Reliable SRT inter-node multicast in DEDOS

Belgers, W.H.B.

Award date:
1994

EINDHOVEN UNIVERSITY OF TECHNOLOGY
Department of Mathematics and Computing Science

MASTER'S THESIS

Reliable SRT inter-node multicast in DEDOS

by

W.H.B. Belgers

Supervisor : dr. P.D.V. van der Stok
Advisors : ir. D. Alstein
dr. J.J.M. Hooman

August 1994

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Belgers, W.H.B.

Reliable SRT inter-node multicast in DEDOS / by W.H.B.
Belgers. – Eindhoven : Eindhoven University of
Technology, Department of Mathematics and Computing
Science
Afstudeerverslag. – Met lit. opg.
ISBN 90-386-0094-1
Trefw.: operating systems

Contents

Introduction

The **DE**pendable **D**istributed **O**perating **S**ystem (**DEDOS**) project is currently under development at the Eindhoven University of Technology, and is meant to study *dependability*, in particular timeliness and reliability. The main purpose of this project is to develop an experimental environment for the building and testing of *embedded distributed systems*. The first stage in the project is the development of a distributed operating system (DEDOS), that will function as a test bed for further activities.

Embedded systems are computer systems which are embedded in another system (the controlled system), and are not directly visible outside the controlled system. Examples of embedded systems are process control systems (like a nuclear plant control system) and telecommunication systems (like a network management system).

Embedded systems are characterised by:

- very high **dependability** requirements on hardware design (robust, correct and reliable) as well as on software design,
- high **complexity**,
- a long system **lifetime**: its design has to meet extra maintainability and extendibility requirements,
- **multi-disciplinary** approach: eventually one total system has to be built, which consists of mechanical parts (valves, pipes etc.), electrical parts (sensors, controllers etc.) and software parts (operating system etc.), this requires extra coordination and control activities,
- geographical **distribution** of the system, not only determined by the location of the controlled system, but also due to other reasons like enhanced reliability (redundancy) and enhanced performance (parallelism),
- complex **user interfaces** for large systems, for the purpose of displaying the system status, changing system configuration and parameters, and collecting system statistics.

These embedded systems require particularly high dependability. A general definition of dependability is the ability to deliver the specified quality of service in such a manner that the user or another system can justifiably rely upon the system.

Attributes of dependability are:

- **timeliness**, the ability to perform the specified service within strictly bounded *deadlines*,
- **reliability**, the probability that the specified services are performed without any failures, measured over a given interval of time,

- **availability**, the probability that the specified services are performed, measured over a given interval of time; in case of failures, small repair and recovery times are required to improve availability,
- **safety**, the ability to avoid unsafe system states, i.e. states of the controlled system that can be dangerous, and to provide protection against intruders by means of encryption and authentication techniques,
- **robustness**, the ability to withstand undetected design errors in both hardware and software, faulty inputs, load variations and environmental influences like heat.

Some reasons for trying to achieve dependability in computer systems are:

- to minimise loss of life: people more and more depend on automatic systems, e.g. telematic services, control of vehicles, control of health critical processes,
- to minimise loss of goods: more goods are automatically produced and shipped,
- to prevent additional costs (caused by the effects of the failed system),

In addition, embedded systems (and computer systems in general) need to be maintainable, extendible (modular and flexible), and scalable, in order to lengthen their lifetime and reduce costs.

Distributed systems are very suitable to implement an embedded system, because embedded systems are very often distributed by nature. A well-structured distributed system can be partitioned into three layers. The bottom layer is the hardware layer, which, among others, includes a collection of processors interconnected by a network.
On top of the hardware is the *Distributed Operating System* (DOS). A distributed operating system appears to be an ordinary centralised operating system to its users, but it runs on multiple independent processors connected via a network. The multiple processors and an interconnected network should be transparent (invisible) to the user.
On top of the DOS layer is the application layer.

The concept of transparency in a DOS is in contrast with a network operating system (NOS), where users are aware of the existence of distinct machines and the network interconnecting these machines. A user of a NOS normally works on his or her machine. Using a different machine requires some kind of 'remote login'. Also, the machines are aware of the locations of the resources and must move files between machines with explicit 'file transfer' commands. Furthermore, if n percent of the machines crash, all users making use of the crashed n percent of computers (typically n percent of the users) will be out of business.
In contrast to a NOS, a DOS makes processors work as an integrated network with common distributed network management. Resource allocation is transparent for users, so users view the system as a (virtual) uni-processor system.

Summarising, a distributed system consists of a collection of interconnected processors, with groups of processes (executing programs) running on them, cooperating to perform a given task. Some features of distributed systems are:

- physical distribution: put the computer power where it is needed,
- enhanced performance by true parallelism,
- incremental growth: if n percent more computer power is needed for new functionally disjoint processors, just n percent more processors have to be added,

– enhanced reliability and availability: this can be established through redundancy (replication) of resources.

As we have seen, a major purpose of the DEDOS project is to offer reliability. One way to achieve reliability is to introduce the concept of *fault-tolerance*. A fault-tolerant system can continue to provide the specified standard service despite the occurrence of component failures. It is evident that the design and understanding of fault-tolerant distributed systems is quite difficult: one has to stay in control not only of the standard system activities when all components behave according to their specifications, but also of the complex situations which can occur when some components fail.

In DEDOS, a process is an instance of a program. DEDOS distinguishes two kinds of processes: *Hard Real-Time (HRT)* and *Soft Real-Time (SRT)* processes. HRT processes have *deadlines* that have to be met at all cost. Exceeding these deadlines can lead to catastrophical situations. A well-known bizarre example of this is a HRT process that operates the landing-gear of an aeroplane that may exceed its deadline and cause a plane to crash. It is clear that one of the important tasks of DEDOS is to guarantee that HRT processes always meet their deadlines, in the absence of unforeseen (hardware) faults. This will be realized by executing programs deterministically at times specified by an off-line schedule. On the other hand, SRT processes are not time critical. When some SRT deadlines can not be met, those SRT processes are aborted. SRT processes are scheduled dynamically (on-line). A typical SRT task would be the air conditioning system in an aeroplane.

The **E**indhoven **M**ulti **P**rocessor **S**ystem (**EMPS**), is the hardware on which DEDOS is built. It is not a purely shared-memory system. The highest level consists of *nodes*, interconnected by a LAN and serial links. Each node contains a number of processors and memory modules, connected to a common bus (see [Dijk]). Processes, executed on different processors, but cooperating to provide a service or perform a given task, have to be able to communicate with each other in a reliable way. An important communication service provided by DEDOS is the *reliable multicast*. This service enables a process to send a message to a given subset of all other processors. The multicast is reliable because it functions correctly, even in the presence of failures (processor-crash, link break down etc). As mentioned before, HRT processes require other timeliness characteristics of the service than SRT processes. Therefore, separate multicast protocols have to be developed for HRT and SRT multicast.

A reliable multicast protocol for the communication within a node between HRT processes [Alst] and SRT processes [Verm], as well as a reliable multicast protocol for the communication between **HRT** processes executing on different nodes [Kenn] have already been developed.
This thesis presents a protocol for reliable communication between **SRT** processes executing on different nodes.

The remainder of this thesis is constructed as follows:
In Chapter 1, multicast is explained in more detail. Some protocols that already exist are presented too. In Chapter 2, we take a look at the system in which the protocol presented here has to work: a short overview of both software and hardware parts of DEDOS and EMPS is given. Chapter 3 presents the major part of this thesis: the reliable SRT inter-node multicast protocol for DEDOS, which I have been working on during my graduation period. Finally, in Chapter 4, a conclusion and some suggestions are given.

Chapter 1

Reliable Multicast and Membership

This chapter gives a general impression of what reliable multicast and membership have to achieve. Some existing protocols are presented, in order to give an impression of how communication problems can be solved. Along the way, some terminology will be introduced.

1.1 What is reliable multicast?

In distributed systems, groups of processes, running on a collection of processors, cooperate in some way or another to provide a number of services. Having multiple processes work on a collective task, but executing them on different processors, increases the reliability and availability of the system. For example, if one or more of the processors fail, the processes on the remaining processors can still offer the required service.
Therefore, most distributed services need some kind of global state to be able to function properly. If processes don't use common memory —e.g. because there is no common memory— *replication* of service state information is necessary. Even if processes do use common memory, replication of information is desirable to avoid single points of failure. Another reason for replicating information is performance. Consider an application that has to perform computations on certain data. The application is divided into several processes to benefit from true parallelism. Replication of the data on the processors on which the processes execute, enables the processes to access the required data locally. This increases performance in case the amount of send-operations is higher than the amount of receive-operations, since processes don't have to access the network to receive the data they need.

However, replication of information needs to be *consistent*. If data replicas are not consistent with each other, replicated processes that use the data replicas, are very difficult to design. Replica consistency makes a replicated storage similar to a centralised, shared storage. Consequently, known concurrent programming techniques based on shared storage can be used in a distributed environment.
To keep replicas consistent with each other, processes have to be able to communicate updates to each other in such a way that, despite component failures and random communication delays, all correct processes can make the same updates in the same order. Communication services that achieve this, are called *atomic multicasts* or *broadcasts*. Broadcast enables a processor to send a message to all destinations in the system. Multicast is similar, but limits the distribution to a

specified subset of all destinations, a *multicast group*.

Either processes or processors can be considered as destinations. Generally, broadcast between processes is not needed, since normally there won't be any kind of information that is of concern to all processes. Typically, only processes that cooperate with each other will like to communicate with each other. So, in the remainder of this thesis, broadcast will denote broadcast between **processors**, while multicast will mean multicast between **processes**.
The protocols that are discussed in the next section all implement reliable multicast between processes as a layer on top of reliable broadcast between processors. A multicast message is reliably broadcast to all processors, and each processor *delivers*, i.e. makes available, the message to its local processes that are members of the multicast group specified in the message.

Different kinds of applications can impose different demands on the multicast service. Applications that are time-critical will demand an upper bound on the time it takes to multicast a message to all destinations in the multicast group. For instance, if there is an application, that controls the automatic pilot of an aeroplane, late delivery of vital information could cost many lives.

For applications that are less time-critical, it is not of vital importance that information is delivered within a bounded period of time. Late delivery will not be catastrophic.
In general, if a multicast group imposes strict timing demands on the multicast, the processes of this group will be *Hard Real-Time* (HRT) processes, otherwise they will be *Soft Real-Time* (SRT) processes.

1.2 Existing Protocols

In [Cri1], Cristian distinguishes two types of protocols for reliable broadcast: synchronous and asynchronous protocols.
Asynchronous protocols, like the ones described in [Chan] and [Mell], use *retransmissions* to guarantee delivery to all correct processors. Asynchronous protocols do not guarantee an upper bound on communication delays, as opposed to synchronous protocols. Synchronous protocols, like the ones described in [Cri0] and [Cri1], send messages along multiple channels to guarantee delivery to all correct processors.

1.2.1 Asynchronous Communication Protocols

In [Chan], a reliable broadcast protocol is presented. All messages pass through an intermediate node, called the *token site*. The system operates as a *positive acknowledgement* system, i.e. the broadcast message will be retransmitted until an acknowledgement is received from each of the receivers, between the sources and the token site. It operates as a *negative acknowledgement* system, i.e. messages that are lost are detected when a higher sequence number than expected is received, between the token site and the remaining receivers. An elegant token-passing protocol is used to detect failures at the token site, to select a new token site, and to retransmit messages affected by the failure. Typically, about three messages are required for each broadcast message, and the latency is reasonable low at low loads but increases at high loads. The need to recognise a failed processor, and to reconfigure the system to exclude it, introduces long delays when a processor fails.
In [Mell], acknowledgement-, timeout- and retransmission-techniques are used to establish reliable broadcast. By acknowledging a message B, that contains an acknowledgement for a message A, a processor indirectly acknowledges message A. With these 'chains' of acknowledgements, a

partial order on the messages is derived. A fully decentralised voting protocol is used to construct, with high probability, a total order from the partial order.
In [Kaa0], a processor sends a message to a central processor called the *sequencer*. The sequencer assigns sequence numbers to the message in order to place a total ordering on them. Then, it stores the message in a *history-buffer* for possible retransmission, and broadcasts it to all processors. Processors that receive the message also store it in a history-buffer. A lost message is detected by a missing sequence number, in which case a processor asks the sequencer for retransmission (negative acknowledgement). Processors acknowledge received messages by piggybacking the sequence number of the last received message they send to the sequencer. From the piggybacked acknowledgements, all processors can determine which messages are received by all processors and may therefore be removed from the history. If the sequencer crashes, the processor that has received the highest sequence number is elected sequencer. For a more detailed description of this protocol, see appendix E.
Reliable multicast can be implemented as a layer on top of these three reliable broadcast protocols, by letting processors deliver a received and ordered message to the processes for which it was intended, or by discarding the message if there is no process that wants this message.

1.2.2 Synchronous Communication Protocols

Synchronous protocols assume that message delays among correct processors are bounded, and guarantee an upper bound on the time it takes to broadcast information. They assume the existence of enough redundant communication paths between processors, so the probability that during the broadcast no path exists between any two processors is negligible.
The protocols in [Cri0], designed for point-to-point networks, use *message diffusion* to reach all correct processors in a broadcast. When a correct processor receives a broadcast message for the first time, it forwards the message on all paths, except the one it arrived on. This way, a message is *diffused* throughout the network, i.e. all communication paths from the initiating processor to every other processor are used.
The protocols in [Cri1], designed for redundant broadcast channels, send a broadcast message over multiple, independent broadcast channels by the initiating processor. Processors decide on which ones. The forwarding rule must ensure that all correct processors receive the message on at least one channel, even if at most f components can be faulty.
After the broadcast is completed, all processes deliver the message to the processors for which it was intended, at the same specific clock time. To enable processors to deliver a multicast message to processes at the same clock time, synchronous protocols assume (approximately) synchronised, monotonically increasing clocks, and the presence of some scheduler that can schedule the delivery at a given clock time. This time of delivery can be calculated from the upper bound on the time it takes to broadcast a message and the time of initiation of the broadcast. The latter one is included in the message as a timestamp. These timestamps are also used to supply a total ordering on the messages.
If there is no process that wants the message, the message is discarded. This way, multicast is implemented as a layer on top of broadcast.

1.2.3 A comparison between Asynchronous and Synchronous Protocols

Synchronous and asynchronous protocols both guarantee *atomicity*, i.e. if a correct processor receives a message, then all correct processors receive and deliver it. Also, they guarantee *order*, i.e. messages are delivered to all correct processes in the same order. Due to random communication delays and component failures, processors do not necessarily receive the messages in the same order. But, they all place the same order on them before delivering them to the processes.

The order in which the messages are delivered depends on the protocol that is used, and is not necessarily the order in which the messages were sent in real-time.
Asynchronous protocols use *time redundancy*, i.e. they send a message several times, using acknowledgements, timeouts and retransmissions, until all destinations have received the message. These protocols also use some kind of message numbering to provide an ordering.
On the other hand, synchronous protocols use *component redundancy*, i.e. they send a message over many communication paths, to ensure that a message arrives at all destinations. Timestamps, read from synchronised clocks, are used to supply an ordering on messages.

The results of using this strategy are:

Synchronous protocols guarantee *bounded termination time*, even when faults occur. Asynchronous protocols do not generally guarantee bounded termination time. They have a variable termination time when failures occur during transmission. However, if no failures occur during transmission, asynchronous protocols will be faster than synchronous ones, because asynchronous protocols send less messages per broadcast, and therefore need less processing time.

Data replicas can be kept 'more' consistent with synchronous protocols than with asynchronous protocols. Because updates that are multicast among replicated processes, using a synchronous protocol, can be processed within a specified interval of clock time after the termination of the multicast. Notice that in this case the processor clocks have to be synchronised.
It is not possible for asynchronous protocols to guarantee that updates are processed within a specified interval of clock time, because they have a variable termination time. In this case, processors will never know exactly at which clock time all processors have received the same update. It can only be guaranteed that the same updates are made in the same order.

Asynchronous protocols offer the possibility of implicitly tolerating failures that cause a *communication network partitioning* by simply retransmitting messages until the network partitioning is resolved. By contrast, there are no known synchronous protocols yet that can tolerate failures resulting in a network partitioning. Separate protocols have to be designed to detect partitions and make data replicas consistent after the network partitioning has been resolved.

Concluding, because of the bounded termination time, synchronous protocols are suitable for HRT applications, that always have to meet their deadlines, even in the presence of failures. Asynchronous protocols will be used by SRT applications, where the cost of not meeting some deadlines is not too high.

1.3 Membership

The membership problem is a fundamental problem of distributed computing, like reliable broadcast/multicast or clock synchronisation, in the sense that once solved it allows easy solutions to other problems in designing *fault-tolerant* distributed applications, e.g. the problem of ensuring high availability of computing services or the problem of load balancing (distributing the load among the available processors).
Consider for instance the situation where a *server process*, providing some service, has been replicated for reliability, availability and/or performance. In other words, there are multiple replicas (of the process and its data, i.e. its service state), running on different processors, all capable of providing the service to a client requesting it. Such a set of processes (with their data) will be called a *server-group* in this thesis. Reliable multicast is used to keep the data replicas consistent with each other. Therefore, a server-group will very often be a multicast-group.
Now, suppose a client process requests the service, and a member p of the server-group starts the service locally. During execution, p crashes, either because the processor it is located on crashes,

or because the operating system stops p because it tries to perform an illegal action (e.g. division by zero or accessing protected memory). Another member of the server-group then has to take over the job, but if no information is known about which processors and server-group members are still functioning, how can one be sure to elect a correct member, or even know that one has to be elected?
A membership service that keeps track of the correctly functioning processors and processes makes these things a lot easier.

In [Cri0], Cristian divides the membership problem into two sub-problems. First, he shows how to achieve *agreement* on the identity of all correctly functioning processors that can execute server processes. He calls this problem the *processor-group membership problem*, a name that is adopted in this thesis as well.
Second, assuming the processor-group membership problem solved, he shows how to solve the *server-group membership problem*. The solution to the latter problem shows how to maintain agreement on the global state of a server-group when server-joins cause the group membership to increase.

