggered protocols transmit at prespecified periodic instants, so they should be clock-driven [24].

4 System Model

This section gives information on the proposed system model for the distributed real-time group communication system, more specifically, assumptions behind synchrony, failure model, programming paradigm, and an explanation on layered architecture of the system.

4.1 Synchrony, failure model and programming paradigm

The decision on synchrony model for our system is related to assumptions made about the time bound on components such as processes and communication links in a distributed system. We

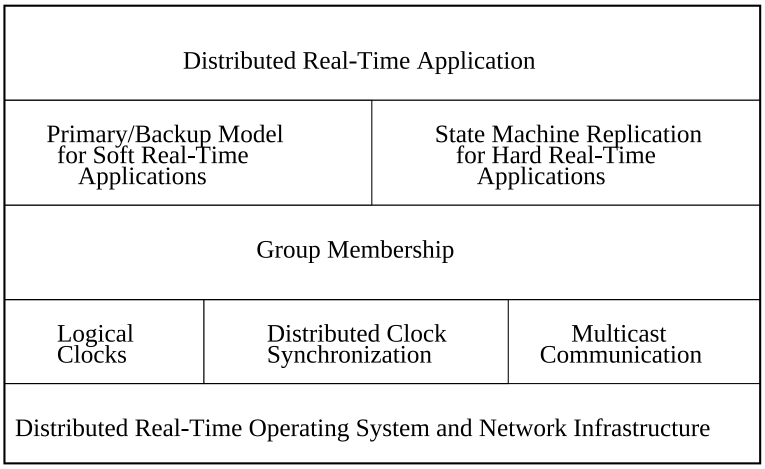

Figure 1: Layered Architecture

assume synchronous system model, since we intend to integrate real-time distributed system requirements with group communication mechanisms.

In the light of concepts discussed above, our system is said to be based on a clock-driven approach.

We assume crash failure model for processes and omission failure model for communication links. This is a realistic and the most common way for modeling today's distributed systems in terms of fault-tolerance.

We assume state machine approach as the programming paradigm of the fault-tolerant distributed system. In this approach, an application is structured as a collection of services that are implemented by multiple processes for the purpose of fault-tolerance. Then, each service is characterized as a state machine, which maintains state variables that are modified in response to commands received from other state machines or environment. Execution of a command is deterministic and atomic with respect to other commands. For fault-tolerance purposes, a state machine is implemented by replicating it and running each replica in parallel in the distributed system. This is also called active replication [19].

4.2 System Architecture

Our proposed layered architecture for a real-time group communication model is given in Fig. 1. The bottom layer of the system is the distributed real-time operating system and network support. Then comes the logical clocks, distributed clock synchronization protocol and multicast communication. Logical clocks are used for event ordering in process groups. Multicast layer provides reliable, atomic and ordered multicast communication primitives. Logical clocks, distributed clock synchronization and multicasting have a close interaction, that is, one module can use the services provided by another one.

Group membership mechanism includes methods for creating, deleting groups, joining a process to a group, and leaving a process from a group. New members may join the group as the number of requests on the service increases or as idle machines volunteer their cycles for the service. Members may leave the group as the load on the service decreases or when a machine crashes. It is useful if a user of such a service can use the process group name to communicate with the service without needing to know the membership of the group or where the members are located. An important issue associated with group membership is that; if a member crashes, it leaves the group. The problem is that the other members may not be aware of this fact and they have to discover this experimentally by noticing that the crashed member no longer responds to anything. Another issue is that joining to and leaving from a group have to be synchronous with messages being sent. When a process joins a group, it must receive all messages sent to that group. Also, as soon as a process leaves a group, it must not receive any more messages from the group, and the other members must not receive any more messages from it. In addition, consensus or agreement in group communication systems should be taken into account implicitly in two aspects: One of them is supporting multicasting which should achieve agreement on the order, that is, history of state updates performed in each group. The other is supporting group membership which achieves agreement on history of group members. Fault-tolerant model is the state machine approach, or active replication which is explained previously. State machine approach is suitable for hard real-time applications, whereas primary/backup model is preferred in the case of soft real-time applications. The top layer consists of the distributed real-time application.

5 Cluster Based Fault Model

The fault tolerance policy developed in this study is an essential feature of the architecture. In the following, the phrase "routine operations" refers to either clock synchronization or process group management.

The fault tolerance can be considered as an extra layer whose interface with the routine operations is indicated with a dotted line as in figures Fig.8, Fig.9 and Fig.10 which give state transition diagrams.

The module can be examined in three different modes of operations: Namely, $NODE$, REP and $LEADER$.

The $NODE$ process is always active performing the routine designated operations. When the node process does not receive information from the representative process on time it goes into the "REP_FAILED" state which indicates that the normal operation mode is interrupted. A timer is started to see if there will be any message from the representative in the mean time. If the message is received before the timer goes off, the node turns back to its routine operations. If not, the representative is assumed to be crashed, it is discarded from the node list and an election process for a new cluster representative is started. When the new representative is elected, the node starts its routine operations from its initial state. If the new representative elected is the node itself, then it wakes up its $REPRESENTATIVE$ process to coordinate routine operations within the cluster.

The node process always knows the address of the $LEADER$ and updates it when new leader information arrives.

When the REP process is made active, the first thing it does is to send the leader the message that it has become the new representative. It then initiates the routine representative operations.

When the REP does not receive information from the leader on time, it goes into the $LEADER\_FAILED$ state which indicates that the normal operation mode is blocked. A timer is started to see if there will be any message from the leader in the mean time. If the message is received before the timer goes off, the representative turns back to its routine operations. If not, the leader is assumed to be crashed, it is discarded from the representative list and an election process for a new leader is started. When the new leader is elected, the representative broadcasts the new leader address to all nodes within the cluster and starts its routine operations from its initial state. If the new leader elected is the representative itself, then it wakes up its $LEADER$ process to coordinate routine operations within the group of clusters.

When the REP process does not receive information from a node process on time it goes into the "$NODE\_FAILED$" state which indicates that there might be something wrong with a node in the cluster. The normal operation mode is continued however, a timer is started to see if there will be any message from the node. If the message is received before the timer goes off, the representative continues to its routine operations. If not, the node is assumed to be crashed, it is discarded from the node list.

If a new node enters the cluster, its address is broadcast to all other nodes in the cluster.

```
Leader_process
{
cycle
   Wait_for_event;
   switch(event_type);
      REP_FAIL :
         start_timer(rep_dead);
      CLUSTER_FAIL :
         discard_cluster(failed_rep, cluster_no);
         repr_broadcast_cluster_data(cluster_list, repr_list);
         operate(cluster_list, repr_list);
      NEW_CLUSTER :
         add_cluster(repr, cluster_no);
         repr_broadcast_cluster_data(cluster_list, repr_list);
         operate(cluster_list, repr_list);
      NEW_REP :
         modify_cluster(repr,cluster_no);
         repr_broadcast_cluster_data(cluster_list, repr_list);
         operate(cluster_list, repr_list);
endcycle
}
```

Figure 2: Leader Fail Mode Algorithm

The new node is sent the routine operation list and the leader address. The cluster routine operations start from the initial state.

The REP process monitors both the nodes within the cluster and the leader simultaneously and enters into the respective failure state to solve any problem.

When the $LEADER$ process is made active, it initiates the routine operations by soliciting information from the representatives.

When the $LEADER$ does not receive information from a representative on time, it goes into the REP_FAILED state which indicates that the normal operation mode is blocked. A timer is started to see if there will be any message from the representative. If the message is received before the timer goes off, the leader turns back to its routine operations. If not, the representative is assumed to be crashed, it is discarded from the representative list and the other representatives are informed. If a new representative information is received either before or after the timer goes off, the leader broadcasts new representative information to other representatives, sends the new representative the representative list and the routine operations start from the initial state.

If the leader receives new cluster join request, it updates the representative list, broadcasts the new cluster representative to other representatives, sends the new representative the representative list and starts the routine operations from the initial state as shown in Fig. 2, Fig. 3 and Fig. 4.

6 Distributed Clock Synchronization

This section gives information on our proposed fault-tolerant distributed clock synchronization approach. Internal clock synchronization approach is used, that is, an approximate global time base among the ensemble of nodes of the distributed system is constructed. By this way, processor clocks are always kept within a certain maximum drift of one another. The clock synchronization protocol maintains local clocks at each processor in the distributed system by periodically becoming active. The algorithm is fault-tolerant in the sense that it tolerates omission failures (such as processor crashes, link crashes and occasional message losses) and

```
Representative_process
{
send_message(new_repr,leader,repr_addr);
cycle
  Wait_for_event;
  switch(event_type);
    LEADER_FAIL :
      start_timer(leader_dead);
      wait_for_message(message_from_leader);
    LEADER_DEAD :
      election_for_new_leader(cluster_list, repr_list, new_leader);
      cluster_broadcast(leader_elected, new_leader);
      if (my_node == new_leader)
        wakeup(Leader_process);
      operate(new_leader, cluster_list, repr_list, node_list);
    NODE_FAIL :
      start_timer(node_dead);
      operate(new_leader, cluster_list, repr_list, node_list);
    NODE_DEAD :
      discard_node(node_address);
      cluster_broadcast(delete_node, node_number);
      operate(new_leader, cluster_list, repr_list, node_list);
    NEW_NODE :
      add_node(node_address);
      cluster_broadcast(add_node, node_number);
      send_message(leader_info, node_address);
      operate(new_leader, cluster_list, repr_list, node_list);
    LEADER_ELECTION :
      election_for_new_leader(cluster_list, repr_list, new_leader);
endcycle
}
```

Figure 3: Representative Fail Mode Algorithm

```
Node_process
{
cluster_broadcast(new_node,node_addr);
cycle
   Wait_for_event;
   switch(event_type);
      REP_FAIL :
         start_timer(rep_dead);
         wait_for_message(message_from_rep);
      REP_DEAD :
         discard_node(failed_rep_addr);
         election_for_new_rep(node_list, new_rep);
         if (my_node == new_rep)
            wakeup(Representative_process);
         operate(leader, node_list, repr_addr);
      NEW_LEADER_ELECTED :
         receive_message(new_leader_elected, new_leader);
         operate(leader, node_list, repr_addr);
      ADD_NODE :
         add_node(node_address);
      DELETE_NODE :
         delete_node(node_address);
      REPR_ELECTION :
         election_for_new_repr(node_list, new_repr);
endcycle
}
```

Figure 4: Node Fail Mode Algorithm

performance failures (such as overloaded processors and slow links). It also handles processor joins.