For the processor-group membership problem, Cristian presents three protocols, the first of which will be briefly described here. To enable any process on any processor to use the processor-group membership service, Cristian implements this service as a group of membership server processes replicated on all processors of the system. The protocol assumes a synchronous reliable broadcast protocol with bounded termination time Δ and synchronised clocks, and works as follows.
Join handling: when a processor is newly started, its membership server invites the other processors to form a new group in which it is included itself. It does so by broadcasting a 'new-group' message with the synchronous broadcast protocol. If the message is broadcast at local clock time T, all processors will have received it by their local clock time $T + \Delta$. In response to a 'new-group' message, each processor, including the one that sent it, broadcasts a 'present' message that contains its identifier and its willingness to join a new group, at clock time $T + \Delta$. The atomicity and bounded termination time properties of the synchronous broadcast protocol ensure that all processors, that are correct during $[T + \Delta, T + 2\Delta]$ receive by clock time $T + 2\Delta$ the same set of 'present' messages, and hence, all compute the same local membership views. After this, all processors leave the group to which they were previously joined, and join the new group. A group is identified by the time of initiation of the 'present' broadcast, in this case $(T + \Delta)$.
Failure Handling: to detect processor failures, all processors periodically broadcast 'present' messages, at local clock times $O = T + \Delta + k\pi$, k being an integer and π a time interval. If at a time $O + \Delta$ a member of processor-group $(T + \Delta)$ does not receive the 'present' message from a member f of $(T + \Delta)$, then —by the atomicity and bounded termination time properties of the synchronous broadcast— no surviving members of $(T + \Delta)$ receive the 'present' message from f. They all conclude that f has crashed, and join a new group with identity (O) with the same members as group $(T + \Delta)$, except for f.
To give effect to the consistent views on processor-group membership, processors only accept messages from members of the group that they have joined, with the exception of 'new-group' messages. In this way, a faulty processor f cannot corrupt correct processors by sending confusing messages.

The objective of the server-group membership (SGM) service is to let each member z of a server-group Z know the membership of the group of correctly functioning Z members. Cristian implements the SGM service as an extension of the processor-group membership service, which implies that an SGM service is running on every processor.
The SGM server maintains a table S of type *(Server, Group, Processor)*; an entry (z, Z, p) from table S to all SGM servers (including itself). When a processor q detects the failure of processor p, the SGM server on q interprets this as the failure of all servers running on p, and broadcasts an update for the removal of all entries with p from S.
When a newly started processor j with its SGM server wants to join an existing group of correct

functioning with their SGM servers, it broadcasts a 'new-group' message at time T. When this message is processed by the processors in the group, they add to the 'present' messages that they send at clock time $T + \Delta$ the values of their local S tables. However, when at time $T + 2\Delta$ processor j receives the 'present' messages with the values for S, these values may be outdated; because updates that were not applied yet to the values initiated between T and $T + \Delta$ and, after receiving the 'present' messages, apply them to S to obtain an up-to-date value. Processor j and its SGM server have now joined the group.

In case a new server z on an already joined processor p wants to join server-group Z, the SGM server on p broadcasts an update to include (z, Z, p) in S to all SGM servers. If z needs the global state of the Z group, this can be established with a protocol that is similar to the protocol described for the join of a new processor with its SGM server. After all, the SGM service is nothing more than a server-group, which just happens to have a server replicated on every processor.

Chapter 2

The Environment

This chapter describes the two important elements of the environments in which the reliable SRT multicast protocol has to work, namely DEDOS and EMPS, the hardware platform for DEDOS. It is beyond the scope of this thesis to give a complete description of the two, only items that are essential to the protocol will be discussed. For a more detailed description of DEDOS and EMPS, the reader is referred to [Stok] and [Dijk], respectively.

2.1 DEDOS

2.1.1 Definitions and nomenclature

The purpose of the DEDOS operating system is to support the real-time execution of application software. For this, DEDOS offers an application language framework supported by methods and tools. The programmer's view of this framework is referred to as the *DEDOS model*. For several reasons, this model is based on the object oriented programming paradigm. As a basis for the programming framework the language DEAL (DEdos Application Language), in fact extended C++, is used. These extensions concern language constructs related to concurrency, real-time and reliability.

In DEDOS an *application* is a collection of (DEAL) *programs*. A program is the unit of compilation. A *system* is the collection of compiled programs, belonging to an application together with a system description file. A system is started in its entirety. When a system is started, a program-instance, called *process*, is created for every program that is loaded. The system description file contains information concerning the processor(s) on which programs are to be loaded. A process is uniquely identified by the program and an instance number. Thus, it is possible to create multiple instances of one program, and to run them on different processors (replication). Also it is possible for a process to import classes or objects from other processes. However, it is required that the latter has exported these classes or objects. By importing classes or objects, a process can use services provided by another process. The process using the services is called a *client*. The process providing these services is called a *server*.

Another administrative unit in DEDOS is an *execution*. An execution is a causally related, actually

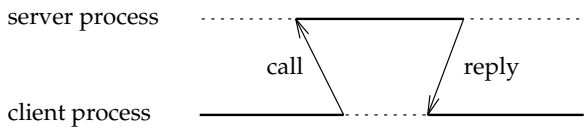

Figure 2.1: Path of one execution going through two processes

executed, sequence of instructions, system wide. For example, when an execution encounters a remote call (a call to an object created by another process), it moves from the client to the server process until control is passed back. This server process may be located on the same or on a different processor. Figure 2.1 illustrates this concept.

Further, within a process, several executions may run (quasi-) concurrently. Within a process, each execution is managed by a separate system entity, called a *thread*. When an execution performs a remote call and thereby enters a server process and no thread is already available for this execution, a new thread in this server is allocated to handle this service request. When a thread executes a remote call or system call (i.e. the execution enters another process), it will be blocked until the execution returns with the reply.

DEDOS distinguishes from other distributed systems, because DEDOS supports two kinds of executions: *Hard Real-Time* (HRT) and *Soft Real-Time* (SRT) executions. HRT deadlines will be guaranteed by DEDOS by executing programs deterministically at times specified by an off-line calculated schedule. A feasible schedule consists of two parts: a global schedule which specifies the allocation of processes to processors, and a collection of local schedules (one for each processor) in which time-slots to access resources are allocated to all threads running on that processor. The calculated schedule is enforced by an *on-line scheduler* on each processor in the system.
Since entire processes are allocated to a processor, all threads within a process reside on the same processor. Threads are executed (quasi) concurrently. HRT threads are dispatched as specified by the off-line calculated schedule. SRT threads can execute in the time not used by HRT executions. They have deadlines, but it is not of vital importance that these deadlines are always met. When a SRT execution exceeds its deadline, it is aborted. They are scheduled dynamically (on-line) by using a preemptive priority scheduling policy. When processor-time is available for SRT, the highest priority ready-to-run thread executes until it suspends or until a higher priority thread becomes ready-to-run. A SRT thread can be preempted by a HRT thread and by a higher priority SRT thread.

HRT executions have to meet their deadlines at all times, therefore it must be guaranteed that a service is provided without interruption despite hardware faults. In DEDOS *active replication* is used to achieve fault tolerance for HRT processes: several copies of the same process are constantly executing on different processors (hot standby or active redundancy). Failing HRT processes may be interrupted by a clock interrupt. A clock interrupt can abort HRT blocks which exceed their allocated time, which points to a processor failure.
SRT processes on the other hand, do not have to meet strict deadlines. Fast and efficient execution is needed under normal circumstances (i.e. no failures), but in case a failure does occur, it is not crucial that services are provided continuously. Therefore, *recovery* and/or *passive replication* are used to achieve fault tolerance for SRT processes. After a process failure has occurred, a backup process is started and the state of the service has to be restored (cold standby or passive redundancy). These principles are also valid when designing a multicast protocol for HRT or SRT processes, respectively.

2.1.2 The DEDOS architecture

In this section, a short overview of the DEDOS architecture will be given. The *hardware platform* for DEDOS, which is EMPS, will be discussed in the next section.

The *kernel* separates the hardware from the other DEDOS layers. The layers above the kernel only see the concepts implemented by the kernel. The kernel manages memory, mailboxes, threads and devices.

Directly op top of the kernel lies the *communication layer*. This layer uses mailboxes and does not directly access the individual processors. It contains protocols for Remote Procedure Call (RPC) communication, multicast communication and membership service.

The *clock synchronisation layer* uses a Clock Synchronisation Algorithm (CSA) to adjust the individual clocks of the processors so that the differences of the clock values are less than a given constant ε. This is needed because the (real-time) execution of a distributed precalculated schedule requires the availability of a central clock or a precise virtual global clock.

The *scheduler layer* contains the dispatcher (scheduler) for both SRT and HRT threads. The execution of requests from HRT threads is strictly determined in time, but data dependencies in the HRT applications impose the presence of more than one possible schedule. The scheduler decides which part of the HRT schedule should be executed.

Concurrency control and recovery of global objects is taken care of in the *object handling layer*. Accesses of SRT threads to global objects occur at unpredictable times, in contrast to accesses of HRT threads, which are prescribed by the HRT schedule. The consistency of the object is maintained for both types of accesses by this layer, just like recovery of SRT threads from stable storage in the case of processor crashes or memory crashes.

2.2 EMPS

2.2.1 The hardware

The hardware of the EMPS system is hierarchically structured as shown in figure 2.2.

The EMPS system consists of *nodes* (or chassis) connected via Ethernet and serial links. Ethernet is used for SRT communication, the serial links for HRT communication. The serial links provide a means of communication with bounded transfer times. Ethernet can only give maximum transfer times in case all communication is scheduled off-line in a way that avoids collisions (processes claim the Ethernet which becomes a resource).

The nodes consist of up to four *clusters* interconnected by a 32-bit VME bus, the *system bus*. The maximum of four clusters per node is due to the capacity of the system bus. Each node has a LAN controller which controls access to the Ethernet and a system controller to solve bus conflicts. This system controller is identical to a 'normal' computer module (see below), except for the presence of additional facilities for the arbitration of the system bus to avoid collisions and ensure fairness of accesses to the bus. A possible node configuration is shown in figure 2.3.

Clusters are composed of up to five modules, which are interconnected by a 32-bit VSB-bus, the *cluster bus*. The maximum on the amount of modules is due to the capacity of the cluster bus. Modules can be computer modules or memory modules.

Computer modules are composed of a 32-bit MC68030 microprocessor, an MC68882 floating point processor, a memory management unit, four 20 Mb/s serial links for HRT communication, a communication register, a real-time clock and 2 Mb of private RAM memory. Each processor runs a local copy of the kernel and the DEDOS operating system contained in its private memory.

The memory modules contain 4 Mb of DRAM (extendable to 64 Mb) with the possibility of byte parity check. It contains a three level arbiter for refresh, cluster bus request and system bus requests.

We see that each cluster constitutes a tightly coupled multiprocessor. A node is a multiprocessor with common memory.

2.2.2 The kernel

The EMPS kernel provides both non-distributed and distributed services. The low-level non-distributed services are present on every processor and are executed locally via *system calls*.

The distributed services are offered for process management and inter-process[1] communication. A (remote procedure) call to one of these services can cause an execution to move to another process (in this case, they execute remotely). This construction implies that all services are offered *transparently* to any process in the system.

The system services needed for implementation of the SRT internode multicast protocol, or closely related to it, are summarised and explained in appendix A.

[1] A process in the EMPS system is similar to a thread in the DEDOS context.

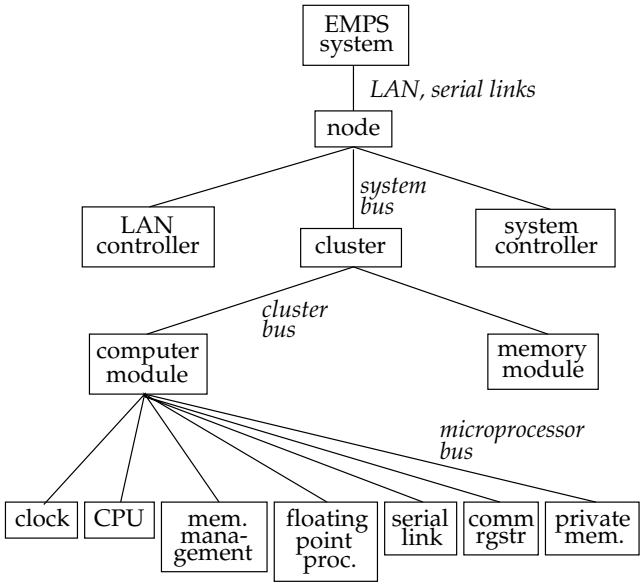

Figure 2.2: The EMPS hardware

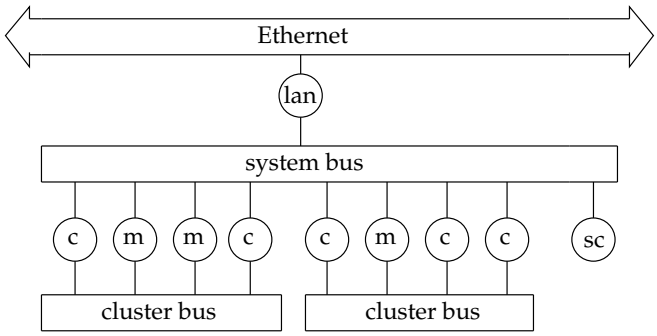

Figure 2.3: A possible node configuration in EMPS

Chapter 3

Reliable SRT inter-node multicast in DEDOS

In this chapter, a conceptual model of reliable inter-node multicast for Soft Real-Time processes in DEDOS is presented. To begin with, the services to be provided to processes in the system are described. Secondly, the requirements for the protocol for multicast communication are formulated. In the third section, the failure classification as proposed by Cristian is summarised, after which the failure hypothesis is stated. Then, the actual protocol is given in pseudo code and a description is given in English. Dynamic group membership will be dealt with in a separate section.

3.1 Services to be provided

In DEDOS, groups of processes, running on a collection of processors, cooperate in some way or another to provide a number of services. Examples of these services are a system management service, a print service, an electronic mail service, a file service, etc. Such a group is called a *multicast group*. Processes of a multicast group from which members execute on different nodes, want to communicate with each other via reliable multicast. They want to send messages to and receive messages from each other. Several multicast groups may coexist in time. Processes can be member of more than one multicast group, so the groups are not necessarily disjoint.
Before multicast communication can take place, a multicast group must be **Create**d. To be able to participate in multicasting within a group, a process must first **Join** the multicast group. A process joins a group either as a *receiver* or as a *sender*. A receiver can only **Receive** messages, a sender can only **Send** messages. When a process no longer needs to receive from or send to a multicast group, it **Leave**s the group. For the sake of flexibility, group membership must be *dynamic*, which means that processes may join and leave groups at any time. When multicasting is no longer needed by a specific group of processes, the multicast group is **Remove**d. A multicast group may only be removed when there are no members left.
The purpose of the reliable SRT inter-node multicast protocol is to offer reliable communication between members of a multicast group executing on processors which reside on different nodes.

3.2 Requirements

The multicast service needs to be offered even in the presence of failures by the protocol presented in this thesis. In other words, the multicast must be *unanimous*, i.e. it has to deliver a message either at every *correct* (i.e. non-failed) receiver, or none at all. In order to exclude the trivial solution of refusing to receive any message, it is also required that it is *reliable*, i.e. if the sender is correct, its messages will be received by every correct receiver.
In addition to reliability and unanimity, the inter-protocol must provide a *multiple source ordering* on all messages. Multiple source ordering on messages means that messages originating from more than one sender are ordered **within** a multicast group. So there is possibly no ordering on messages from different groups. If for instance, processes p and q are both members of the multicast groups **A** and **B**, it is required that p and q receive the messages for **A** in the same order and the messages for **B** in the same order. No statement is made about the interleaving of messages for **A** and **B** as received by p and q. Receiver p may well decide to first receive all message for **A**, and then those for **B**, while q decides to do it the other way around. If the interleaving is the same for all processes, the ordering is called a *total ordering*. This protocol only provides multiple source ordering.
The SRT multicast protocol also has to be **efficient** under normal circumstances. It must occupy resources as short as possible, resources being the processor, memory, the cluster bus, the system bus and the LAN. This way, the SRT multicast protocol is fast and can execute well in the time slots not used by HRT scheduled processes. However, in case a failure occurs, efficiency is sacrificed in order to maintain consensus, validity and ordering requirements. The execution times are unbounded since the protocol is intended for SRT applications.
Summarising, the SRT inter-node multicast protocol presented in this thesis must satisfy the following requirements:

requirement 1, *reliability:*
if a correct sender sends a message for multicast group **A**, all correct receivers of **A** will receive the message,

requirement 2, *unanimity:*
if a correct receiver of multicast group **A** receives a message for **A**, all correct receivers of **A** will receive the message,

requirement 3, *ordering:*
there is a multiple source ordering on the messages.

3.3 Failures

Any protocol, designed for fault-tolerant distributed systems, will have to specify exactly the kind and number of failures it tolerates. Here, this is done as well. The classification is based on the classification of failures as proposed by Cristian in [Cri0].

3.3.1 Failure Classification

Each system component (e.g. processor, system bus, etc.) is required to respond to inputs in a way that is consistent with its specification. This specification prescribes both what output should be produced in response to any sequence of input events and the real-time interval in which the

output should be produced. A system component is called *correct* if its response to inputs is consistent with its specification. Otherwise, a *failure* has occurred; the component is said to have failed. Cristian distinguishes the following type of failures:

An *omission failure* occurs when a component does not respond to an input. An important subclass of omission failures is the class of *crash failures*. If, after a first omission failure, a component systematically fails to respond to subsequent inputs until restart, it is said to suffer a crash failure. Examples of omission failures are a processor crash or a link breakdown. A *timing failure* occurs when a component gives the specified output too early or too late. Most timing failures observed in practice are late timing failures, called *performance failures*. When some coordinated action is taken by a processor too soon (perhaps because a timer was too fast), we speak of an early timing failure.