We will first give definitions and assumptions concerning the distributed clock synchronization protocol. Then objectives of our model are explained. After that, distributed clock synchronization algorithms for leader, representative and node are given together with their descriptions and state transition diagrams.

6.1 Definitions and Assumptions

Let LC_p be defined as the local clock on processor p, let $LC_p(t)$ be the local clock time on processor p at time t and let $Clock_p(t)$ be the value of hardware clock on processor p at real time t.

Assumption 1: $Clock_p(t)$ of a processor p measures the passage of time between any two successive real-time instants t, t' correctly:

$$(1+X)^{-1} < (Clock_p(t') - Clock_p(t))/(t'-t) < (1+X)$$

Given by this assumption, correct clocks drift from real time by at most X.

Assumption 2: There is a known upper bound α on message delay. α consists of the time it takes for sending, transporting and receiving a message over a link. This assumption is valid for correct processors communicating over a correct link.

Assumption 3: During clock synchronization, no partition of the system of correct processors and links occur.

By using assumptions 2 and 3, if a message is sent by a correct processor A to a correct processor B and there exists N processors in the system, this message requires at most $maxdelay = (N-1) * \alpha$ of maximum delay bound between processors A and B.

6.2 Objectives

The goal of the clock synchronization approach is to ensure the properties of monotonicity, precision and linear envelope (also known as interval preservation).

Objective 1: Monotonicity: The local clock is a monotonically increasing function of time, such that:

$$\forall i : LC_i(t + \Delta t) \geq LC_i(t) \text{ where } \Delta t \geq 0$$

Objective 2: Precision: The synchronized local clocks of any two correct processors p and q are always within some maximum deviation of each other:

$$\forall t :\mid LC_p(t) - LC_q(t) \mid < MAXDEV$$

MAXDEV is the specified synchronization precision.

Objective 3: Linear envelope (or, interval preservation): any interval measured by the synchronized clocks are within some linear function of real time interval:

$$(1 + \beta)^{-1} \leq (LC_p(t) - LC_p(t'))/(t - t') \leq (1 + \beta)t > t'$$

That is, every processor p has a local clock LC_p with known bounded rate of drift $\beta \geq 0$ with respect to real time. β is the maximum clock drift rate.

6.3 Algorithm Description

Algorithms of leader, representative and node clock operations for our distributed clock synchronization protocol are given in figures Fig. 5, Fig. 6, and Fig. 7. Also, figures 8, 9, and 10 show state transitions for leader, representative and node clock synchronization respectively.

The distributed clock synchronization protocol works in two phases, namely intracluster clock synchronization (CS) and intercluster clock synchronization (CS). The protocol works in a periodic manner. The leader process broadcasts a $STARTINTERCLUSTER$ message to representative processes to initiate a CS period. Upon receiving this message, each representative starts the intracluster CS phase in its cluster. During this phase, a representative in each cluster is responsible for agreeing on a local clock value within the cluster. After determining its cluster clock value, the representative process returns it to the leader. By using cluster clock values collected from all correct representatives, the leader determines a new clock value for the distributed real-time system, and broadcasts this value to all cluster representatives. Then, each representative broadcasts new clock value to nodes in its cluster. Finally, each node sets its new local clock to that value.

Our protocol uses five timers which are called $TDETECTINTERCLUSTER$, $PERIOD$, $TDETECTINTRACLUSTER$, $LEADER\_FAIL$ and REP_FAIL. The leader process uses $TDETECTINTERCLUSTER$ timer to place a bound on collecting cluster clock values from cluster representatives. By this way, the leader does not wait responses from faulty clusters. In addition, the leader uses PERIOD timer to set a clock synchronization period. The representative process uses $TDETECTINTRACLUSTER$ timer to place a bound on collecting local clock values from nodes in its cluster. By this way, the representative does not wait responses from faulty nodes. Also, the representative process uses $LEADER\_FAIL$ timer to detect the case when current leader fails. Similarly, the node process uses REP_FAIL timer to detect case when current cluster representative fails.