The following failure class, the class of *Byzantine failures*, is only mentioned to complete the classification of Cristian. They will not be used elsewhere in this thesis. A Byzantine failure occurs when the component does not behave in the manner specified: either no output occurs, the output is outside the real-time interval specified, the output is different from the output specified, or the outputs, in response to one input, distributed to different processes may be different (corrupted output). It is clear that this class of failures is the most general one. A message altered by a processor or a link (because of a random fault) is an example of a Byzantine failure. A special subclass of Byzantine failures contains those failures which are detectable when using some kind of authentication algorithm. These failures are called *authentication-detectable Byzantine failures*. Error detecting codes are an example of authentication techniques which can ensure that failures are detected with high probability. If the authentication protocol employed enables the receiver of the message to detect the alteration, we have an authentication-detectable Byzantine failure.

When a component exhibits only crash failures, it is called *fail-silent*: either its output is correct, or it does not give any output at all, and remains silent until restart.
The classes of crash, omission, timing, authentication-detectable Byzantine and Byzantine failures are, in this order, of increasing generality. Thus, algorithms tolerant of such failures will have increasing cost and complexity in this order.

3.3.2 Failure Hypothesis

The kind and number of failures the protocol can tolerate can now be specified.

- Processors (computer modules) may suffer crash or timing failures. When a processor crash occurs, all processes located on the crashed processor will crash. The kernel, located at some processor, can crash too. This is equivalent to a processor crash. In principle, an arbitrary number of processors can crash.
- Application processes may suffer only crash or timing failures. An arbitrary number of processes may crash. Some process crashes are noticed by the kernel, e.g. division by zero, in which case the membership service is immediately notified of the crash. Processes can also crash without the kernel noticing it, e.g. when a process executes an eternal loop. It is the duty of all DEDOS packages to be alert and notify the membership service of any detected process crash.
- Let G denote the communication network of network processors and the LAN. Let F be a set of processors and the part of the LAN that experiences failures during an execution of the reliable SRT inter-node multicast protocol. Further, let G–F be the surviving network consisting of the remaining correct network processors from G and the remaining correct

part of the LAN to connect them to each other. Now, we assume that G–F is fully connected, i.e. there are no partitions in the surviving network.

- Entire nodes may crash too. This happens when all processors in that node have crashed.
- Common memory modules are assumed to be fail-silent. The memory modules have independent failure probabilities. Memory failures are detectable by processes accessing a module: if a read or write operation on common memory by a process is unsuccessful, the process is notified of a crash (e.g. by means of an exception mechanism). Subsequent reads from any process yield the same result.
- It is assumed that at most f memory modules fail. Thus, if $f + 1$ memory modules are created, it is certain that at least one of them remains correct.
- A system bus, cluster bus or system controller failure causes a node crash.

If the system bus fails, the clusters in the node are separated. Furthermore, the entire node is disconnected from the LAN and thus from all other nodes. The other nodes regard the node as being crashed. The same is true for the system controller, since it regulates access to the system bus. The system bus is highly reliable and the probability that it fails is much smaller than the probability that a computer module fails. The system controller's hardware, however, is identical to that of a computer module. Therefore, it is undesirable that it constitutes a single point of failure. In an operational environment, care should be taken that, after a system controller crash, another computer module can take over its task. Within the context of this thesis, failures of the system/cluster bus and the system controller are treated as node crashes, because in case of a crash of one of them, the node in question is no longer part of the system.

3.4 Assumptions

The following assumptions have to be made to guarantee the correct working of the reliable SRT inter-node multicast protocol:

- There exists a membership service that will notice process and processor crashes.
- Every node has one or more processors that are connected to the LAN. These are the *networked processors*.
- There exist primitives for sending and broadcasting messages over the LAN that are reliable, i.e. when the receiver of the message has crashed, the sending fails, and when the receiver nor the LAN or any backup LAN have crashed, the message will be delivered. Messages that are delivered by the send or broadcast primitives are received in the same order they were sent.
- At most f memory modules may fail per node.
- When the private memory of a computer module crashes, the processor crashes.
- The *mailbox server* controls the creation and removal of mailboxes, and the access of processes to these mailboxes. Every mailbox has a system wide unique name. A process that wants access to a mailbox, performs a call to the mailbox server, which returns a reference to the mailbox. When communication is no longer needed, the process disconnects from the mailbox.

– An assumption made to guarantee efficiency, not correct working, is that multicast groups are rather large. Receivers of multicast groups are often located on many different nodes of the system.

3.5 Design

The protocol for SRT communication between processes of a multicast group, executing on processors which reside on the *same* node, is described in [Verm] and appendix C, and is referred to as the intra-(node) protocol. The protocol for SRT communication between processes executing on processors that reside on *different* nodes, is presented in this chapter, and is referred to as the inter-(node) protocol. The inter-protocol will interact with the existing intra-protocol which has to be adapted to cooperate with the inter-protocol. These adaptations will be presented in this chapter as well.

The EMPS system is built in a hierarchical way. Therefore, we expect to obtain good results using a hierarchical protocol. In the protocol presented in this chapter, there will be a central process at the top, several processes (one per node) that relay messages in the middle, and all senders and receivers at the bottom. In figure 3.1, we can see that a sender uses an adapted version of the reliable SRT intra-node multicast protocol to communicate with a mid-level process which will send the message via the network to the central process at the top level, using the SRT inter-node multicast protocol. The central process will distribute the message to the processes in the middle level. These processes use an adapted version of the SRT intra-node protocol to deliver the message to the local receivers at the bottom level.

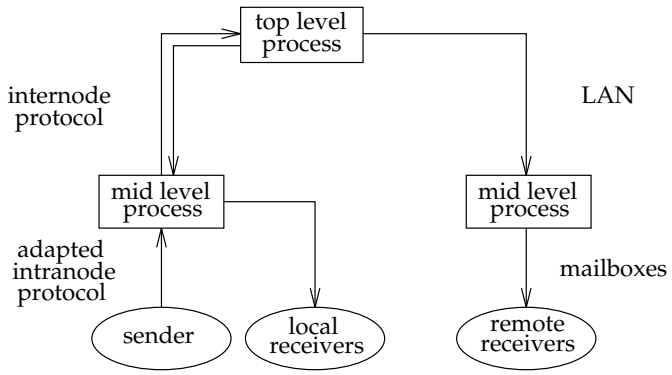

Figure 3.1: The global working of the inter-protocol

The inter-node part of the protocol is based on a protocol described in [Kaa0] and appendix E. This is an asynchronous central-node protocol which puts a total ordering on the messages. It uses unreliable broadcast messages to implement reliable broadcast. Note that the central *node* is in fact a *process*, and not a node like those in the EMPS system.
A total ordering on the messages is not required for the inter-node protocol. Multiple source ordering, which is less restrictive, is enough. Therefore, we use a central node per multicast group instead of just one central node for the whole system. This has several advantages: the performance will be better as there is more load balancing (assuming the different central nodes reside on different processors), and there is no communication bottleneck (assuming the different central nodes reside on different nodes of the EMPS system). Also, when there are several central

nodes, there is no single point of failure.
The protocol described in [Kaa0] implements *reliable broadcast*, while the inter-protocol has to implement *reliable multicast*. Because we assume multicast groups to have receivers on many nodes, using broadcast messages instead of point-to-point messages is still efficient. One broadcast message occupies the LAN for a much shorter time than several point-to-point messages. However, a broadcast message will cause an interrupt on all processors connected to the LAN.

So, each multicast group has a central node. We will call this central node the *distributor* in the remaining part of this thesis. It will broadcast incoming messages for its multicast group to the middle layer, the relaying processes, thereby introducing the ordering. We will call these processes in the middle layer the *intermediates*. There has to be an intermediate for every multicast group on every node. We can combine the intermediates for all multicast groups per node, in which case there will be as many intermediates as there are nodes in the system. This will increase the reliability because there is less chance of an intermediate crashing, but recovery will take more time. Because the inter-protocol is meant for SRT communications, we accept a longer recovery time, and implement one intermediate per node.
An intermediate acknowledges messages it receives from the distributor, asks for retransmissions when necessary and distributes the received messages to the receiving processes on its node using an adapted version of the existing intra-protocol. The intra-protocol uses mailboxes located in common memory for local distribution. Each multicast group has exactly one mailbox per node. So, a member of a multicast group only needs to know the identity of the corresponding mailbox to receive messages from all other members of the multicast group, without having to know how many or which processes are members of this multicast group. The identities of receivers of a multicast group are stored in the corresponding mailbox. Since common memory modules may fail, the mailbox forms a single point of failure that introduces reliability problems. Consider the following situation: a given sender has put a message in the mailbox; then receiver p reads this message from the mailbox. Just when another receiver q wants to read the message, the common memory module, which contains the mailbox, crashes. Receiver p has read the message and receiver q has not, so the unanimity requirement is violated. Therefore, the mailbox must be replicated in different independent memory modules. The intra-protocol uses spare mailboxes on at least f other memory modules. The lost data is recovered in one of the spare boxes once the mailbox has failed. This technique is called *cold standby* or *passive redundancy*.
The protocol presented in this chapter replicates messages on all nodes using mailboxes located in common memory to achieve reliability and unanimity, but also holds a copy of the message in the local memory of the distributor to achieve efficiency; messages that have been missed by intermediates can be retrieved from the distributor. Intermediates therefore do not need to know how to reach other intermediates.
In the existing intra-protocol, messages are stored in the private memory of senders and receivers, in order to be able to rebuild the mailbox in case of a crash. In the inter-protocol, when a mailbox crashes, local senders and receivers may not be able to restore all the messages that were in the crashed mailbox, as some messages may be originating from different nodes, that have not yet been received by any local receiver. The recovery procedure therefore has to be extended so it will restore lost messages that originated from different nodes with cooperation of the distributor and other intermediates.

Summarising, a sender will send a message by putting it in the local mailbox. The intermediate will send it to the distributor who introduces the ordering on the messages before broadcasting them to all intermediates. The intermediates will store the message in the local mailbox for local delivery. When a mailbox crashes, the receivers and intermediate together can retrieve the lost messages from their private memory and the distributor, respectively. When the distributor crashes, it can ask intermediates to retransmit lost messages.

3.6 Protocol Description

This section gives an informal description of the protocol. First, the data structures are presented. Then the communication protocol, consisting of the primitives Send_inter, Intermediate_incoming, Intermediate_outgoing, Distributor and Receive_inter, is described. Application processes can invoke these primitives via system calls, which means that they are executed in the context of the invoking process, except for the Intermediate_outgoing primitive. Also, the measures that are taken to handle process, mailbox and network failures are discussed.
Although it is assumed in this section that there is a fixed set of senders and receivers in each multicast group, the protocol was designed to enable easy extension with dynamic group membership, see section 3.7. As already mentioned, the membership service has all the information about multicast groups and which processes are members (receivers) of them.

3.6.1 Data Structures

The protocol uses message elements that can be stored in mailboxes. Each node has one mailbox and f spare boxes for each multicast group that has senders and/or receivers in that node. For the sake of simplicity and readability, messages sent over the network are of the same type as message elements in the mailboxes. So, messages will have fields that are not used, e.g. a message from an intermediate to a distributor requesting a retransmission has Ackflags. If we assume empty fields to have length zero, this does not decrease efficiency nor performance.

- The mailbox and spare boxes of an arbitrary multicast group **A** consist of the following data:
  - Rlist, the list of PIDs of receivers in **A**
  - Rflags, a list of flags for each receiver in Rlist, used when the mailbox is being rebuilt
  - Qset, the set of message elements in the mailbox
  - Fset, the set of free message elements of Qset
  - Tset, the set of message elements of Qset in transmission

  Fset and Tset are disjoint, and their union is equal to Qset. After a message in Tset has been received by all correct receivers, the protocol will move it from Tset to Fset.
- A message element consists of the following fields:
  - Type, the type of the message, used for communication only
  - MessageId, a message identifier
  - SenderId, the PID of the sender of the message, used for communication only
  - OrigSenderId, the PID of the original sender, used to identify a message together with MessageId
  - SequenceNr, a sequence number for the ordering of the messages
  - Acklist, an array that holds the acknowledgement flags for all local receivers for the message; these flags will be set by receivers when they have received the message, used in mailboxes only
  - AckRecRc, a piggybacked acknowledgement of messages received by (all) receivers, used for communication only
  - Data, the actual message to be sent

The Acklist takes a PID as argument. This enables extension of the protocol with dynamic group membership. If receivers are allowed to join and leave, the Rlists will vary in time, so they cannot be used to determine which receivers must receive and acknowledge a message. Furthermore, the membership service needs the receiver's PIDs in the Acklist to take appropriate action after a receiver crash.

- In the system, the global array Dist holds the PIDs of the distributors for all multicast groups. The procedure NextDist takes the PID of a distributor as argument and will return the PID of the next available distributor. Similarly, the procedure NextIm takes the PID of an intermediate as argument and returns the PID of the next available intermediate. If there is no next distributor or intermediate left, the result value will be $\perp$.
- There exists a global variable ImIds that is the set of all PIDs of the correct intermediates.
- Each sender and receiver stores local references to the mailbox and all spare boxes. These references are called *ports*. The boxes (mailbox and spare boxes) are ordered, and this ordering is the same for every sender and receiver.
  There is no difference in data structure between the mailbox and the spare boxes. The first correct box serves as active mailbox. When a process notices that the mailbox has crashed, it will select the first correct box, and from then on, considers that box to be the mailbox, until that box crashes.
- The distributor maintains a local *history buffer* or Hbuffer of messages to be able to retransmit messages intermediates have missed. The Hbuffer contains message elements as described before.
- Receivers locally store the SequenceNr of the message they have last received, in order to know which message to expect next. They also maintain a Hbuffer of messages they received in order to rebuild the mailbox in case of a mailbox crash.

In the description of the protocol, it is assumed that multicast groups have been created, and that a fixed set of senders and receivers has joined the group.

In the next two subsections, the primitives are described. For ease of understanding, they are described in natural language.
The primitives may not be executed concurrently by different processes, because they might share data located in common memory. Therefore, some form of *mutual exclusion* is required. Apart from the use of common memory, there is no restriction on the execution of the primitives.
At this stage, mutual exclusion is abstracted from. It is assumed that the primitives are executed in such a way, that the result is the same as when they would be executed in a sequential order.
We will also assume there is only one multicast group. The generalisation to more than one multicast group is straightforward.

3.6.2 The Communication Protocol

The protocol consists of five primitives. There are primitives for the sending and receiving of a message. These primitives replace the corresponding primitives of the intra-protocol. Furthermore, there is a distributor primitive which is executed for every multicast group. These distributor primitives can be executed on any networked processor. The remaining two primitives are executed at every node on a networked processor. One of them sends multicast requests to the distributor, the other receives multicasts from the distributor. All primitives, except the ones on the nodes sending multicast requests, are executed when a sender wants to send a message, a receiver wants to receive a message or a message is received from the network. The primitive

that is sending multicast requests is not triggered by incoming messages but executed at regular intervals.

During normal operation, senders will send a message to a multicast group by putting the message in the local mailbox. Senders will also set the appropriate flags and tag the message with their own PID in the OrigSenderId field and a message identifier MessageId. MessageId is an integer that is unique for each sender. So, every message can be uniquely identified by the PID of the sender and the MessageId.
The intermediates check the mailbox in their node at regular intervals to see if there are messages waiting to be multicast. To make messages to be sent distinguishable from messages that are in the mailbox to be received, senders will give messages they put in the mailbox an undefined SequenceNr. If an intermediate finds such messages, it will send them to the distributor.
The distributor will then place the multiple source ordering on incoming messages by giving them SequenceNrs which are increasing, so each message from one distributor will get a unique SequenceNr. So, all messages on the system can be uniquely identified by the combination of either the PID of the sender and the MessageId, or the multicast group identifier and the SequenceNr. The distributor will send the message to the intermediates using the broadcast primitive which makes use of unreliable broadcast provided by the LAN.
The intermediates will store incoming messages in the local mailbox so local receivers can read them, in order of increasing SequenceNrs. Receivers keep the SequenceNr of the message last read in the variable LastSeqNr so they know what message to receive next. A receiver that has read a message will set its Ackflag on the message. When all Ackflags on a message are set, the intermediate can move the message element from Tset to Fset.

As a consequence, the following invariants hold:

- The sequence numbers of the messages in Tset form a consecutive set of integers.
- If the Ackflag for correct receiver p is set on a message m in the mailbox, then the LastSeqNr variable of receiver p is at least m.SequenceNr.
- If for two messages m_1 and m_2 in the mailbox m_1.SequenceNr $\leq$ m_2.SequenceNr holds, then the set of Ackflags that is reset for correct receivers of m_1 is a subset of the set of Ackflags that is reset for correct receivers of m_2. In other words: for each correct receiver p, (m_1.SequenceNr $\leq$ m_2.SequenceNr $\wedge$ m_2.Ackflag.p = set) $\Rightarrow$ (m_1.Ackflag.p = set) holds.

These invariants ensure that an intermediate can determine from the Ackflags on messages which receivers have received which messages, and that messages are removed from the mailbox and the Hbuffer in order of their SequenceNrs.

The protocol just described provides reliable multicast in the absence of failures. However, several components may fail in various ways. We will now extend the protocol to cope with the failures that can occur.

First, messages may be lost on the LAN. If a multicast request from an intermediate to a distributor is lost, the message will not be multicast. When the intermediate checks the mailbox again for messages to be broadcast, the original message will still be in the mailbox and sent to the distributor again. If the message was correctly multicast however, but not received (or too late) by the sending intermediate, the intermediate will resend the message to the distributor while it already has been multicast. In order to prevent messages from being multicast more than once, the distributor keeps a log of all messages it multicasts in its local memory: the Hbuffer. The distributor can now check if a message is a duplicate by looking if there is a message in the Hbuffer with the same sender PID and MessageId. Duplicates will be ignored.
If a broadcast message from the distributor to an intermediate is lost, a problem arises as local receivers will miss it too. Therefore, we let the intermediate check if the message received is the

one that has the expected SequenceNr (one higher than the sequence number of the previous received message). If the SequenceNr is correct, the message will be stored, else the lost message has to be resent to the intermediate in some way.
The distributor has all messages in its Hbuffer, so when an intermediate detects it has missed some messages, it can ask the distributor to retransmit these messages. To keep the protocol simple, the message that was received out of order is discarded by the intermediate. The intermediate will then not only ask for retransmissions of the missed messages, but also of the message it just discarded. This implies that the SequenceNrs of messages an intermediate has available never shows a gap, so the invariant given earlier still holds.