7 Fault Tolerance by Process Groups

The model can be extended to cover the process group concept to provide fault tolerance to a distributed real-time system.

```
Leader_clock_process
{
cycle
  start_timer(TDETECTINTERCLUSTER);
  start_timer(PERIOD);
  repr_broadcast(STARTINTERCLUSTER, repr_list);
  Wait_for_event;
  switch(event_type);
    CLUSTER_CLOCK :
      update clustertimes array;
    Tout TDETECTINTERCLUSTER :
      if all_reprs_responded {
        determine_newlocalclock();
        repr_broadcast(NEWLOCALCLOCK,repr_list);
        start_timer(TDETECTINTERCLUSTER); }
        else event_type=CLUSTER_FAIL;
    Tout PERIOD:
  start_timer(PERIOD);
  repr_broadcast(STARTINTERCLUSTER, repr_list);
endcycle
}
```

Figure 5: Leader Clock Operation Algorithm

```
Representative_clock_process
{
cycle
  start_timer(TDETECTINTRACLUSTER);
  start_timer(LEADER_FAIL);
  Wait_for_event;
  switch(event_type);
    NODE_CLOCK :
      update times array;
    NEW_LOCAL_CLOCK :
      set_newlocalclock();
      cluster_broadcast(NEW_CLOCK,node_list);
    Tout TDETECTINTRACLUSTER:
      if all_nodes_responded {
        determine_newclusterclock();
        send_message(CLUSTER_CLOCK,Leader); }
        else event_type=NODE_FAIL);
    STARTINTERCLUSTER :
      send_message(CLUSTER_CLOCK, Leader);
      start_timer(LEADER_FAIL);
    Tout LEADER_FAIL :
      event_type=LEADER_FAIL;
endcycle
}
```

Figure 6: Representative Clock Operation Algorithm

```
Node_clock_process
{
cycle
   start_timer(REP_FAIL);
   Wait_for_event;
   switch(event_type);
      NEW_CLOCK :
         setnew_localclock();
      STARTINTRACLUSTER :
         read_localclock();
         send_message(NODE_CLOCK, Repr);
         start_timer(REP_FAIL);
      Tout REP_FAIL :
         event_type=REP_FAIL;
endcycle
}
```

Figure 7: Node Clock Operation Algorithm

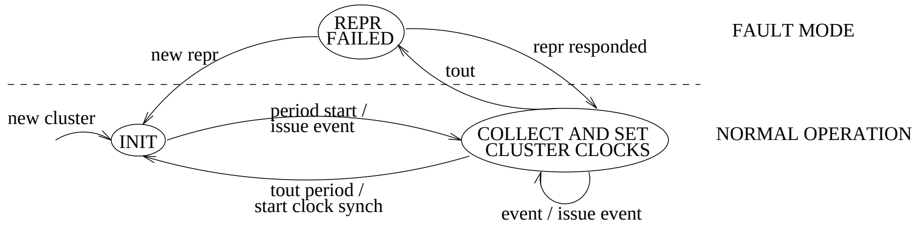

Figure 8: State diagram for leader clock synchronization

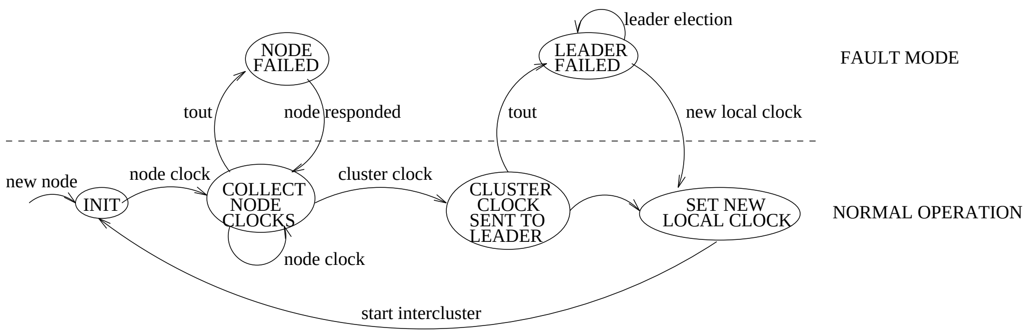

Figure 9: State diagram for representative clock synchronization

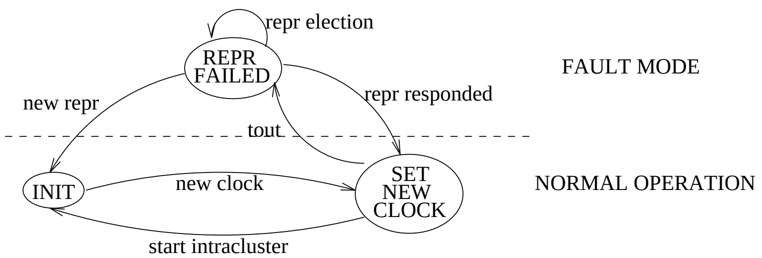

Figure 10: State diagram for node clock synchronization

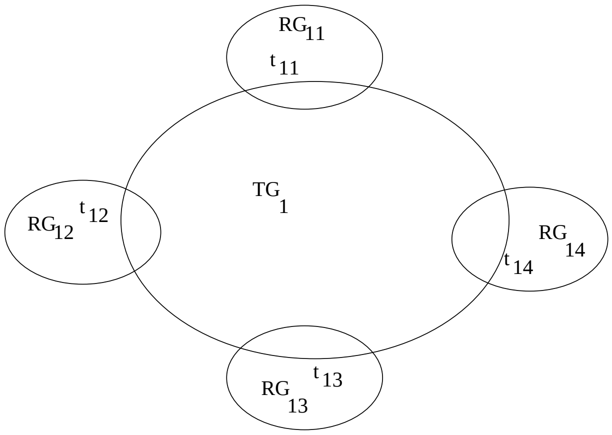

Figure 11: An Example Group

We assume that the distributed hard real-time application consists of periodic hard real-time tasks T_i. We further assume that each T_i consists of task threads t_{ij} *for j=1..*n_i where n_i is the number of threads of task T_i. The threads of a task can be illustrated by a graph $G_i = (V_i, E_i)$ where V_i is the set of task threads (nodes) and E_i is the set of edges denoting communication costs among these threads.

We provide a *Cluster Allocation* algorithm which partitions G_i using suitable heuristics such as *Heavy Edge Matching* [5]. In the second phase, the *Task-to-processor Mapping* algorithm provides the final allocation sequence for the task threads.

All threads of a task constitute the task group TG_i. The task threads have replicates t_{ijk} distributed over the network where i is the task, j is the thread and k is the replicate number and k=0 means the original task thread. All the replicas of a task thread constitute the replica group RG_{ij}. The relation between TG_i and RG_{ij} is depicted in Fig. 11. There are four threads of task T_1 belonging to group TG_i and each of these threads have their own groups as shown.

The aim of the fault tolerance using process groups is twofold; firstly, we need to provide ordering of two types of events; multicast communication among group members and group membership (join/exit) operations within the task thread group TG_i . The ordering of these events is needed so that the same sequence is absorved by each member for correct operation of the system. We also need to provide that all events related to a task thread are observed in the same sequence by all replicas of this thread. For example, if t_{14} sends a unicast message m_1 to t_{12} and t_{11} sends a message m_2 to t_{12} such that $m_1 \prec m_2$ (m_1 precedes m_2), we should provide that every task thread replica in RG_{12} receives these messages in the same order. i.e. $m_1 \prec m_2$. Furthermore , say t_{11} sends a multicast message to task threads in TG_1, we need to provide that every thread replica in groups RG_{11}, RG_{12}, RG_{13} and RG_{14} receive this message and other multicast messages, group membership operations in the same order.

To be able to solve the problems addressed above, we use the leader and concepts discussed in section 4. The process group members in each cluster whether task group or the replica groups are managed by a *representative process(REP)* in the cluster and one of these representatives is the $LEADER$ similar to what is described for processor clusters in section 3. We provide a *token passing protocol* on the logical ring of $REPS$. In this protocol, the $LEADER$ periodically issues a $FREE$ token and sends it to the first REP in line. Upon receipt of the free token, each REP thread inserts the pending events in its cluster with their timestamps into the token queue. Once this free token circulates and is received back by the $LEADER$, it sequences the events with respect to their time stamps and sends the $ORDER$ token to dictate the order of events to the REPs. Upon receipt of this token, each REP dictates this order to the threads and their replicas in its cluster as shown in Fig. 12. The protocol applied by the REPs and the $NODE$s are depicted in Fig. 13 and Fig. 14. The failure of the REP or the $LEADER$ or the $NODE$ threads are treated in the same manner as the same processes in the distributed clock synchronization module.

```
Event_Leader() { Initialize(); /* set current Representatives and Leader */
cycle
  Wait_for_event;
  switch(event_type);
    PERIOD_START : Issue a FREE_TOKEN;
                   Send FREE_TOKEN to first representative;
    FILLED_TOKEN : Order events w.r.t. timestamps;
                   Send ORDERED_TOKEN to first rep;
    ORDERED_TOKEN : Check ACKs in token;
                   If an ACK is missing,
                     send MONITOR_MSG to each rep;
    MISSING_TOKEN : Send MONITOR_MSG to each rep
                   Wait for MONITOR_ACK;
    MONITOR_ACK : Update configuration;
    REP_FAILED : Update configuration;
    NEW_REP : Update configuration;
endcycle }
```

Figure 12: Event Leader Algorithm

```
Event_Representative() Inputs:
{ Initialize(); /* set current Representatives and Leader */
cycle
  Wait_for_event;
  switch(event_type);
    FREE_TOKEN : Place event list in the token;
                 Send token to next rep;
    NODE_EVENT : Update event_list;
    MONITOR_MSG : Send MONITOR_ACK to leader;
    ORDERED_TOKEN : Store ordered events in ord_event_list;
              Broadcast ORD_EVENTS to nodes;
              Wait for ACKs from nodes
              If all ACKs not received change node config;
              Set ACK in token;Send token to next rep;
endcycle }
```