An invariant of the distributor's Hbuffer is, assuming the distributor has a variable NextSeqNr that holds the SequenceNr of the next multicast:

- The SequenceNrs of the messages in the distributor's Hbuffer form a consecutive set of integers, the highest number being NextSeqNr – 1.

Other LAN failures will be discussed when we look at process failures.

Before we cover the last possible communication failure, that between a process and a mailbox, we take a look at the acknowledgements sent from distributors to intermediates and vice versa.
Because common and private memory are both finite, messages in the Hbuffer of the distributor as well as in any mailbox, have to be discarded when possible to make room for new messages. The distributor may safely discard messages from its Hbuffer that are read by all receivers of the multicast. The information concerning which receivers have received a message is collected and distributed in a hierarchical way. Receivers will set an Ackflag on any message they read from the mailbox. When all receivers in a node have set their Ackflag, the intermediate can acknowledge the local reception of the message to the distributor by sending the sequence number of the message. The invariants imply that when a message m is read by all correct receivers in a node, all previous messages (i.e. messages with a lower sequence number) have also been read by all correct receivers in that node. The intermediate therefore can suffice with sending the sequence number of the last message correctly received by all correct local processes. The intermediate will send this sequence number together with multicast requests when possible. This technique is called *piggybacking*. When after some period, there is still no multicast request to be sent to the distributor, the intermediate will send a special message containing the sequence number. This is done to prevent the Hbuffer of the distributor getting full although messages can be removed from the Hbuffer.
The distributor will have an array to store the sequence numbers that are acknowledged this way. New acknowledgements always imply older acknowledgements as messages cannot become unread after they have been read. To prevent failures in case acknowledgements are received out of order, the distributor will always choose the maximum value of an incoming acknowledgement and the stored acknowledgement to be the new value in the array.
The minimum value of the sequence numbers in the array is the last message that is read by all receivers. Also, all previous messages have been received by all receivers. Therefore, all messages with a sequence number up to and including the minimum in the array can be discarded from the distributor's Hbuffer. This sequence number is also sent to the intermediates, piggybacked on multicasts. All intermediates can remove messages from the mailbox with a sequence number up to and including the one being acknowledged. The intermediates will wait until **all** receivers, not just the local ones, have received the message before they remove the message from the mailbox, in order not to violate the requirements. This will be explained later.

New invariants are:

- All correct receivers have read all messages with a SequenceNr equal to or smaller than the minimum value in the distributor's acknowledgement array (AckRecRcArray).

– All correct receivers in a correct node have received all messages with a sequence number equal to or smaller than AckRecRcArray[NodeId].

To complete the enumeration of communication failures, we will examine communication via mailboxes. This form of communication cannot fail as long as the mailbox has not crashed. A mailbox crash is noticed when a process tries to read from or send to a mailbox. A process noticing a mailbox crash will choose a spare box and use it as the new mailbox. Because we have f spare boxes and assume at most f memory modules to fail, such a spare box always exists.
There is a total ordering on mailboxes so every process will choose the same spare box.

Finally, processes can crash. When a processor or a node crashes, all processes on that processor or node crash, so we do not have to separately investigate these cases.
When a receiver crashes, this is eventually detected by the membership service. The membership service will then remove the crashed receiver from the list of receivers (Rlist) that is in the mailbox. New messages that are put in the mailbox will not hold an Ackflag for the crashed receiver. However, there may be messages in the mailbox that have an Ackflag for the crashed receiver that has not yet been set. Therefore, when the membership service finds a receiver to have crashed, it will also set the Ackflag for the crashed receiver on all messages that include an Ackflag for the crashed receiver.

When a sender crashes, we have to make sure that if a receiver has received the message, all others will too. This guarantees unanimity. The protocol presented thus far cannot guarantee unanimity.
There are several adaptations to the protocol that will ensure unanimity. One of them is keeping the receivers from reading the message until the message is available at every node. This solution is not very efficient as it will introduce extra delays, and also extra acknowledgements to be sent back and forth. Another solution is letting all receivers store received messages in a history buffer, and letting the sender store sent messages in a history buffer. The sender has to store the message to achieve reliability; the receivers to achieve unanimity. We will give two examples to demonstrate this.
Suppose senders do not store messages they send in a Hbuffer. Now suppose a message is sent by a sender and received by all intermediates. No receiver has received the message when all mailboxes and the distributor crash. Now, the message is lost forever, and no receiver has received it. This violates the reliability requirement.
Suppose receivers do not store messages they received in a Hbuffer. Consider an EMPS system with two nodes. On each node there are two receivers. Node #1 has a sender which sends a message. This messages is multicast by the distributor but only reaches node #2. On node #2, only the first receiver has received the message when all mailboxes, the distributor and the sender crash. Now, the message is lost forever, while there is only one of the four receivers that has received the message. This violates the unanimity requirement.

The Hbuffers of senders and receivers are not infinite. Therefore, messages have to be removed from the Hbuffers when possible. Messages may be removed when all (correct) receivers have received the message. So, there has to be a way for senders and receivers to find out if a message has been read by all receivers. If a message is read by all receivers, the intermediate will remove the message from the mailbox. So, when the message is removed from the mailbox, senders and receivers can also remove the message from their Hbuffer. But, when a mailbox crashes just after a message has been read by all receivers, and just before processes checked the mailbox, they will recover the message although this is not needed. This causes no problems with respect to receivers, as the intermediate will, when it removes a message read by all local receivers, also removes all older messages. The sender however can never be informed with total certainty its message has been received by all correct receivers, because the intermediate communicates with the sender via the mailbox only, and between the moment the intermediate changes the mailbox

to mark the message as been read by all receivers, and the moment the sender checks the mailbox, the mailbox can always crash.
We adapt the protocol to deal with this problem in the following way: senders will put the message to be sent in all boxes, including the mailbox, with the SequenceNr undefined. When the message is received from the distributor, the intermediate will set the SequenceNrs in all boxes. When the message has been read by all receivers, the intermediate itself can remove the message from the boxes. The sender therefore does not need a Hbuffer anymore, as a message lost due to a mailbox crash does not have to be recovered by a sender as it is already in all spare boxes. Storing the message in all boxes guarantees reliability. We have to adapt the protocol in case of a sender crash: when a sender crashes, its messages have to be removed from all boxes.
This solution to maintain reliability is not very efficient, because senders will have to actively store messages they want to multicast in all mailboxes. But because we use communication via mailboxes this solution is chosen.

One additional problem to be overcome is that during a mailbox recovery, the intermediate may not store messages in the mailbox as this may result in deadlock. To demonstrate this, take the following example. A mailbox has crashed in a node with only one receiver. This receiver has a full Hbuffer. Then, the intermediate stores a message in the mailbox, after which it is full. Now, the receiver cannot receive the new message because its Hbuffer is full, and it cannot recover any more messages from its Hbuffer because the mailbox is full. To overcome this problem, the mailbox is extended with an Rflag for each process in Rlist, which indicates whether or not a receiver is restoring messages from its Hbuffer. While, after a mailbox crash, not all Rflags are reset, the intermediate will not store messages in the mailbox.
The protocol presented in this chapter lets receivers store messages they send and receive in the Hbuffer in their private memory, until the message has been received by all correct receivers in the multicast group. Note that when the Hbuffer of a receiver crashes, the receiver must also have crashed because of the assumptions that were made. Therefore, we do not have to recover Hbuffers of receivers.
When an intermediate crashes, no more messages and acknowledgements will be sent to the distributor, which will not be able to remove messages from its Hbuffer. When a distributor has a full Hbuffer, it starts a *synchronisation phase* or sync phase by sending a special message to the intermediates containing NextSeqNr – 1, the SequenceNr of the message last multicast. During the sync phase no multicast requests will be honoured. The intermediates will instead ask the distributor to retransmit missing messages until they are up-to-date. When they are up-to-date, they send a special message to the distributor. The distributor will wait until all intermediates are up-to-date. During the sync phase, piggybacked acknowledgements will be used to remove messages from the Hbuffer of the distributor.
If the mailbox in a node gets full during the sync phase, the mailbox was chosen too small. When this happens, the sync phase will take extra time to complete. The Hbuffer of the distributor can also get full when it has been chosen too small and messages are dissipated slowly, e.g. because receivers receive at a lower rate than senders send.

New invariants are:

- A message sent by a sender is always available in the local mailbox until all receivers have received it.
- Each correct receiver holds, between the moment it reads a message from the mailbox and the moment that message has been read by all correct receivers in the multicast group, that message in its private memory.

As we have seen, the first invariant is needed to provide reliability: the message a sender wants to multicast will always be available until all correct receivers have received it when the sender remains correct. The second invariant is needed to provide unanimity: when a receiver has read

a message, the message will be available until all correct receivers have received it when the receiver remains correct.

An intermediate crash is noticed by the distributor because without acknowledgements of the intermediate, the Hbuffer of the distributor gets full. If the intermediate does not respond to a message indicating the start of the sync phase, either the LAN has failed, messages have been missed or the intermediate has crashed. When a process has crashed, this will eventually be noticed by the membership service, as all processes regularly send HRT *'present'* messages to the membership service if they are correct. The distinction between a LAN crash or messages being either lost on the LAN or still being processed can never be made. This follows directly from the absence of time limits on the communication for SRT executions.
When an intermediate does not respond and the membership service reports it as having crashed, the distributor will kill the process in case the membership service was wrong (which rarely happens) and start up a new intermediate if there are still networked processors in the node of the crashed intermediate. To start the new intermediate, the distributor will send a special message including the sequence number of the message that was multicast last. If all networked processors have crashed, the node has crashed and the membership service will be informed of that fact.
In the description of the inter-protocol, we will assume primitives to send point-to-point and broadcast messages exist that use backup LANs when necessary. These procedures will diagnose both LAN crashes and slow processes causing messages not to be received in time correctly. With this assumption, the protocol presented here can abstract from the problem of distinguishing between LAN crashes and e.g. slow processes. The send and broadcast primitives might be implemented using component redundancy (sending the message over more than one LAN), or sending the same message several times sequentially in the hope at least one will get through (time redundancy). Another possibility might be to set the interval processes wait for acknowledges in such a way there is a high probability an reply will come before the timer expires. The first solution is inefficient and decreases performance. The second solution may decrease performance when a process waits until the timer expires, even when the reply is received when the timer has not yet expired. We will abstract from the solution that is chosen to implement the send and broadcast primitives and also assume the primitives for sending messages take the appropriate actions when they detect process or LAN crashes.

When a distributor crashes, this is notified by an intermediate sending a multicast request and not receiving the multicast within a certain period of time. Because the protocol makes use of the special send and broadcast primitives, we know this cannot be caused by LAN failures, so the distributor must have crashed. A new distributor then will be chosen by the intermediate.
The distributor has to rebuild its Hbuffer. The messages that were in the Hbuffer are now available at the intermediates. We can extend the protocols in two ways. One possibility is letting intermediates send the contents of their entire mailbox to the distributor. The distributor will receive all messages that were in its Hbuffer, but a lot of them will be sent several times, decreasing the efficiency and causing a high network load. The distributor can also collect the sequence numbers of the messages intermediates have and ask retransmissions from the intermediate that has all messages up to the last message that was in the crashed Hbuffer. The inter-protocol presented in this chapter lets the distributor collect the sequence numbers so it can ask retransmissions from the intermediate that has all required messages. When a distributor is newly started, it will send a message to all intermediates asking for the SequenceNr of the last message they have in their local mailbox. When all intermediates have replied, the intermediate with the highest SequenceNr is asked to send its entire mailbox contents.
When, during this process, messages are missed, the distributor will notice a gap in the sequence numbers and ask the same intermediate to send the contents of its mailbox beginning with the first message that was missed. Using this method, a lot of messages will be sent more than once. The messages that do not arrive in order will simply be discarded. This is not very efficient, but it makes the protocol and invariants easier. Also we assume messages to be missed very rarely.

Note that, in order for this method to be efficient, we assume the message sent using the send primitive arrive in the same order they were sent.

The crash of a distributor might violate the invariant stating that SequenceNrs of the messages in the distributor's Hbuffer form a consecutive set of integers, the highest number being NextSeqNr – 1. The following invariant still holds however (when we assume variables must be explicitly given a value for them to be defined):

- When a distributor's NextSeqNr is unequal to $\perp$, the SequenceNrs of the messages in its Hbuffer form a consecutive set of integers, the highest number being NextSeqNr – 1.

Summarising, when an intermediate crashes, the distributor will start a new one. When a distributor crashes, an intermediate will start a new one. To rebuild the Hbuffer, all intermediates will cooperate. When a mailbox crashes, the local receivers can recover messages they have received. Messages that were sent by local senders are already in the newly chosen mailbox. Other messages that are lost are retrieved from the distributor. When a distributor has a full Hbuffer, multicasting will be suspended until all outstanding messages have been received by all intermediates. When a mailbox is full, intermediates will not accept multicasts until the mailbox contains free message elements. If there are no free message elements available until the distributor's Hbuffer is full, multicasts will not be accepted and the distributor will start a synch phase. This situation can be prevented by choosing the mailboxes not too small.

3.7 Dynamic Group Membership

In the previous sections, it was assumed that there was a fixed set of senders and receivers, which could only decrease as a result of a process crash. Also, it was assumed that the membership service had all information about the multicast groups and their members. In this section, the protocol is extended with two primitives, Join and Leave, that enable dynamic group membership without violating the correctness requirements.

Join:
A process that wants to join a multicast group, must first notify the membership service of this. Then, it asks the mailbox server (see appendix A) for the locations of the boxes. If it is permitted to join the group, it gets the locations of the mailbox and all spare boxes.
After that, the process, if it is a receiver, adds an entry to the Rlists of all boxes, in their normal order. It then sets its LastSeqNr to the highest SequenceNr of messages in the mailbox.

Leave:
If the process that wants to leave is a receiver, is does not have to receive any messages with a SequenceNr higher than its current LastSeqNr. Thus, from the viewpoint of the multicast group, a receiver that leaves the group can be treated the same way as a crashed receiver. The same is true for senders. However, it is possible that a receiver has to participate in a recovery after a mailbox crash, because it might have messages in its Hbuffer that are nowhere else available. To avoid that a leaving receiver has to wait until it is sure that is does not have to participate with a recovery, it already recovers the messages in its Hbuffer into all spare boxes. From then on, the receiver has no bonds with the multicast group anymore, and it executes the DeleteMember primitive, which is also executed by the membership service when a sender or receivers crashes. After that, it notifies the membership service of the leave.

The proof of correctness in appendix B does not include the primitives Join and Leave.

3.8 Pseudo Code

3.8.1 Type Definitions

The following type definitions are used in the pseudo code of the inter-protocol:

- PIDType, represents all possible process identifiers
- TIME, all possible time representations
- data_type, represents all possible message data; NULLDATA is the empty element in data_type
- msg_type, one of MC_REQUEST, ACKS_ONLY, STARTDIST, DISTLOAD, MULTICAST, RETRANS, UPTODATE, SYNCINIT, HIGHEST, NOT_SENT
- SequenceNrType, an integer or $\perp$; $\perp$ is the unicity element of min and max
- FlagType = [set,reset,$\perp$]
- type MessageType = **record**

```
    Type:          msg_type
    SequenceNr:    SequenceNrType
    MessageId:     SequenceNrType
    SenderId:      PIDType
    OrigSenderId:  PIDType
    AckRecRc:      SequenceNrType
    Acklist:       set of FlagType
    Data:          data_type
end
```

3.8.2 Functions

The following functions are used in the pseudo code of the inter-protocol:

- DelayProcess(Δtime), see appendix A
- send(Id,Message), sends Message over the LAN until the message is received; if the recipient has crashed the membership service will be notified of this
- broadcast(Id,Message), broadcasts Message over the LAN until the message is received; if the recipient has crashed the membership service will be notified of this
- myid(), returns the PID of the calling process
- MailboxCrashed, returns a boolean value indicating weather or not the local mailbox has crashed
- ChooseSpareBox, selects the first correct box
- NextDist(Id), selects the next distributor to handle multicasts, where Id is PID of the current distributor

- NextIm(NodeId), selects the next intermediate to handle multicasts in the node with Id NodeId
- Node(Id), gives the identifier of the node on which the process calling this function resides

3.8.3 Notations

The following notations are used in the pseudo code:

- $:\in$
  This operator picks an element from a set. The element will not be removed from the set.
- $(\forall$ x : P(x) : Q(x))
  This evaluates to true iff. for all elements x for which P(x) holds, also Q(x) holds.
- $(\exists$ x : P(x) : Q(x))
  This evaluates to true iff. there is at least one element x for which both P(x) and Q(x) hold.
- $(\iota$ x : P(x) : Q(x))
  This gives the element x for which both P(x)and Q(x) hold. There has to be precisely one such element.
- **on** Interrupt(Event) **do** Q **done**
  This instruction introduces an exception mechanism. Whenever Event happens, the process will execute command list Q, after which normal operation is resumed.
- $\oplus$ and $\ominus$ are the set-wise addiction and substraction

3.8.4 Pseudo Code of the Send_inter Procedure

This procedure is called with the data to be sent and the multicast group identifier as parameters by any process that wants to send a multicast message.

```
1.  procedure Send_inter(data: data_type)

2.  var
3.    m : MessageType
4.    MessageId : SequenceNrType

5.  begin
6.    on Interrupt(MailboxCrash)
7.    do
8.      ChooseSpareBox
9.    done

10.   m.Data := data
11.   MessageId := SetFlags(m)
12.   StoreMessage(m)
23. end
```

The SetFlags procedure fills the flags of a message for the sender and returns the appointed MessageId. Its parameters is the message the flags have to be set for.

```
1.  procedure SetFlags(m: MessageType): SequenceNrType

2.  /* NextSeqNr is a global variable of type SequenceNrType per sending process */

3.  var
4.    Id : PIDType

5.  begin
6.    m.OrigSenderId := myid()
7.    m.MessageId := NextSeqNr
8.    NextSeqNr := NextSeqNr + 1
9.    forall Id in Rlist
10.   do
11.     m.Acklist.Id := reset
12.   done
13.   /* give message undefined SequenceNr so the intermediate knows
            this message has to be multicast
14.   */
15.   m.SequenceNr := ⊥
16.   return m.MessageId /* return appointed MessageId */
17. end
```