Figure 13: Event Representative Algorithm

```
Event_Node() Inputs:
{ Initialize(); /* set current Representatives and Leader */
cycle
  Wait_for_event;
  switch(event_type);
    EVENT : Record event with a timestamp;
                 Send NODE_EVENT to cluster rep;
    ORD_EVENTS : Perform ordered events;
                 Send ACK to cluster rep;
endcycle }
```

Figure 14: Node Algorithm

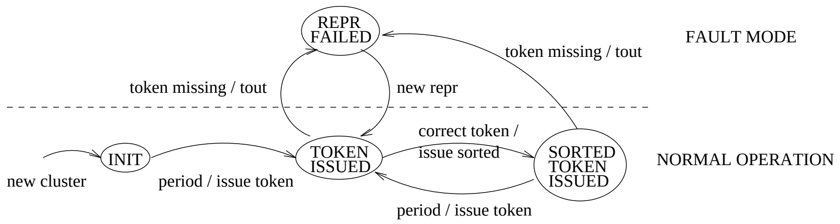

Figure 15: State diagram for leader group management

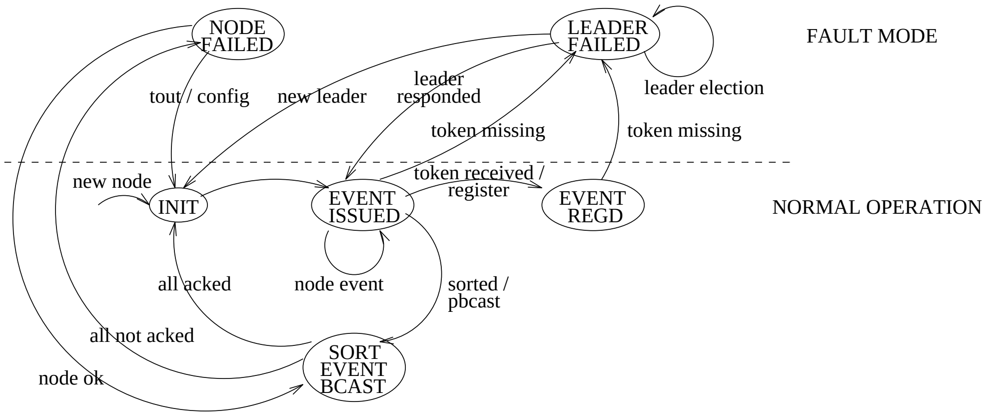

Figure 16: State diagram for representative group management

The state diagrams are depicted in Fig. 15, Fig. 16 and Fig. 17.

8 Implementation and Performance Considerations

The proposed model of distributed clock synchronization was implemented with four clusters that have 10 Mbps Ethernets connected by a 100 Mbps FDDI ring as shown in Fig. 18. We have two representatives and one leader process. Cluster A consists of 8 DEC Alpha 2000 workstations, Cluster B consists of a DEC Alpha 2000, a Sun Ultra 1 and a PC all running Unix derivatives such as OSF1/Mach on Alphas, Solaris on Sun and Linux on PC, Cluster C, D and E are all similar with a DEC Alpha 2000 server and various PCs.

The datagram protocol (UDP) using sockets was used as the network interface with an added layer of reliability for communication between the nodes and the representatives. The $LEADER$ forks a child to handle UDP communications with the REPs of other clusters and System V message queues were used for interprocess communication between the children of the $LEADER$ and itself which are in the same node as shown in Fig. 19

The first test involved taking time measurements for the settling of the time among the $REPs$ by the direction of the $LEADER$. This settling time t_{sync} was measured against the