The StoreMessage procedure will wait until there is a free message element in Fset to store m in, and the mailbox is not being recovered. The message m is given as parameter.

```
1.  procedure StoreMessage(m : MessageType)

2.  /* TimeOutSend is the interval the procedure waits for a free message element */

3.  var
4.    l : MessageType

5.  begin
6.    while Fset = ∅ ∨ ¬(∀ Id : Id ∈ Rlist : Rflag.Id = set)
7.    do
8.      DelayProcess(TimeOutSend)
9.    done
10.   /* there is a free message element and the mailbox is not being restored */
11.   forall boxes in reverse order
12.   do
13.     l :∈ Fset
14.     Fset := Fset ⊖ {l}
15.     l := m
16.     Tset := Tset ⊕ {l}
17.   done
18. end
```

3.8.5 Pseudo Code of the Intermediate_outgoing Procedure

This procedure constantly checks for messages to be multicast which will be sent to the distributor. It is run on a networked processor.

```
1.  procedure Intermediate_outgoing()

2.  /* TimeOutAcks is the maximum time interval between communication from
           the intermediate to the distributor
3.     TimeOutSend is the interval the procedure waits for correct reception of the message
4.     AcksTime is a global variable of type TIME per processor
5.     RecSet is a set of PIDType × SequenceNrType
           that is global per processor
6.     SyncPhase is a global boolean variable per processor
7.     LastSeqRecRc is a vriable of type SequenceNrType per processor
8.     Dist is a global varibale that holds the PID of the distributors for the multicast group
9.  */

10. var
11.   SendSet : set of PIDType × SequenceNrType
12.   s : set of MessageType
13.   m : MessageType

14. /* initialisation */
15. SyncPhase := false
16. LastSeqRecRc := ⊥
17. Dist := PickDist()

18. begin
19. on Interrupt(MailboxCrash)
20.   do
21.     ChooseSpareBox
22.   done

23. while true do
24.   if (∀ Id : Id ∈ Rlist : Rflag.Id = set) →
25.     if ¬SyncPhase →
26.       s := {x ∈ Tset | x.SequenceNr = ⊥}
27.       if s = ∅ →
28.         if GetTime() – AcksTime > TimeOutAcks →
29.           /* send acks only */
30.           m := SetAcks(m)
31.           m.Data := NULLDATA
32.           m.Type := ACKS_ONLY
33.           send(Dist,m)
34.         fi
35.       else
36.         /* store sent messages to be able to check if they are multicast in time */
37.         SendSet := ∅
38.         forall m in s
39.         do
40.           m := SetAcks(m)
41.           m.Type := MC_REQUEST
```

```
42.          send(Dist,m)
43.          /* keep identifier of message sent */
44.          SendSet := SendSet ⊕ (m.OrigSenderId,m.MessageId)
45.        done
46.        /* store time at which last communication to the distributor took place */
47.        AcksTime := GetTime()
48.        DelayProcess(TimeOutSend)
49.        if (SendSet ⊖ RecSet) ≠ ∅ →
50.          /* Dist crashed, we're not in sync phase */
51.          Dist := PickDist()
52.        fi
53.        SendSet := SendSet ⊖ RecSet
54.        RecSet := ∅
55.      fi
56.    else
57.      /* we're in sync phase */
58.      if GetTime() – AcksTime > TimeOutAcks →
59.        m := SetAcks(m)
60.        m.Data := NULLDATA
61.        m.Type := ACKS_ONLY
62.        send(Dist,m)
63.      fi
64.    fi
65.  fi
66. end
```

The SetAcks procedure takes a message m as parameter and sets the acknowledgement flags the distributor needs before it returns the message.

```
1.  procedure SetAcks(m: MessageType): MessageType

2.  /* LastSeqRecRc is a global variable of type SequenceNrType, per processor */

3.  var
4.    RecRcSet : set of MessageType
5.    RecRcSeqSet : set of SequenceNrType

6.  begin
7.    /* calculate new LastSeqRecRc */
8.    RecRcSet := {x ∈ Tset | (∀ p : p ∈ Rlist : x.Acklist.p = true)}
9.    if RecRcSet ≠ ∅ →
10.     RecRcSeqSet := {x | (∃ m1 : m1 ∈ Tset : (∃ m2 : m2 ∈ Tset :
                  x = m1.SequenceNr ∧ (x–1 = m2.SequenceNr ∨ x–1 = LastSeqRecRc)))}
11.   LastSeqRecRc := max{RecRcSeqSet}
12.   fi
13.   m.AckRecRc := LastSeqRecRc
14.   m.SenderId := myid()
15.   return m
16. end
```

The PickDist procedure will start up a new distributor.

```
1.  procedure PickDist()
```

```
2.  var
3.    m : MessageType

4.  begin
5.    Dist := NextDist(Dist)
6.    /* there is still a distributor, otherwise there would not be any networked processors
            left in this node, and there would be no intermediate
7.    */
8.    m.Data := NULLDATA
9.    m.Type := STARTDIST
10.   m.SenderId := myid()
11.   m := SetAcks(m)
12.   send(Dist,m)
13. end
```

3.8.6 Pseudo Code of the Intermediate_incoming Procedure

This procedure is called by the intermediate with message m as parameter when a message is received from the network.

```
1.  procedure Intermediate_incoming(m: MessageType)

2.  /* IncomingSet is a set of PIDType × SequenceNrType
          that is global per processor
3.     NextSeqNr is a global variable of type SequenceNrType per processor,
          initially ⊥
4.     SyncPhase is a global boolean per processor
5.     TimeOutRecover is the time waited for the mailbox to be recovered
6.     RecSet is a set of PIDType × SequenceNrType
          that is global per processor
7.  */

8.  var
9.    l : MessageType
10.  s : set of MessageType
11.  LastToReceive: SequenceNrType

12.begin
13.  on Interrupt(MailboxCrash)
14.  do
15.   ChooseSpareBox
16.   repeat
17.    DelayProcess(TimeOutRecover)
18.   until (∀ Id : Id ∈ Rlist : Rflag.Id = set)
19.   NextSeqNr := max{x.SequencNr | x ∈ Tset} + 1
20.  done

21.  /* always check the acknowledgements on the message */
22.  CheckAckIm(m)
```

```
23. case m.Type of
24. NOT_SENT →
25.   /* distributor has not crashed but cannot process message */
26.   RecSet := RecSet ⊕ (m.OrigSenderId,m.MessageId)

27. HIGHEST →
28. /* distributor wants to know what message I received last */
29. m.SequenceNr := NextSeqNr – 1
30. m.SenderId := myid()
31. send(m.SenderId, m)

32. MULTICAST || RETRANS →
33. /* a multicast message has been received */
34.   RecSet := RecSet ⊕ (m.OrigSenderId,m.MessageId)
35.   if m.SequenceNr > NextSeqNr →
36.     /* broadcasts have been missed */
37.     Retrans(NextSeqNr,m.SequenceNr)
38.   else
39.     /* we have received the right message */
40.     if Rlist ≠ ∅ →
41.       if Node(m.OrigSenderId) ≠ Node(myid()) →
42.         /* message has to be stored in the mailbox */
43.         if Fset = ∅ ∨ ¬(∀ Id : Id ∈ Rlist : Rflag.Id = set)
44.           /* no place or mailbox being recovered, discard message */
45.           skip
46.         else
47.           StoreMsgIm(m))
48.           if SyncPhase →
49.             /* was this the last message during the sync phase? */
50.             CheckEndSync(m, LastToReceive)
51.           fi
52.         fi
53.       else
54.         /* message is from a local sender and already in mailbox
55.            set m.SequenceNr in all boxes
56.         */
57.         forall boxes in reverse order
58.         do
59.           l := (ι x : x ∈ Tset : x.OrigSenderId = m.OrigSenderId ∧
                      x.MessageId = m.MessageId)
60.           Fset := Fset ⊖ {l}
61.           l.SequenceNr := m.SequenceNr
62.           Fset := Fset ⊕ {l}
63.         done
64.       fi
65.     fi
66.   fi

67. SYNCINIT →
68.   /* distributor has started the sync phase or I am a new intermediate */
69.   SyncPhase := true
70.   LastToReceive := m.SequenceNr
71.   /* am I a new intermediate? */
72.   if NextSeqNr = ⊥ →
```

```
73.    /* we have received all multicasts up to the one
                with the highest SequenceNr in Tset or m.AckRecRc
74.    */
75.    NextSeqNr := max{max{x.SequenceNr | x ∈ Tset}, m.AckRecRc} + 1
76.   fi
77.   if NextSeqNr = LastSeqToReceive →
78.    /* we're already up-to-date */
79.    CheckEndSync(m, LastSeqToReceive)
80.   else
81.    /* we need missed messages to be retransmitted */
82.    Retrans(NextSeqNr,m.SequenceNr)
83.   fi

84.  DISTLOAD →
85.   /* request to send mailbox after a new distributor started */
86.   forall i : m.SequenceNr ≤ i ≤ NextSeqNr – 1
87.   do
88.    l := (ι m : m ∈ Tset : m.SequenceNr = i)
89.    l.Type := DISTLOAD
90.    l.SenderId := myid()
91.    send(m.SenderId,l)
92.   done

93.  esac
94.end
```

The procedure CheckAcksIm has parameter m and uses the flags on this message to update the corresponding mailbox.

```
1.   procedure CheckAcksIm(m: MessageType)

2.   var
3.    s : set of MessageType

4.   begin
5.    s := {x ∈ Tset | x.SequenceNr ≤ m.AckRecRc}
6.    forall m ∈ s
7.    do
8.     if Node(m.OrigSenderId) = Node(myid()) →
9.      /* message is from local sender, remove from all boxes */
10.     forall boxes in reverse order
11.     do
12.      Fset := Fset ⊖ {m}
13.      Tset := Tset ⊕ {m}
14.     done
15.    else
16.     /* remove message from mailbox only */
17.     Tset := Tset ⊖ {m}
18.     Fset := Fset ⊕ {m}
19.    fi
20.   done
21.  end
```

The procedure Retrans will, when called with parameters From and To, ask the distributor to retransmit all messages with SequenceNrs from From up to and including To.

```
1.  procedure Retrans(From,To: SequenceNrType)

2.  var
3.    i : SequenceNrType
4.    m : MessageType

5.  begin
6.    forall i: From ≤ i ≤ To
7.    do
8.      m.SenderId := myid()
9.      m.Type := RETRANS
10.     m.Data := NULLDATA
11.     m.SequenceNr := i
12.     send(Dist,m)
13.   done
14. end
```

The procedure StoreMsgIm will store a message in the mailbox and adjust the appropriate flags.

```
1.  procedure StoreMsgIm(m: MessageType)

2.  /* precondition: there is a message element in Fset */

3.  var
4.    l : MessageType
5.    p : PIDType

6.  begin
7.    l :∈ Fset
8.    Fset := Fset ⊖ {l}
9.    l := m
10.   forall p in Rlist
11.   do
12.     l.Ackflag.p := reset
13.   done
14.   Tset := Tset ⊕ {l}
15. end
```

A message and the sequence number of the last message to be received in the sync phase are passed to the CheckEndSync primitive which will check whether or not the message was the last to be received. If it was, the distributor is notified that this intermediate is up-to-date.

```
1.  procedure CheckEndSync(m: MessageType, LastToReceive: SequenceNrType)

2.  var
3.    m : MessageType

4.  begin
5.    if m.SequenceNr = LastToReceive →
```

```
6.    /* we're up-to-date again */
7.    m.SenderId := myid()
8.    m.Data := NULLDATA
9.    m.Type := UPTODATE
10.   send(Dist,m)
11.   SyncPhase := false
12.  fi
13. end
```

3.8.7 Pseudo Code of the Distributor Procedure

This procedure is called by the distributor with incoming message m as argument.

```
1.  procedure Distributor(m: MessageType)

2.  /* TimeOutStartDist is a time interval needed to get an acknowledgement
          from all intermediates
3.     DistLoadTime is a time interval needed to recover the Hbuffer after a crash
4.     NextSeqNr is a global variable of type
          SequenceNrType per processor
5.     TimeOutStartDist is the interval waited for present messages of intermediates
          after the StartupPhase has started
6.     From is a global variable of type PIDType per processor
7.     StartupPhase and SyncPhase are initially false
8.  */

9.  var
10.  StartupPhase, SyncPhase : boolean
11.  s : set of MessageType
12.  SyncAckArray : array [ImIds] of SequenceNrType
13.  SyncAckArray : array [ImIds] of boolean
14.  Id, From : PIDType
15.  LastSeqToRec : SequenceNrType

16. begin
17.  /* always use acks to update information about which messages
          have been received by all receivers
18.  */
19.  UpdateAcks(m)
20.  if SyncPhase →
21.   SyncAckArray[m.SenderId] := true
22.  fi
23.  case m.Type of
24.  ACKS_ONLY →
25.   /* acks have already been processed */

26.  HIGHEST →
27.   /* message containing highest SequenceNr or messages received,
          used in startup phase
28.   */
```

```
29.   StartAckArray[m.SenderId] := m.SequenceNr

30.  MC_REQUEST →
31.   if {x ∈ Hbuffer |
32.         x.OrigSenderId = m.OrigSenderId ∧ x.MessageId = m.MessageId} ≠ ∅ →
33.    /* no duplicate */
34.    if SyncPhase ∨ StartupPhase →
35.     NoSend(m)
36.    elseif Hbuffer_is_full →
37.     NoSend(m)
38.     SyncPhase(m, SyncAckArray)
39.    else
40.     /* message can be multicast */
41.     MultiCast(m)
42.    fi
43.   fi

44.  RETRANS →
45.   From := m.SenderId
46.   m := (ι x : x ∈ Hbuffer : x.SequenceNr = m.SequenceNr)
47.   m.Type := RETRANS
48.   m.SenderId := myid()
49.   send(From,m)

50.  UPTODATE →
51.   SyncAckArray[m.SenderId] := true

52.  STARTDIST →
53.   if ¬StartupPhase →
54.    /* I'm a new distributor */
55.    StartupPhase := true
56.    forall i ∈ ImIds
57.    do
58.     StartAckArray[i] := false
59.    done
60.    StartAckArray[m.SenderId] := true
61.    repeat
62.     /* ask intermediates which message they received last */
63.     m.Type := HIGHEST
64.     m.Data := NULLDATA
65.     m.SenderId := myid()
66.     broadcast(m)
67.     DelayProcess(TimeOutStartDist)
68.     s := {Id ∈ ImIds | ¬StartAckArray[Id]}
69.     if s = ∅ →
70.       /* all intermediates replied */
71.       AckDistLoad()
72.       /* NextSeqNr and LastSeqToRec are set */
73.     else
74.      forall Id in s
75.      do
74.       CheckIm(Id, StartAckArray)
75.      done
76.     fi
```

```
77.     DelayProcess(DistLoadTime)
78.     until s = ∅ ∧ NextSeqNr = LastSeqToRec + 1
79.     /* Hbuffer is restored, all nodes have an intermediate */
80.     StartupPhase := false
81.    fi

82.  DISTLOAD →
83.    if m.SequenceNr = NextSeqNr →
84.     /* expected message, Hbuffer not full */
85.     Hbuffer := Hbuffer ⊕ {m}
86.     NextSeqNr := NextSeqNr + 1
87.    else
88.     /* DISTLOAD failed */
89.     AskDistLoad()
90.    fi

91.  esac
92. end
```

The UpdateAcks procedure takes a message as argument and calculates new acknowledgements given the acknowledgements on the message.

```
1.  procedure UpdateAcks(m: MessageType)

2.  /* AckRecRcArray is a global array per distributor of type SequenceNrType
3.     AckRecRc is a global variable per distributor of type SequenceNrType
4.  */

5.  begin
6.    AckRecRcArray[m.SenderId] := m.AckRecRc
7.    m.AckRecRc := min{AckRecRcArray}
8.    /* messages read by all receivers can be removed from Hbuffer */
9.    Hbuffer := Hbuffer ⊖ {x ∈ Hbuffer | x.SequenceNr < m.AckRecRc}
10. end
```

The SyncPhase procedure informs the intermediates the SyncPhase has started.

```
1.  procedure SyncPhase(m: MessageType, SyncAckArray: array [ImIds] of boolean)

2.  /* TimeOutSyncAck is a time interval needed to get acks from all intermediates
3.     NextSeqNr is a global variable per distributor of type SequenceNrType
4.  */

5.  var
6.    Id : PIDType

7.  begin
8.    forall Id in ImIds
9.    do
10.     SyncAckArray[Id] := false
11.   done
12.   m.Type := SYNCINIT
```

```
13.  m.Data := NULLDATA
14.  m.SenderId := myid()
15.  m.SequenceNr := NextSeqNr – 1
16.  broadcast(m)
17.  repeat
18.   DelayProcess(TimeOutSyncAck)
19.   s := {Id ∈ ImIds | ¬SyncAckArray[Id]}
20.   forall Id in s
21.   do
22.    CheckIm(Id, SyncAckArray)
23.   done
24.  until s = ∅
25.  /* every intermediate up-to-date, end SyncPhase */
26.  SyncPhase := false
27. end
```

The procedure CheckIm will take a PID as parameter and check what component is failing: the intermediate that has the given PID of the Ethernet. It will update either SyncAckArray or StartupAckArray.

```
1.  procedure CheckIm(Id: PIDType, Array: array[ImIds] of boolean)

2.  /* NextSeqNr is a global variable per distributor of type SequenceType
3.     ImIds is a global set of PIDs of intermediates
4.  */

5.  var
6.   NewId : PIDType
7.   m : MessageType

8.  begin
9.   ask_membership_service(Id)
10.  if Id crashed →
11.   ImIds := ImIds ⊖ {Id}
12.   NewId := NextIm(Node[Id])
13.   if NewId = ⊥ →
14.    /* no more networked processors in node */
15.    notify_membership_service_of_nodecrash(Node[Id])
16.    Array[Id] := true
17.   else
18.    ImIds := ImIds ⊕ {NewId}
19.    m.Type := SYNCINIT
20.    m.Data := NULLDATA
21.    m.SequenceNr := NextSeqNr – 1
22.    m.SenderId := myid()
23.   fi
24.  fi
25. end
```

The procedure NoSend will reply to an intermediate so the intermediate will know the distributor did not crash but can not accept the multicast request.

```
1.  procedure NoSend(m: MessageType)
```

```
2.  var
3.    From : PIDType

4.  begin
5.    From := m.SenderId
6.    m.Type := NOT_SENT
7.    m.Data := NULLDATA
8.    m.SenderId := myid()
9.    send(From,m)
10. end
```

The procedure MultiCast will do the actual broadcasting of the message for which a multicast request came in.

```
1.  procedure MultiCast(m: MessageType)

2.  /* NextSeqNr is a global variable of type SequenceNrType per distributor */

3.  begin
4.    m.SequenceNr := NextSeqNr
5.    NextSeqNr := NextSeqNr + 1
6.    Hbuffer := Hbuffer ⊕ {m}
7.    m.SenderId := myid()
8.    m.Type := MULTICAST
9.    broadcast(m)
10. end
```

The AckDistLoad procedure will ask the intermediate with the most messages in its local mailbox to restore the Hbuffer of the distributor after a distributor crash. The return value is the SequenceNr of the last message that has to be retrieved.

```
1.  procedure AckDistLoad()

2.  /* StartAckArray is an array of type SequenceNrType that is global per distributor
3.     NextSeqNr and LastSeqToRec are global variables of type SequenceNrType
4.  */

5.  var
6.    DistLoadIm : PIDType
7.    m : MessageType

8.  begin
9.    NextSeqNr := (min{StartAckArray} + 1) max NextSeqNr
10.   LastSeqToRec := max{StartAckArray}
11.   /* choose an intermediate that has all messages needed */
12.   DistLoadIm :∈ {Id ∈ ImIds | StartAckArray[Id] = LastSeqToRec}
13.   m.Type := DISTLOAD
14.   m.Data := NULLDATA
15.   m.SenderId := myid()
16.   m.SequenceNr := FirstSeqToRec
17.   send(DistLoadIm,m)
```

```
18.  return LastSeqToRec
19. end
```

3.8.8 Pseudo Code of the Receive_inter Procedure

This procedure is run by processes who want to receive from a multicast group. It returns the first available message-data.

```
1.  procedure Receive_inter(): data_type

2.  /* TimeOutReceive is the interval the procedure waits for a new message
3.     LastSeqNr is a global variable per receiver
4.  */

5.  var
6.   m: MessageType
7.   i : SequenceNrType

8.  begin
9.   on Interrupt(MailboxCrashed)
10.  do
11.   Recover()
12.  done

13.  while {x ∈ Tset | x.SequenceNr ≥ LastSeqnr + 1} = ∅
14.  do
15.   DelayProcess(TimeOutReceive)
16.  done
17.  m := (ι x : x ∈ Tset : x.SequenceNr = LastSeqNr + 1)
18.  Hbuffer := Hbuffer ⊕ {m}
19.  forall m in {x ∈ Tset | x.SequenceNr ≤ LastSeqNr + 1}
20.  do
21.   m.Ackflag.myid() := set
22.  done
23.  i := min{ x.SequenceNr | x ∈ Tset }
24.  Hbuffer := Hbuffer ⊖ { x ∈ Hbuffer | x.SequenceNr < i }
25.  LastSeqNr := LastSeqNr + 1
26.  return m.Data
27. end
```

The Recover procedure will recover messages in a receiver's Hbuffer into a newly chosen mailbox.

```
1.  procedure Recover()

2.  var
3.   l,m : MessageType

4.  begin
5.   ChooseSpareBox()
6.   forall m ∈ Hbuffer
7.   do
```

```
8.      l :∈ Fset
9.      /* only restore messages not restored by others */
10.     if {x ∈ Tset | x.SequenceNr = l.SequenceNr} = ∅→
11.       Fset := Fset ⊖ {l}
12.       l := m
13.       l.Ackflag.myid() := set
14.       Tset := Tset ⊕ {l}
15.     fi
16.   done
17.   /* done restoring mailbox */
18.   Rflag.myid() := set
19. end
```

3.8.9 Pseudo Code of the Join Procedure

This procedure is executed by a processor that wants to join a multicast group, either as a sender or a receiver. Its argument is the PID of the calling process.

```
1.  procedure Join(Id: PIDType)

2.  var
3.    Id : PIDType

4.  begin
5.    notify_the_membership_service_of_the_Join
6.    if Id is a sender→
7.      forall boxes
8.      do
9.        port := Connect(key)
10.     done
11.   else /* receiver */
12.     forall boxes
13.     do
14.       port := Connect(key)
15.       Rlist := Rlist ⊕ {Id}
16.       if box is mailbox →
17.         LastSeqNr := max{x.SequenceNr | x ∈ box}
18.         Rflag.Id := set
19.       else
20.         Rflag.Id := reset
21.       fi
22.     done
23.   fi
24. end
```

3.8.10 Pseudo Code of the Leave Procedure

This procedure is called by an arbitrary process that wants to leave a multicast group, and takes as argument the PID of the calling process.

```
1.  procedure Leave(Id: PIDType)

2.  var
3.    l,m : MessageType
4.    Id : PIDType

5.  begin
6.    if Id is a receiver →
7.      forall spare boxes
8.      do
9.        forall m ∈ Hbuffer
10.       do
11.         l :∈ Fset
12.         forall Id ∈ Rlist
13.         do
14.           Ackflag.Id := reset
15.         done
16.         Ackflag.myid() := set
17.       done
18.     done
19.   fi
20.   DeleteMember(Id)
21.   Disconnect(ports)
22.   notify_the_membership_service_of_the_Leave
23. end
```

3.8.11 Pseudo Code of the DeleteMember Procedure

The DeleteMember procedure is called by the Leave procedure and by the membership service for a leaving process, which PID is given as parameter.

```
1.  procedure DeleteMember(Id: PIDType)

2.  var
3.    m : MessageType

4.  begin
5.    /* kill process just in case membership service was wrong */
6.    KillProcess(Id)
7.    if Id is a receiver →
8.      forall boxes in reverse order
9.      do
10.       if box is mailbox →
11.         forall m in Tset
12.         do
13.           m.Ackflag.Id := set
14.         done
15.       fi
16.     done
17.     Rlist := Rlist ⊖ {Id}
18.   fi
19. end
```

Chapter 4

Conclusion

In this chapter, a summary is given of the work presented in this thesis. Some suggestions are made to extend the protocol.

4.1 Summary

In this thesis, a multicast protocol is presented for Soft Real-Time applications. It enables a sending process to reliably send messages to a collection of receiving processes, that can be anywhere on the system. The protocol is hierarchical: the senders and receivers communicate with an intermediate that is located in their own node, and intermediates communicate with a distributor. The protocol is tolerant of any number of process, processor or LAN crashes, and of a maximum number of mailbox crashes.
The protocol is designed in a way it that the existing intra-protocol only has to be slightly adapted. This is achieved by replicating the mailbox at every node.
The protocol shows no abnormal delivery latency as long as the mailboxes and distributor remain correct, and the local history buffer of the distributor is not full. As soon as a message is written into a mailbox by an intermediate, it can be read by receivers. During recovery after a mailbox or distributor crash, or during the synchronisation phase to flush the distributor's history buffer, some delivery latency will occur.
The protocol does not guarantee bounded termination time but works efficiently in terms of the amount of messages needed for a multicast and network load. Multiple processes, using the protocol to communicate within a multicast group, may access the mailbox concurrently. Therefore, some kind of mutual exclusion is required to ensure that the contents of the mailbox remains consistent.
The protocol assumes the existence of primitives for sending or broadcasting messages on the LAN that can detect LAN or process crashes and, with high probability, distinguish between them to take the appropriate actions.
Invariants are formulated that suggest the correctness of the protocol. A more formal proof of correctness is given in appendix B.

4.2 Suggestions

- The protocol presented in this chapter assumes there is only one multicast group. When the same primitives will be used for all multicast groups, there will be several intermediate processes per node. It is more efficient to have one Intermediate_incoming and one Intermediate_outgoing process per node.
- When messages are missed by an intermediate, the intermediate will notice this when it receives the next message from the distributor. It will discard this message and request retransmissions of all missed messages including the one it just discarded. It would be more efficient if the intermediate would temporarily store the message in its private memory.
- Likewise, when a new distributor is started, it will ask an intermediate to send all messages it has in its mailbox. When the distributor misses messages, it will discard all subsequent messages and ask retransmissions of all messages starting with the first message that was missed. The distributor could also store all messages and only ask retransmissions of the missed messages. This increases the efficiency.
- Each multicast group has a distributor. To balance the processor and network load, the protocol could be extended to distribute the distributors over the nodes.
- The inter-protocol implicitly assumes each multicast group to have a mailbox on each node. To reduce the unnecessary use of common memory, the protocol could be extended in such a way that it would only keep mailboxes for multicast groups in nodes on which senders or receivers for that multicast group reside.

Appendix A

EMPS Kernel Services

The EMPS kernel, which is part of the EMPS operating system, is used to separate the rest of DEDOS from the hardware. The kernel consist of several parts: memory management, process management, process synchronisation, time management, device management and inter process communication. Each part of the kernel offers some services via system calls or remote procedure calls (RPC's). We will now take a brief look at these parts.

Memory Management

In EMPS, physical memory is subdivided into *pages* of 1 kbyte *(Pagesize)*. At each processor, the kernel maintains a list of free pages of its private memory, the *FreePrivateList*. This list is stored in the private memory of the corresponding processor. Equally, for each node, a list of free pages of common memory, *FreeCommonList*, is maintained. This list is stores in common memory.
The following services are provided for allocating and releasing a page of memory from the list of free pages, which can be invoked with a system call:

- **ptr = Allocate(memory_type, size)**
- **Release(ptr)**

The **Allocate** primitive removes **size** pages (each *Pagesize* kbytes) from either the *FreePrivateList* or the *FreeCommonList*, indicated by **memory_type**. This primitive returns the virtual address of the address **ptr** to the corresponding list of free memory pages.
These primitives are mutually exclusive, i.e. multiple processes cannot access the lists of free pages simultaneously. For the *FreePrivateList* and the *FreeCommonList* this is differently implemented.

Process Management

Processes are sequential programs which can run concurrently on a processor in EMPS. Process management has the task to add and remove processes to and from the system, to start the

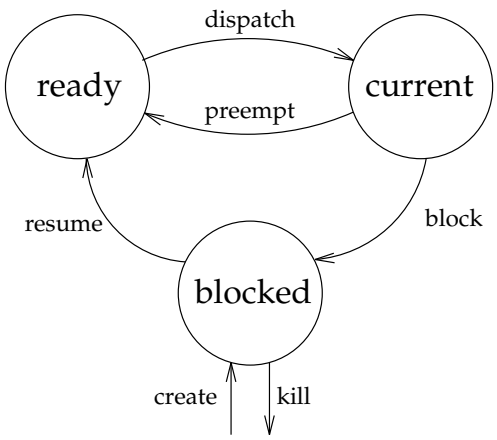

Figure A.1: Process states in the EMPS kernel and the transitions between them

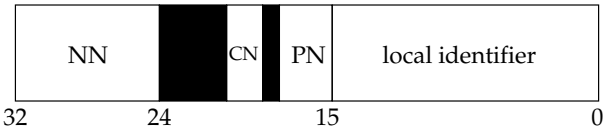

Figure A.2: The process identifier in the EMPS kernel

execution of a process and to allocate the CPU to a process according to some scheduling rules. All processes in the EMPS system are memory resident. Of course at any point in time, at most one process can be allocated to the CPU. This process is referred to as CURRENT. A process can change from ready to current when of all highest priority processes, it is ready for the longest time. The possible states and state changes are shown in figure A.1.

In the EMPS kernel, processes are identified by a 32-bit globally unique *process identifier*. A process identifier is composed of an 8-bit node number (NN), a 2-bit cluster number (CN), a 3-bit processor number (PN) and a 16-bit locally unique identifier. Each cluster can contain at most 5 modules, and each node can contain at most 4 clusters. See figure A.2.

A *process server*, which executes on each processor of the system, is a system process that offers three services to other processes. Two of those will be described here:

- **StartProcess(pid)**
- **KillProcess(pid)**

The **StartProcess(pid)** operation starts the execution of the process with process identifier *pid*. This process will then be in state ready. By using the **KillProcess(pid)** operation, one can remove the process with process identifier *pid* from the system. All resources held by the process (e.g. memory and devices) will be released. Depending on the process identifier of the invoking process and on the process identifier of the started/killed process, these two primitives will actually be called and executed on different processors in the system. Therefore, these services will be offered by means of a RPC.

Time Management

In the EMPS system, a computer module contains a *clock counter* that represents the current time in milliseconds. The following services can read and/or set the clock counter:

- **time = GetTime()**
- **SetTime(time)**

Further, processes can invoke the following system routines:

- **DelayProcess(Δtime)**
- **WaitUntil(time)**

A process invoking **DelayProcess(Δtime)** will be suspended for Δtime. Usage of the routine **WaitUntil(time)** suspends the process until its clock counter reaches time.

Interprocess Communication

Two communication paradigms are supported in the EMPS system, based on the client/server model:

1. **Datagram service:** A client transfers a message to a server. Two kinds of addressing types can be distinguished:

   Singlecast: A message is sent to precisely one destination.

   Multicast: A message is sent to a subset of all possible destinations.

2. **RPC service:** A client sends a request message to a specific server and waits for a reply message from the server. The RPC service always uses singlecast addressing.

Datagram and RPC communication both use *mailboxes*. A mailbox is a data structure that provides location transparency for communication processes. This means that the communication between processes does not depend on their physical location. Using mailboxes for communication has the following advantages: efficient reconfiguration, multicast messages can be ordered and group membership can be dynamic.
Mailboxes can be created dynamically by processes, either to support a datagram service or a RPC service. A mailbox has a globally unique name which serves as the access *key*. Processes, having this key, can be connected to a mailbox via a *port*, being a local reference to a mailbox. Processes that are not connected to a mailbox cannot access that mailbox. Multiple clients and servers can be connected to one mailbox.
A number of mailbox services are provided which are implemented by a *mailbox server*. This is a system process that is executing on each processor of the system. These services are:

- **CreateMailbox(key,type,buffers,node)**
- **port = Connect(key)**

- **Disconnect(port)**
- **RemoveMailbox(key)**

Mailboxes can be created dynamically by the **CreateMailbox** service. The mailbox will be created in the common memory of a **node**. A node maintains a list of mailboxes in that node. The CreateMailbox service checks for the uniqueness of the **key**. The **type** parameter determines whether the mailbox provides a datagram service or a RPC communication service. The **buffers** parameter determines the maximum number of message buffers that is allocated to the mailbox. Further, the appropriate data structure is installed. This data structure consists of a globally unique access key, two lists of process identifiers that are connected to the mailbox via the initiator or responder ports and two queues of processes that are ready to transmit or receive a message (the *client queue* and the *server queue* respectively).
A process can establish a connection to a mailbox by invoking the **Connect** service. This service connects a client or a server process to the mailbox by inserting the process identifier of the process into the list of client or server processes respectively. This connection can be broken by using the **Disconnect** service.
Mailboxes can be removed from the system by invoking the **RemoveMailbox** service. Only mailboxes to which no ports are connected can be removed. This implies that the lists of clients and server processes as well as the client and server queue of the mailbox are empty.

ad.1) Datagram communication

The following services are provided for datagram communication:

- **Send(port,message)**
- **message = Receive(port)**

The client process can transmit a message, pointed to by **message**, by executing the **Send** operation on the initiator port indicated by **port** (a process can be connected to several mailboxes). The server process receives the message by executing the **Receive** operation on its responder port.

ad.2) RPC communication

RPC communication is provided for the following services:

- **Call(port,message)**
- **message = Get(port)**
- **Reply(port,message)**

The client process transmits a request message and waits for the reply by invoking the **Call** service on its initiator port. The server receives this request message by invoking the **Get** operation on its responder port. After executing the requested service, the server transmits a reply message back to the client by performing a **Reply** operation on its responder port. This message will cause the suspended client process to become ready-to-run again.

Notice that the following services are offered via system calls (SC's): Allocate, Release, GetTime, SetTime, DelayProcess, WaitUntil, Send, Receive, Call, Get and Reply.
Services that are offered via Remote Procedure Calls (RPC's) are: NewProcess, StartProcess, KillProcess, CreateMailbox, Connect, Disconnect and RemoveMailbox.

Appendix B

Proof of Correctness

In this appendix, the protocol presented in chapter 3, is proven to satisfy the correctness requirements reliability, unanimity and order. The proof does not include the primitives Join and Leave. It is assumed that group membership is static.

In the following, NAC(m) will denote the set of receivers that have not set their Ackflags in Acklist(m) yet, and CNAC(m) denotes the set of correct receivers in NAC(m).
First, we will assume no failures occur.

Lemma B.1 *Let m_1 and m_2 be messages in Tset, and let m_1.SequenceNr $\leq$ m_2.SequenceNr. Then, in each node, CNAC(m_1) $\subseteq$ CNAC(m_2).*

Proof:
If m_1.SequenceNr = m_2.SequenceNr, then $m_1 = m_2$, so trivially CNAC(m_1) = CNAC(m_2) holds.
For m_1.SequenceNr $<$ m_2.SequenceNr, it is enough to prove that any receiver r, r $\in$ NAC(m_1) $\wedge$ r $\notin$ NAC(m_2), has crashed. Two situations are possible:

1. r $\in$ NAC(m_1) $\wedge$ r $\notin$ Acklist(m_2).
2. r $\in$ NAC(m_1) $\wedge$ r $\in$ Acklist(m_2) $\wedge$ r's Ackflag in Acklist(m_2) is set.

Ad 1: Since the Acklist of a message is copied from the Rlist in the mailbox, r was still in Rlist at the time m_1 was put in the mailbox, but not anymore at the time m_2 was put in the mailbox. This means that r has crashed and the membership service removed its entry from the Rlist.

Ad 2: Messages received in order of their SequenceNrs. If r would be correct, it would receive and acknowledge m_1 first, and m_2 later. The only way situation 2 could occur, is that at some point r crashed, the membership detected this and acknowledged the newest message in the Tset at that time, viz. m_2.