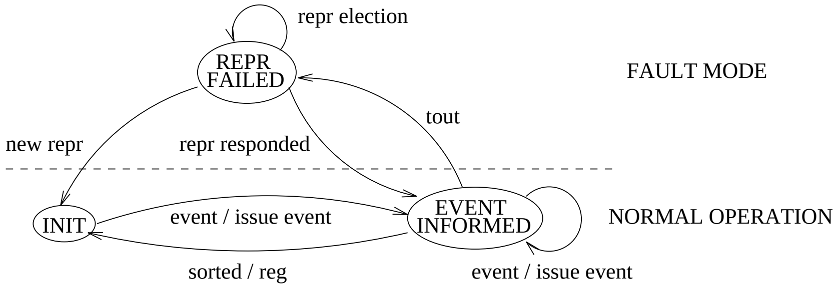

Figure 17: State diagram for node group management

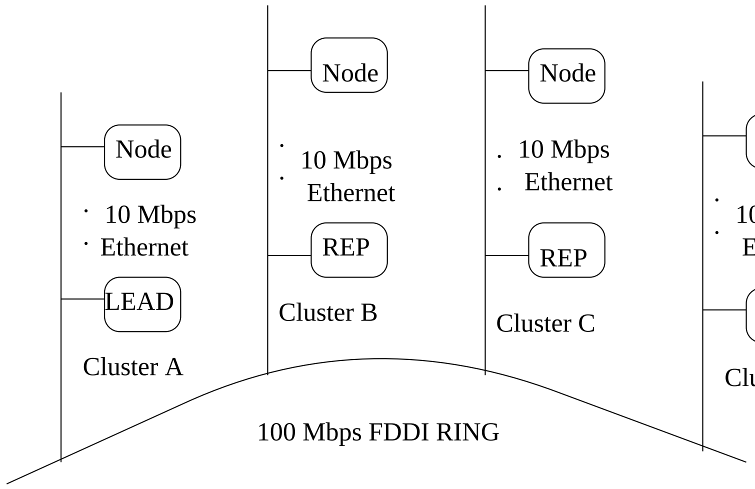

Figure 18: The Test Environment

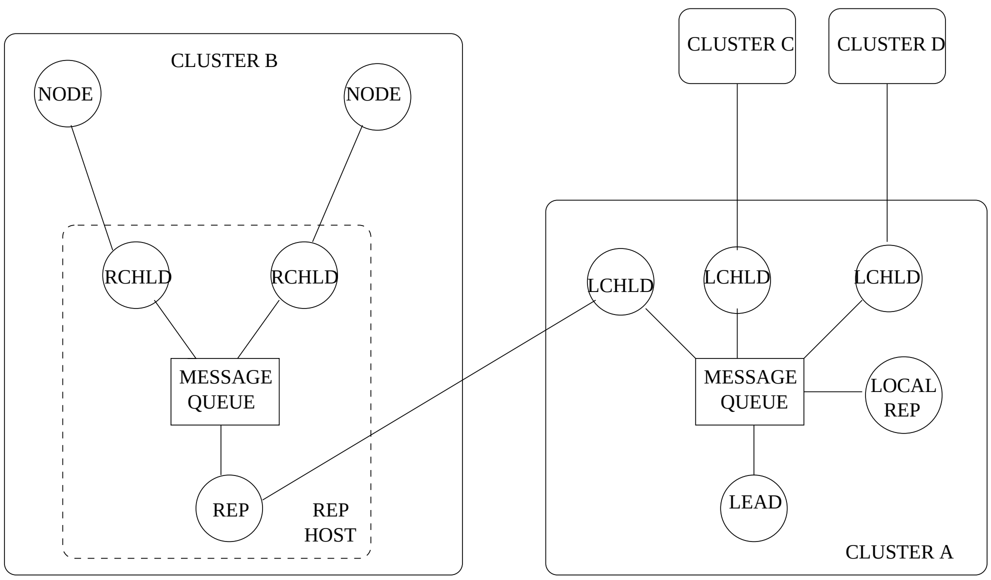

Figure 19: The Test Programming Model

Table 1: Settling time versus number of clusters

Number of clusters	2	3	4
Settling time(ms)	27.32	42.21	55.22

number of clusters varying from 2 to 4. The number of nodes was kept unity and as depicted in Table 1, t_{synch} varies linearly with the number of clusters.

The second test involves the measurement and calculation of the standard deviation of the clock drifts td_i of the clusters with respect to number of clusters and the duration of the synchronization period T_s. From the tests carried so far, we have found that td_i, irrespective of the cluster, varies within 1 ms in psitive and negative direction. However, this drift does not change with T_s. We were expecting that clock drifts would increase with increased T_s.

9 Conclusions

We have proposed system model to manage time and provide fault tolerance using process groups in a distributed real-time system. We also provide a separate fault tolerance module to survive crashes of the so called REP, $LEADER$ and $NODE$ processes. An important aspect of this model is that the management model and the architecture of the fault tolerance module are independent of the layer under consideration whether this is management of time or process groups or load balancers. Our intended contribution is threefold: We claim that any resource in a distributed real-time system can be managed by this synchronous cluster based model. Secondly, for any layer of the system that is managed by the cluster scheme, the fault tolerance concepts developed for the managing processes are valid. Thirdly, fault tolerance can be achieved conveniently by this cluster model.

It should be noted that the fault tolerance layer which uses process groups for replication of task threads aims at the ordering of events for thread and replica groups so that any application hard real-time thread failing can be replaced by a replica that is in the same state as the failed thread. The managing processes for this layer however use the generic fault management concepts to replace a failed $LEADER$, REP or $NODE$.

Our initial tests show that the overhead of the distributed clock synchronization module increases linearly with the number of clusters. The clock drifts seem to be independent of the clock correction period which is unexpected.

Our work is ongoing and we are in the process of testing the process group management module for fault tolerance. We are also extending these concepts to static and dynamic load balancing in distributed real-time systems which have hard and soft real-time tasks. Our current implementation covers hard real-time tasks and static scheduling of these task threads first to clusters using graph partitioning methods and allocation of these threads to individual processors of the clusters in the second phase. The next layer to implement is the dynamic load balancing of the soft real-time tasks using the same concepts so that the management of the load in a cluster is performed by a REP and the $LEADER$ manages the overall load in the system. In this case, each REP is responsible for the dynamic allocation decisions in its own cluster and if this is not possible, it can request transfer of the load to a lightly loaded cluster under the supervision of the $LEADER$.