□

Lemma B.2 *A message is not removed from the Tset until is is received by all correct receivers.*

Proof:
A message is removed from the Tset only by the intermediate when its SequenceNr is smaller

than or equal to AckRecRc, the piggybacked minimum value of SequenceNrs acknowledged as being received by all local receivers in each node.
So, if a message m is removed from Tset, the minimum value in AckRecRc must be m.SequenceNr at least. Therefore, for each node, all local receivers have received a message with a SequenceNr that is higher than or equal to m.SequenceNr. From Lemma B.1 follows that all these receivers have also received m.
□

Lemma B.3 *A message that is sent by a receiver will be received by all correct receivers as long as the sender remains intact.*

Proof:
Because of Lemma B.2, it is enough to show that the message sent is received by one correct receiver. A message m that is sent is put in the local mailbox, and will be sent to the distributor. The distributor will give incoming messages subsequent SequenceNrs. Therefore, receivers who have received all messages up to the message that has been sent, will receive that message next.
□

Theorem B.4 *The communication protocol satisfies the correctness requirements, if no failures occur.*

Proof:
The reliability, unanimity and ordering requirements follow directly from Lemmas B.1, B.2 and B.3 respectively.
□

What remains to be proven is that the lemmas above still hold in the presence of failures. Failures that can occur are omission failures, process crashes and mailbox crashes.

Intermediate crash:
When an intermediate crashes, it will not acknowledge message that have been received by all local receivers to the distributor any more. The distributor therefore cannot remove messages from its Hbuffer which will get full. It will then start the sync phase during which it notices the crashed intermediate. It will start a new intermediate in the node if possible. If there is no networked processor left, the node has crashed, so no lemma involves this node anymore.
If there a new intermediate has started, it will check the mailbox. If the mailbox holds any messages, the next message that is expected from the distributor is set to the highest SequenceNr in the mailbox plus one. If there are no message in the mailbox, NextSeqNr is set to the value of AckRecRc + 1. AckRecRc is the SequenceNr of the last message read by all receivers. This value is piggybacked on the message that starts the new intermediate. All local (correct) receivers have read the message with SequenceNr AckRecRc, and no newer messages are in the mailbox, so the next message that is expected is AckRecRc + 1.
□

Mailbox crash:
A mailbox crash has a higher priority than an intermediate crash, meaning that when both a mailbox and an intermediate crash (in the same node), first the mailbox will be rebuilt after which the new intermediate will continue normal operation. Messages that were in the crashed mailbox that were sent by local senders will already be in the new mailbox. If this message has already been received from the distributor, it will already have its SequenceNr, that is also in the new mailbox.
Messages that were received by some (not all) receivers in the node will be recovered into the new mailbox by the receivers. Messages that are not recovered by the receivers will be recovered

by the intermediate when it receives a multicast. Therefore, the intermediate has to check the mailbox to know what message it is expecting next.
□

Distributor crash:
A distributor crash is noticed by an intermediate sending a multicast request and not receiving it. The intermediate will then start a new distributor. The new distributor has to rebuild its Hbuffer. It will first ask the intermediates which messages they have. The lowest number is the message that every intermediate has, so all older messages have been read by all receivers and do not have to be stored in the Hbuffer. The highest number is the last message that has to be put in the Hbuffer. If an intermediate has crashed, a new intermediate is chosen. When a mailbox crashes, which contains messages only available in that node, these messages will be recovered by the receivers. If no receiver has received it, the message will still be available in the node the sender resides.
All messages that are available in the system will be sent to the distributor after which normal operation will resume. Because the highest SequenceNr of messages in the system is only known after the distributor's Hbuffer is recovered, the distributor will not accept multicasts during the startup phase.
□

Omission failures:
Missed messages that have SequenceNrs, such as multicasts and retransmissions, will be detected as such when the next message arrives and there is a gap in the SequenceNrs. All other communication uses timeouts to check if a message has been missed. The membership service will know whether a process crashed, the LAN crashed or a message was lost.
□

Theorem B.5 *The reliable SRT inter-node multicast protocol as presented in this thesis satisfies the requirements of reliability, unanimity and order.*

Proof:
This follows from Theorem B.4 and the fact the Lemmas stated earlier are still valid when failures occur.
□

Appendix C

Reliable SRT intra-node multicast in DEDOS

The protocol presented earlier in this thesis, the inter-protocol, is meant for the communication between processes executing on different *nodes. The protocol for reliable multicast, the intra-protocol, is intended for the communication between processes residing on the* same *node.*
This protocol, the intra-protocol, also uses replicated mailboxes located in common memory to exchange messages, on which a multiple source ordering is established. The protocol can handle a certain amount of memory crashes and process or processor failures. A protocol description and a corresponding proof of correctness of this protocol is given here. For more detailed information, the reader is referred to [Verm].

Protocol Description

The intra-protocol uses mailboxes located in common memory for the communication between senders and receivers of a multicast group. One of these mailboxes is the *active* mailbox, the rest are *spare* boxes. During normal operation, only the active mailbox is used to increase efficiency. When the active mailbox crashes, a spare box is appointed to be the new active mailbox. The senders and receivers will have to fill the new active mailbox with the messages that were in the crashed mailbox. Therefore, both senders and receivers keep messages they send resp. receive in their private memory, in the history-buffer.

The protocol is split in two parts, the *Communication Protocol* and the *Recovery Protocol*. The Communication Protocol performs efficient SRT multicast when the active mailbox functions correctly. When the active mailbox crashes, the Recovery Protocol will appoint a new mailbox and recover the messages from the crashed mailbox using the history-buffers of the multicast-group members.

The Communication Protocol consists of the Send and Receive primitives, the Recovery Protocol of the StartRecovery and Recover primitives. There are three more primitives, DeleteMember, Join and Leave that are necessary to implement dynamic group membership. The Send and Receive primitives may be executed by at most processor at a time, because these primitives operate om common memory that has to stay consistent. Therefore we need some sort of mutual

exclusion algorithm.

First, we will give the definitions of some terms used in this appendix.

- Each sender and receivers keeps messages it sends resp. receives in a history-buffer in its local memory. We will denote the history-buffer with **Hbuffer**.
- In the mailbox, every message has a list of boolean **Ackflags** and process-Ids, one for each receiver of the message. This list of flags and process-Ids is called the **Acklist**. Receivers set their flag to acknowledge the receipt of the message. The Acklist is needed to make it possible to see if all receivers have received a message.
- The process-Ids of receivers of a multicast group are stored in a list called the **Rlist**, together with a **Pflag** for each receiver. The Pflag (or present-flag) is used in the recovery protocol to indicate a sender or receivers is still functioning.
- When a message element in a mailbox is free, i.e. the message element can be used to store a message in, it is said to be in **Fset**, the set of free message elements in the mailbox.
- When a message element in a mailbox holds a message, it is said to be in **Tset**, the set of message elements in the mailbox that are 'in transmission'.

Next we will take a look at the primitives.

Communication Protocol:
Senders send messages using the Send procedure, receivers will receive messages using the Receive procedure. When a process notices a mailbox crash, it starts the Recovery Protocol.

Send procedure: to send a message, there must be a free message element that can be taken i.e. there must be a message in Fset. If there is no free message element, the sender must wait until one becomes available. The sender will fill the free message element with the message data and fill the Acklist which the process-Ids of the receivers that are in Rlist, and resets all Ackflags. The message is given a unique sequence number (Seqnr) and moved to Tset, the set of messages in the mailbox in transmission.
Meanwhile, messages the receiver has in its Hbuffer, that are already read by all correct receivers, are deleted. The mailbox holds Oldestseqnr, the sequence number of the latest message acknowledged by all correct receivers to make this possible. To conclude, the sender will add the message to its local Hbuffer.

Send(l), invoked by an arbitrary sender s, where l is a local message buffer, with l.Data containing the data to be sent:

```
1  while Fset = ∅ do
2      DelayProcess(TimeOutSend);
3  m :∈ Fset;
4  m.Data ← l.Data;
5  m.Acklist ← receivers in Rlist, with Ackflags cleared;
6  m.Seqnr ← Nextseqnr;
7  Nextseqnr ← Nextseqnr + 1;
8  Tset ← Tset ⊕ {m};
9  l.Acklist ← m.Acklist; /* without the Ackflags */
10 l.Seqnr ← m.Seqnr;
11 Hbuffer(s) ← Hbuffer(s) ⊕ {l};
12 Oldestseqnr(s) ← Oldestseqnr;
13 Hbuffer(s) ← Hbuffer(s) ⊖ {x ∈ Hbuffer(s) | x.Seqnr < Oldestseqnr(s)};
```

Receive procedure: a receiver will expect to receive the message with sequence number Lastseqnr + 1. It will wait until such a message becomes available in Tset. When such a message is available, or all Pflags are set, meaning that the message has just been recovered (see Recovery Protocol), the receiver will copy it to its local Hbuffer, increment Lastseqnr, store the message locally and acknowledge the receipt by setting its Ackflag in the Acklist of the message in the mailbox.
If all Ackflags are set, the receiver was the last to receive the message and it can move the message element from Tset to Fset. It will also increment Oldestseqnr in the mailbox so other receivers can delete the message from their Hbuffer. Of course, the receiver itself will also remove the message from its Hbuffer.
Finally, the message is passed from local memory to the application.

Receive(l), invoked by an arbitrary receiver r, where l is a local message buffer:

```
1   while { x ∈ Tset | x.Seqnr > Lastseqnr(r)} = ∅ do
2       DelayProcess(TimeOutReceive);
3   repeat
4       locate the message element m in Tset,
            with m.Seqnr = min { x.Seqnr | x ∈ Tset ∧ x.Seqnr > Lastseqnr(r)};
5       if m.Seqnr ≠ Lastseqnr(r) + 1 ∧ not all Pflags in the mailbox are set then
6           DelayProcess(TimeOutReceive);
7   until m.Seqnr = Lastseqnr(r) + 1 ∨ all Pflags in the mailbox are set;
8   l ← m; /* without the Ackflags */
9   Hbuffer(r) ← Hbuffer(r) ⊕ {l};
10  Lastseqnr(r) ← l.Seqnr;
11  pass l.Data to application process;
12  set m.Ackflag(r);
13  if all Ackflags of m are set then
14      Oldestseqnr ← Lastseqnr(r) + 1;
        /* let Ackedset = {x ∈ Tset | x.Seqnr < Oldestseqnr} */
15      Tset ← Tset ⊖ Ackedset;
16      Fset ← Fset ⊕ Ackedset;
17  Oldestseqnr(r) ← Oldestseqnr;
18  Hbuffer(r) ← Hbuffer(r) ⊖ { x ∈ Hbuffer(r) | x.Seqnr < Oldestseqnr(r)}
```

Recovery Protocol:
When a mailbox crash is noticed, a spare box will be located. All processes will locate the same spare box. After that, the StartRecovery procedure is executed. To recover the messages in the processor's Hbuffer the Recover procedure is executed repeatedly executed until all messages from the local Hbuffer have been restored in the mailbox. Then the process will set its Pflag. If the process is a sender, it may start sending messages again when all Pflags are set. A receiver may start receiving messages after it has set the Pflag.

StartRecovery: this procedure is executed after a spare box has been chosen. All senders and receivers will unanimously choose the same spare box. A sender or receiver that executes StartRecovery compares the Oldestseqnr of the new mailbox with its own Oldestseqnr. If its own Oldestseqnr is higher, it moves all message elements in the new mailbox with a sequence number smaller than its own Oldestseqnr from Tset to Fset, and sets Oldestseqnr in the mailbox to its own Oldestseqnr. If its own Oldestseqnr is lower, it copies Oldestseqnr from the mailbox, and removes messages with a sequence number older than Oldestseqnr from its Hbuffer.

StartRecovery, invoked by an arbitrary process p:

```
1   if Oldestseqnr(p) > Oldestseqnr then
```

```
2       Oldestseqnr ← Oldestseqnr(p);
        /* let Ackedset = { x ∈ Tset | x.Seqnr < Oldestseqnr} */
3       Tset ← Tset ⊖ Ackedset;
4       Fset ← Fset ⊕ Ackedset;
5   else
6       Oldestseqnr(p) ← Oldestseqnr;
7       Hbuffer(p) ← Hbuffer(p) ⊖ { x ∈ Hbuffer(p) | x.Seqnr < Oldestseqnr(p)}
```

Recover: first the messages with a Seqnr older than the Oldestseqnr from the mailbox are removed from the Hbuffer. If the first message in the Hbuffer of the process is not yet in the Tset of the mailbox, the process will add it. It sets the Ackflags of receivers that have an Ackflag but are not in Rlist (the list of receivers) anymore. If the recovering process is a receiver, it will also set its own Ackflag and update Nextseqnr. Because both senders and receivers can be recovering messages, and senders only store their own messages in their Hbuffers, it is possible that there are messages in the Tset that have a higher Seqnr than the message being recovered. Nextseqnr is then also higher. Therefore, after adding the message to the Tset, Nextseqnr is set to the maximum of Nextseqnr and the Seqnr of the message + 1. If the message was already in the Tset, and if the process is a receiver, is sets its Ackflag in the message in the mailbox. If all Ackflags are set, it moves all messages with a Seqnr less than or equal to the Seqnr of the message just acknowledged from the Tset to the Fset, and sets Oldestseqnr to the Seqnr of the message + 1.

Recover(l), invoked by an arbitrary process p, where l ∈ Hbuffer(p):

```
1   while Fset = ∅ do
2       DelayProcess(TimeOutSend);
3   m :∈ Fset;
4   m ← l;
5   clear all Ackflags of m;
6   set m.Ackflag(p);
7   forall q in m.Acklist do
8       if q not in Rlist then
9           set m.Ackflag(q);
10  Oldestseqnr(p) ← Oldestseqnr;
11  if m.Seqnr < Oldestseqnr(p) then
12      Fset ← Fset ⊕ {m};
13  else
14      if {x ∈ Tset | x.Seqnr = m.Seqnr} = ∅ then
15          Nextseqnr ← max {Nextseqnr, m.Seqnr + 1};
16          Tset ← Tset ⊕ {m};
17      else /* let n be the message element with n.Seqnr = m.Seqnr */
18          if p is a receiver then
19              set n.Ackflag(p);
20              if all Ackflags of n are set then
21                  Oldestseqnr ← n.Seqnr + 1;
                    /* let Ackedset = {x ∈ Tset | x.Seqnr < Oldestseqnr} */
22                  Tset ← Tset ⊖ Ackedset;
23                  Fset ← Fset ⊕ Ackedset;
24                  Oldestseqnr(p) ← Oldestseqnr;
25  Hbuffer(p) ← Hbuffer(p) ⊖ {x ∈ Hbuffer(p) | x.Seqnr < Oldestseqnr(p)}
```

Dynamic Group Membership:
With the Join and Leave procedures, processes can join and leave multicast groups.

Join: the join procedure will first notify the membership service and ask the mailbox server (see

appendix A) for the location of the mailboxes and spare boxes which it will get when joining is permitted.
After that, the join procedure will wait until all Pflags in the mailbox are set. If so, it adds an entry for itself in the Slists or Rlists of all boxes. It will also set its Pflag in the mailbox and reset its Pflag in all spare boxes. Then the procedure will wait until all Pflags are set which means the mailbox is not in recovery.

Join, invoked by an arbitrary process p:

```
1   notify the membership service of the Join;
2   while not all Pflags in the mailbox are set do
3       DelayProcess(TimeOutJoin);
4   if p is a sender then
5       forall boxes do
6           port = Connect(key); /* ask mailbox server for location of box */
7           if box is mailbox then
8               Slist ← Slist ⊕ {(Pid(p);set)};
9           else
10              Slist ← Slist ⊕ {(Pid(p);cleared)};
11  else /* p is a receiver */
12      forall boxes do
13          port = Connect(key); /* ask mailbox server for location of box */
14          if box is mailbox then
15              Lastseqnr(p) ← Nextseqnr – 1;
16              Rlist ← Rlist ⊕ {(Pid(p);set)};
17          else
18              Rlist ← Rlist ⊕ {(Pid(p);cleared)}
```

Leave: a leaving process can be treated the same way as a crashed one, with one difference: the process has messages in its Hbuffer that might be needed in case of a mailbox recovery. Therefore, the Leave procedure will just recover all those messages in all spare boxes, using the StartRecovery and Recover primitives. After that, it executes DeleteMember, a procedure that is also executed by the membership protocol when a sender or receiver crashes. To conclude, the membership is notified of the Leave.

Leave, invoked by an arbitrary process p:

```
1  forall spare boxes do
2      StartRecovery;
3      forall x ∈ Hbuffer(p) do
4          Recover(x);
5  DeleteMember;
6  Disconnect(ports);
7  notify the membership service of the Leave
```

DeleteMember: this procedure will kill the process that leaves the group or has crashed. If the process was a sender, its id and Pflag are removed from the list of senders (Slist) in all boxes.
If it is a receiver, its id and Pflag are removed from the list or receivers (Rlist) in all boxes. In the mailbox, the latest message still has an Ackflag for the deleted receiver. The procedure will set this flag. It will then check if this Ackflag was the last one. If so, the message is moved from Tset to Fset, and Oldestseqnr is updated.

DeleteMember, invoked by the membership service or a leaving process p:

```
1   KillProcess(Pid(p));
2   if p is a sender then
3       forall boxes in reverse order do
4           Slist ← Slist ⊖ {(Pid(p);Pflag(p))};
5   else /* p is a receiver */
6       forall boxes in reverse order do
7           if box is mailbox then
8               locate the message element m in Tset,
                with m.Seqnr = max{x.Seqnr | x ∈ Tset};
9               set m.Ackflag(p);
10              if all Ackflags of m are set then
11                  Oldestseqnr ← m.Seqnr + 1;
                    /* let Ackedset = {x ∈ Tset | x.Seqnr < Oldestseqnr} */
12                  Tset ← Tset ⊖ Ackedset;
13                  Fset ← Fset ⊕ Ackedset;
14          Rlist ← Rlist ⊖ {(Pid(p);Pflag(p))};
```

Proof of Correctness of the intra-protocol

Theorem: The protocol satisfies the requirements of reliability, unanimity and ordering.

We will prove the theorem for static multicast groups only. In this proof, a field Fn of a message m is denoted by m.Fn, NAC(m) denotes the set of receivers that have not set their Ackflags in Acklist(m) yet, and CNAC(m) denotes the set of *correct* receivers in NAC(m).