From the results obtained so far, we can say that this hierarchical model is suitable to manage large and complex distributed real-time systems as the overheads increase linearly with the number of clusters and the programming of the system is simplified by the proposed divide and conquer method.

References

[1] Erciyes, K. Ozkasap, O., Aktas, N., A Semi-Distributed Load Balancing Model for Parallel Real-Time Systems, Informatica- An International Journal of Computing and Informatics, Special Issue: Parallel and Distributed Real-Time Systems, Vol.19, No.1, February, 1995, ISSN 0350-5596.

[2] Cristian, F., Synchronous and Asynchronous Group Communication, IEEE Workshop on Fault-tolerant and Parallel Distributed Systems. Honolulu, Hawaii, 1994.

[3] El-Rewini H., Task Partitioning and Scheduling on Arbitrary Parallel Processing Systems. PhD. Thesis, Oregon State University, Computer Science Dept., 1989.

[4] Goscinski, A., Distributed Operating Systems: The Logical Design. Addison-Wesley, ISBN 0-201-41704-9, p. 300-344, 1991.

[5] Karypis G., J. & Kumar J., Analysis of Multilevel Graph Partitioning, University of Minnesota, Dept. of Computer Science,Tech. Report 95-037, 1995.

[6] Birman K.P., Cooper R., Gleeson B., Design Alternatives for Process Group Membership and Multicast, Technical Report - TR911257, Dept. of Computer Science, Cornell University, December 20, 1991.

[7] Moser L.E., Melliar-Smith P.M., Agarwal D.A., Budhia R.K., et. al, Totem: A fault-Tolerant Multicast Group Communication System, Communications of the ACM, Special Section on Group Communication, 39(4), April, 1996.

[8] Dolev D., Malki D., The Transis Approach to High Availability Cluster Communication, Communications of the ACM, Special Section on Group Communication, 39(4), April, 1996.

[9] Reiter M., Distributing Trust with the Rampart Toolkit, Communications of the ACM, Special Section on Group Communication, 39(4), April, 1996.

[10] Van Renesse R., Birman K.P., Maffeis S., Horus: A Flexible Group Communication System, Communications of the ACM, Special Section on Group Communication, 39(4), April, 1996.

[11] Schiper A., Raynal M., From Group Communication to Transactions in Distributed Systems, Communications of the ACM, Special Section on Group Communication, 39(4), April, 1996.

[12] Van Renesse R., Birman K.P., Protocol Composition in Horus, Technical Report - TR951505, Dept. of Computer Science, Cornell University, March 29, 1995.

[13] Van Renesse R., Hickey T.M., Birman K.P., Design and Performance of Horus: A Lightweight Group Communications System, Technical Report - TR941442, Dept. of Computer Science, Cornell University, August, 1994.

[14] Birman K.P., The Process Group Approach to Reliable Distributed Computing, Communications of the ACM, 36(12), December, 1993.

[15] Van Renesse R., Birman K.P., Fault-tolerant Programming Using Process Groups, IEEE Distributed Open Systems, December 1993.

[16] Birman K.P., Cooper R., The ISIS Project: Real experience with a fault-tolerant programming system, Dept. of Computer Science, Cornell University.

[17] Babaoglu O., Davoli R., et. al, RELACS: A Communications Infrastructure for Constructing Reliable Applications in Large-Scale Distributed Systems, Technical Report - UBLCS-94-15, Laboratory for Computer Science, Univ. of Bologna, Italy, January, 1995.

[18] Malki D., Multicast Communication for High Availability, Ph.D. Thesis, Hebrew Univ. in Jerusalem, 1994.

[19] Shivakant M., Schlichting R., Abstractions for Constructing Dependable Distributed Systems, Technical Report - TR92-19, Dept. of Computer Science, University of Arizona, August 1992.

[20] Kopetz H., Ochsenreiter W., Clock Synchronization in Distributed Real-Time Systems, IEEE Transactions on Computers 36(8), August 1987.

[21] Cristian F., Aghili H., Strong R., Clock Synchronization in the Presence of Omission and Performance Failures, and Processor Joins, Technical Report, IBM Research, Almaden Research Center, 1986.

[22] Jalote P., Fault Tolerance in Distributed Systems, Prentice Hall, Inc., 1994.

[23] Attiya H., Herzberg A., Rajsbaum S., Optimal Clock Synchronization under Different Delay Assumptions, November, Technical Report, Dept. of Computer Science, Technion, Haifa, Israel, November 1994.

[24] Mullender S. (editor), Distributed Systems, ACM Press - Addison-Wesley Publishing Comp., Second edition, 1993.

[25] Cristian, F., Understanding Fault-Tolerant Distributed Systems, Technical Report, Dept. of Computer Science and Engineering, Univ. of California, San Diego, May 1993.

[26] Hadzilacos V., Toueg S., Fault-Tolerant Broadcasts and Related Problems, Distributed Systems - Chapter 5, Edited by Sape Mullender, ACM Press, 2nd edition, 1993.