Lemma C.1 *Let m_1 and m_2 be messages in the Tset, and $m_1.Seqnr \leq m_2.Seqnr$. Then $CNAC(m_1) \subseteq CNAC(m_2)$.*

Proof:
If m_1.Seqnr = m_2.Seqnr, then m_1=m_2, this is trivial.
If m_1.Seqnr<m_2.Seqnr, what remains to be proven is that a receiver r has to be a crashed receiver, when $r \in$ NAC(m_1) $\wedge r \notin$ NAC(m_2). So, $r \in$ NAC(m_1), and two situations are possible:

1. $r \notin$ Acklist(m_2)
2. $r \in$ Acklist(m_2) $\wedge$ r's Ackflag in Acklist(m_2) is set.

Ad 1: Since the Acklist of a message is copied from the Rlist in the mailbox, r was still in the Rlist at the time m_1 was sent, but not anymore at the time m_2 was sent. This means that r has crashed and that the membership service removed its entry from the Rlist.

Ad 2: Messages are sent and received in order of their Seqnrs. If r would be correct, it would receive and acknowledge m_1 first, and m_2 later. The only way that situation 2 could occur, is that at some point r crashed, the membership service detected this and acknowledged the newest message in the Tset at that time, viz. m_2.

□

Lemma C.2 *A message is not removed from the Tset until it is received by all correct receivers.*

Proof:
A message m_1 is only removed from Tset when there is a message m_2, with m_2.Seqnr$\geq m_1$.Seqnr, for which all Ackflags in Acklist(m_2) are set. This implies that CNAC(m_2)=$\emptyset$, and by Lemma 1, also CNAC(m_1)=$\emptyset$, in other words, m_1 has been received by all correct receivers.
□

Theorem C.3 *The Communication Protocol satisfies the correctness requirements, if the mailbox remains intact.*

Proof:
Reliability and unanimity follow directly from Lemma C.2. Order follows from the fact that messages are sent and received in order of their Seqnrs.
□

Next, it is proved that after a mailbox crash, the correctness is maintained by the Recovery Protocol.

Lemma C.4 *If a correct receiver still has to receive a message, then that message is recovered in the new mailbox.*

Proof:
If a message still has to be received by a correct receiver, then the sender of that message is still correct, or at least one process that has the message in its Hbuffer, because senders store the messages they send and receivers store the messages they receive in their Hbuffers. Furthermore, during the Communication Protocol, a message is not removed from any Hbuffer before it is removed from the Tset of the mailbox (now crashed), in which case it was already received by all correct receivers, according to Lemma C.2.
Eventually, one of these correct processes notices the mailbox crash and recovers the message in the new mailbox.
□

Lemma C.5 *Lemma C.1 still holds, during and after recovery.*

Proof:
The trivial case is skipped, and it is proved that any receiver r, $r \in$ NAC(m_1) $\wedge$ $r \notin$ NAC(m_2), where m_1 and m_2 are messages in the Tset and m_1.Seqnr<m_2.Seqnr, has crashed. So, $r \in$ NAC(m_1) and two situations are possible:

1. $r \notin$ Acklist(m_2)
2. $r \in$ Acklist(m_2) $\wedge$ r's Ackflag in Acklist(m_2) is set.

Ad 1: Since the Acklist of a message that is recovered in the new mailbox was copied from the Rlist of the old one, r was still in the Rlist at the time m_1 was sent, but not at the time m_2 was sent. This means that r has crashed and that the membership service removed its entry from all Rlists.

Ad 2: During recovery, messages are (received and) acknowledged in order of their Seqnrs. A Seqnr is skipped only when all Pflags are set, indicating that no more messages will be recovered, so that no messages are missed or have to be (received and) acknowledged out

of order.
If r would be correct, it would (receive and) acknowledge m_1 first, and m_2 later. The only way that situation 2 could occur, is via the following scenario: r has crashed, and at some point during recovery, the membership service is notified of this. It acknowledges the newest message in the Tset at that time (let m_3 be that message), and removes r from the Rlist. Then, new messages with Seqnrs higher than m_3.Seqnr are recovered (added to the Tset) with r's Ackflag already set. In this case, m_2 is either the same message as m_3, or a message with m_2.Seqnr>m_3.Seqnr.

□

Lemma C.6 *Lemma C.2 still holds, during and after recovery.*

Proof:
The proof is similar to the proof for Lemma C.2.
□

Theorem C.7 *The correctness requirements are maintained by the Recovery Protocol.*

Proof:
Reliability and unanimity follow from Lemmas C.4 and C.6 and the fact that Seqnrs are only skipped by receivers when all Pflags are set, in which case recovery is complete. Order follows from the fact that messages are received and acknowledged in order of their Seqnrs, and that Seqnrs are only skipped by receivers when all Pflags are set.
□

Appendix D

Lazy Replication

At first, the Lazy Replication Protocol*, developed by Rivka Ladin et al. [Ladi] intuitively seemed like a good protocol to implement multiple source ordering with. This appendix will show that this protocol turned out to be unsuitable after all.*

Short outline of the protocol

This protocol was meant to run on a system that consists of several nodes connected via a LAN. Each node consists of several processes.

Processes that want to read or send messages interact with a front-end that is located on the same node via *update* (viz. send in DEDOS) and *query* (viz. receive in DEDOS) messages. The front ends communicate with (one of more) replicas who in turn communicate with each other via the LAN using 'gossip' messages. The front-end introduces the dependencies between the messages, e.g. it can mark a message p in a way that all receivers that will receive p will only do so when they have already received message q *(causal ordering)*. Timestamps are used to label messages.

If we were to implement this protocol on the EMPS system, we would have a front-end and a replica on each node. We could even include the front-end primitives in the kernel of each processor. Then, receivers could communicate directly with the replica in the node they reside in. The replicas could communicate via Ethernet when necessary (e.g. there is no communication necessary between replicas when a sender sends a message) which improves the efficiency.
Because messages can be received only when they are replicated at other replicas, the protocol is reliable. A crashed replica can retrieve lost messages from other replicas.

Communication between different processes use different type of messages. The different type are:

from	to	message type
process	front-end	query/update messages
front-end	replica	call messages
replica #m	replica #n	gossip messages
replica	front-end	call message replies
front-end	process	query/update message replies

We would like to use this protocol to achieve multiple source ordering, i.e. the messages in one multicast group must be read by all receivers in the same order (but these messages may be interleaved with other messages in different ways by different receivers).

The order is introduced by the front-ends but can also be introduced by the processes if needed. This implies that in this basic case, there will be no multiple source ordering as messages can be sent simultaneously by several senders that communicate with different front-ends. Suppose front-ends f_1 and f_2 have received all messages in a multicast group up to message m_1 and m_2 respectively. When they both send a message for that multicast group, they can introduce a dependency upon message m_1 (or m_2) and earlier messages only, not the message being simultaneously sent by the other front-end. So, the messages sent by f_1 and f_2 do not depend on each other and can therefore be received in either order (first the message from f_1 which might be related to message m_1, then the message from f_2 which might be related to message m_2, or vice versa).

In addition to causal operations, the lazy replication protocol offers *forced* and *immediate* operations. Forced operations are performed in the same order at all replicas. They can be differently interleaved with causal operations at different replicas. Immediate operations are performed in the same order relative to *all* operations at all replicas.

Forced operations are more inefficient than causal operations, immediate operations are even more inefficient than forced operations and should be used infrequently.

Using only forced operations guarantees multiple source ordering (to quote [Ladi]: "Our system can be instantiated with only forced operations. In this case it provides the same order for all updates at all replicas, and will perform similarly to other replication techniques that guarantee this property"). Because all processes can send messages to all multicast groups, it follows that all updates must be made using forced operations.

Now we know that we can implement multiple source ordering using the lazy replication protocol. Also we know *how* we can implement it: by using forced operations only. We do not need immediate operations and the part of the protocol that handles these operations can be discarded.

Implementation of multiple source ordering

Let us take a look at how forced operations are implemented. The operations are ordered by using a primary replica (with backup replicas in case of a crash) that introduces the order using timestamps. So, all updates are processed at one primary replica and from there distributed to other replicas and, finally, the receivers. This means that we now have, in fact, a *central node protocol*. This leads us to the conclusion that implementing multiple source ordering using Ladin's lazy replication protocol instead of a central node protocol is a waste of effort as the protocol offers extra functionality we are not interested in.

Appendix E

Efficient Reliable Broadcast

The protocol presented in this thesis was inspired by the Efficient Reliable Broadcast protocol that was developed by M.F. Kaashoek, see [Kaa0] and [Kaa1]. In this appendix, a short outline of this protocol, that achieves reliable atomic broadcast, and is meant for use on broadcast networks, is given. The terminology is adapted from [Kaa0] and might be different from the terminology used elsewhere in this thesis.

Short outline of the protocol

The protocol is a broadcast protocol that is meant for communication from 1 sender to $n-1$ receivers in an n-process network. This broadcast protocols uses unreliable broadcast messages. Most networks, including Ethernet, provide these broadcast messages. Obviously, the use of one broadcast message is more efficient than using $n-1$ point-to-point messages.

For the sake of simplicity, we will assume that each process runs on a different processor and the system runs a simple application. Generalisations to multiprogramming and multiple applications are straightforward.

When a process wants to broadcast a message M, it sends the message to a special process called the *sequencer*. The sequencer process accepts messages to be broadcast. The protocol can be easily extended to unanimously8 elect a new sequencer in case the current sequencer crashes. If we were to implement this protocol on the EMPS system, we would locate the sequencer in one node. All senders would send their multicasts over the LAN to the sequencer.

The sequencer will allocate a unique sequence number s for an incoming message M. M and s will be broadcast using one message. Because there is only one process that broadcasts messages, the protocol guarantees a total ordering on the messages. Note that if we want to implement this protocol on the EMPS system, all processors should be connected to the Ethernet.

When a process receives a broadcast, it can see by the sequence number if it has missed any broadcasts, as the sequence numbers of received messages will show a gap. The process will send a special message to the sequencer asking for a retransmission of the missed message (or messages, if several were missed). Therefore, the sequencer must store old broadcasts in its history buffer. Because the sequencer has a finite amount of memory available, it cannot store old broadcasts forever. To solve this problem, all processes send piggybacked acknowledgements of received

messages in their broadcast requests. The acknowledgement consists of the SequenceNr of the last message received. The sequencer can maintain a table with all acknowledged SequenceNr, compute the lowest value in this table and discard all broadcast messages with a SequenceNr smaller than or equal to that minimum. Because processes may be quiet for some time, they will explicitly acknowledge the last correctly received message when they have not requested a broadcast for a certain interval of time Δt.

When the sequencer still has no room left in the history, its starts a synchronisation phase in which it will broadcast the sequence number of the last message broadcast. The receivers will then ask for retransmissions of missed broadcasts and inform the sequencer when they are up-to-date. When all receivers are up-to-date, the history is flushed and normal operation is resumed. During the synchronisation phase, no broadcast requests will be honoured.

The protocol as described in this appendix is not yet reliable. Suppose a receiver missed a broadcast message after which the sequencer crashes. The message is then lost and the consensus requirement is violated. Therefore, message have to be replicated.

Primitives

Broadcast:
When a process wants to broadcast a message, it sends the sequencer a data message containing LastSeqReceived, MessageId and its process ID. MessageId will then be incremented and a timer is started. If the timer expires before the broadcast is received, the message will be resent.

DataMessage:
The sequencer can uniquely identify an incoming message by its MessageId and the processor number and will discard messages it already broadcasted. It will also inform the sender of any duplicate message. To be able to tell if a message is a duplicate, MessageIds of received messages are stored in an array MessageNr. The sequencer also has an array SequenceNr to store the acknowledgements.
The received message then has to be stored in the history. If there is still room, the message will be stored and broadcast together with a unique sequence number, which is then incremented. If, on the other hand, there is no room in the history, all acknowledged messages are deleted. If there is still no room after this operation, the synchronisation phase is entered.

AcceptBroadcast:
When a broadcast message arrives, the receiving kernel will execute AcceptBroadcast. First it will check if the sequence number is the one it is expecting. If so, LastSeqReceived is incremented. If not, the sequencer is asked for retransmissions of missed broadcasts.

Broadcast, invoked by an arbitrary sender, with data the data to be sent:

```
broadcast(data)
char *data;
{
/* Algorithm executed by the kernel on the machine wanting to broadcast. */
   struct header h;                    /* outgoing header is built here */

   h.type = DATA;                      /* DATA means request for broadcast */
   h.SequenceNr = LastSeqReceived;     /* piggybacked acknowledgement */
   h.MessageNr = MessageId;            /* unique message identifier */

   send(sequencer, &h, data);          /* point-to-point message to sequencer */
```

```
    MessageId++;        /* use different message id next time */
    StartTimer();       /* just in case a broadcast does not arrive */
    block();            /* wait for broadcast */
}
```

DataMessage, invoked by the sequencer with h the header and data the data of the incoming message:

```
DataMessage(h, data)
struct header *h;
char *data;
{
/* Algorithm used by the sequencer when a request-to-broadcast arrives. */
    struct message *m, GetFromHistory();

    if (h–>MessageNr == MessageNr[h–>SenderID]) {
      /* This is a duplicate. Sender must have timed out. */
      m = GetFromHistory(h–>MessageNr, h–>SenderID);      /* Fetch sequence nr */
      send(h–>SenderID, m–>header, NULLDATA);             /* Tell sender */
    } else {
      /* new message, not yet broadcast. */
      if (!FullHistory()) {                               /* is history full? */
        /* No, there is room. */
        MessageNr[h–>SenderID] = h–>MessageNr;            /* save unique msg-id */
        h–>SequenceNr = NextSeqToUse;                     /* assign sequence number */
        StoreInHistory(h, data);                          /* save message */
        PassToApplication(data);                          /* upcall to application */
        NextSeqToUse++;                                   /* for next time */
        h–>type = BROADCAST;
        broadcast(h, data);                               /* do the broadcast */
      } else
        EnterSyncPhase();                                 /* history is full, flush it. */
    }
}
```

AcceptBroadcast, executed when a broadcast message is received, with h and data the header resp. the body of the broadcast message:

```
AcceptBroadcast(h, data)
struct header *h;
char *data;
{
    struct header rh;

    if (h–>SequenceNr == LastSeqReceived) {
      /* expected sequence number */
      LastSeqReceived++;                    /* one more broadcast received */
      PassToApplication(data);              /* pass to application */
    } else {
      /* wrong sequence number */
      rh.type = RETRANSMIT;                 /* ask sequencer for the missing ones */
      rh.SequenceNr = LastSeqReceived;      /* tell where we are */
      send(SequencerID, &rh, NULLDATA);     /* send to sequencer */
      SetTimer();                           /* make sure that we get a reply */
    }
}
```

Bibliography

[Alst] D. ALSTEIN AND P. VAN DER STOK, *Hard Real-Time Reliable Multicast in the DEDOS system*, Eindhoven University of Technology Computing Science Note 93/17, June 1993.

[Chan] J. CHANG AND N.F. MAXEMCHUK, *Reliable Broadcast Protocols*, ACM Transactions on Computer Systems, Vol. 2, No. 3, August 1984, pp. 251-273.

[Cri0] F. CRISTIAN, H. AGHILI, R. STRONG AND D. DOLEV, *Atomic Broadcast: from Simple Message Diffusion to Byzantine Agreement*, Technical Report RJ5244 (54244), IBM Almaden Research Centre, revised December 1989.

[Cri1] F. CRISTIAN, *Synchronous Atomic Broadcast for Redundant Broadcast Channels*, IBM Almaden Research Centre, revised April 1990.

[Dijk] G.J.W. VAN DIJK, *The EMPS System: An Overview*, Eindhoven University of Technology, November 1991.

[Kaa0] M.F. KAASHOEK, A.S. TANENBAUM, S.F. HUMMEL AND H.E. BAL, *An Efficient Reliable Broadcast Protocol*, Operating Systems Review, Vol. 23, No. 4, October 1989, pp. 5-20.

[Kaa1] M.F. KAASHOEK, A. TANENBAUM, *Efficient Reliable Group Communication for Distributed Computer Systems*, Vrije Universiteit Amsterdam, pp. 44-86.

[Kenn] J.J.P. KENNES, *Reliable HRT inter-node multicast in DEDOS*, Master's Thesis, Eindhoven University of Technology, August 1993.

[Ladi] R. LADIN ET AL, *Providing High Availability Using Lazy Replication*, ACM Transactions on Computer Systems, Vol. 10, No. 4, November 1992, pp. 360-391.

[Mell] P.M. MELLIAR-SMITH, L.E. MOSER, V. AGRAWALA, *Broadcast Protocols for Distributed Systems*, IEEE Transactions on Parallel and Distributed Systems, Vol. 1, No. 1, January 1990.

[Stok] P.D.V. VAN DER STOK, O.S. VAN ROOSMALEN, E.J. LUIT, W.H.J.M. LEMMENS AND D.K. HAMMER, *The Dependable Distributed Operating System DEDOS. An Overview*, Eindhoven University of Technology, August 1990.

[Verm] G.A. VERMEULEN, *Reliable Multicast for Soft Real-Time Processes in DEDOS*, Master's Thesis, Eindhoven University of Technology, April 1992.

Acknowledgements

First of all, I would like to thank Peter van der Stok who was one of my advisors when I started writing this thesis, but became my supervisor when Prof. Hammer went on a Sabbatical year. Peter van der Stok still managed to find the time to give useful comments on my work and help me in finding the right direction. I would also like to thank Dick Alstein, my first advisor, who offered me a lot of different views on the subject and showed his continuous interest. My second advisor was Jozef Hooman. I would like to thank him for reading my thesis.
For me, it was a pleasure working with all the aforementioned.

Some other people I would like to thank are my parents for all the support during my study, Ivana for preventing me from losing my mind when the stress was at its highest, the people from student society GEWIS who gave me a great opportunity during my studies to learn social and organising skills which are, I think, very important for an engineer, and my friends The TimeWasters. Without the TimeWasters, I would have had to spend 3 months learning UNIX and LaTeX before I could even have started writing this thesis.

"How difficult it may be, it is of utmost importance to schedule the writing of a Master's Thesis Hard Real-Time, and to schedule your girlfriend Soft Real-Time." —Walter.