The Design of the
EMPS Multiprocessor Executive
for Distributed Computing

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van
de Rector Magnificus, prof. dr. J.H. van Lint, voor
een commissie aangewezen door het College
van Dekanen in het openbaar te verdedigen op
dinsdag 16 maart 1993 om 16.00 uur

door

GERARDUS JOHANNES WICHARDUS VAN DIJK

geboren te Nijmegen

druk: wetenschappelijke drukkerij, Nijmegen.
Dit proefschrift is goedgekeurd
door de promotoren
prof. dr. dipl. ing. D.K. Hammer
en
prof. dr. S.J. Mullender

copromotor:
dr. ir. K. Kopinga
# Contents

1 Introduction .................................................. 1
   1.1 Application areas ....................................... 1
      1.1.1 Real-time data-processing and experiment control 1
      1.1.2 Dependable Distributed Computing ................. 4
   1.2 Distributed operating systems .......................... 7
   1.3 Design philosophy ...................................... 10
   1.4 Development stages .................................... 12
   1.5 Overview .............................................. 12

2 Design of the Hardware Architecture ................. 15
   2.1 Parallel processing architectures ...................... 16
      2.1.1 Organization of system memory .................... 17
      2.1.2 Interconnection network .......................... 18
         2.1.2.1 Hierarchical cluster systems ................. 19
   2.2 EMPS Hardware architecture .......................... 20
      2.2.1 The computer module .............................. 22
      2.2.2 The memory module ............................... 25
      2.2.3 The system controller ............................ 26
      2.2.4 The local area network controller ............... 26
      2.2.5 Inter-node communication ......................... 26

3 Design of the Software Architecture ................. 27
   3.1 The EMPS Nucleus ..................................... 30
      3.1.1 Memory management ................................ 32
      3.1.2 Process management ............................... 35
         3.1.2.1 Process address ............................... 35
         3.1.2.2 Process states ................................. 36
         3.1.2.3 Process switching ............................. 37
         3.1.2.4 Tasks and processes .......................... 40
         3.1.2.5 Memory management ............................ 41
3.1.3 Process synchronization .................................. 45
3.1.4 Time management ........................................ 46
3.1.5 Interrupt handling ....................................... 48
3.1.6 Interprocess communication (IPC) ....................... 50
  3.1.6.1 Mailbox communication ............................ 52
  3.1.6.2 Mailbox address .................................. 54
  3.1.6.3 Mailbox ........................................ 54
  3.1.6.4 Port ........................................... 56
3.2 The EMPS Kernel ............................................ 58
  3.2.1 Process management .................................. 61
  3.2.2 Mailbox management ................................ 62
  3.2.3 Device management .................................. 63
3.3 The EMPS operating system ................................. 66
  3.3.1 File access ......................................... 66
3.4 System initialization ....................................... 68

4 Implementation of Interprocess Communication (IPC) .... 69
  4.1 Service layer ............................................ 70
    4.1.1 Service layer interface ........................... 72
      4.1.1.1 Service data ................................ 72
      4.1.1.2 Services .................................... 73
    4.1.2 Service layer implementation ....................... 74
      4.1.2.1 Messages .................................... 74
      4.1.2.2 Protocol for transparent mailbox access .... 76
      4.1.2.3 Mailboxes .................................. 78
  4.2 MAC layer ................................................ 81
    4.2.1 MAC layer interface ................................ 81
      4.2.1.1 Service data ................................ 81
      4.2.1.2 Services .................................... 82
    4.2.2 MAC layer implementation .......................... 82
      4.2.2.1 Protocols for datagram communication ....... 83
  4.3 Example of communication ................................ 87
  4.4 Implementation issues .................................. 89
  4.5 Performance evaluation .................................. 89

5 Efficient Process Migration ................................. 95
  5.1 Related work ........................................... 96
  5.2 Process migration ....................................... 98
    5.2.1 Interface ......................................... 98
    5.2.2 Implementation ................................... 99
      5.2.2.1 Process context ................................ 100
      5.2.2.2 Additional mailbox services ................. 100
1

Introduction

1.1 Application areas

The design of the EMPS (Eindhoven Multi-Processor System) executive for distributed computing was triggered by two different application areas, viz. real-time data-processing and control of physics experiments, and dependable distributed computing.

1.1.1 Real-time data-processing and experiment control

In 1979, the Department of Physics of the Eindhoven University of Technology initiated the standardization of hardware and software for data-acquisition and control of physics experiments. This resulted in the design of a general-purpose Physics Data Acquisition System (PhyDAS) and an interpretative development environment EPEP (Eindhoven Program Editor and Processor).

The goal in the overall design of PhyDAS is to provide a general-purpose real-time measurement system that can be used as a framework for intelligent, state-of-the-art interface modules, e.g. analog-to-digital converters (ADC), digital-to-analog converters (DAC), scalers (SCA), preset scalers (PSC), multi channel analyzers (MCA), and transient recorders (TRC) (See Fig. 1.1).

PhyDAS has a separate measuring bus (PhyBUS) that interconnects the interface modules. By separating the computer bus and measuring bus, the interface modules are independent of the type of computer that is used in the experiment, and the real-time behavior of the system is more predictable.
Figure 1.1: Typical EMPS–PhyDAS configuration. The EMPS system consists of several computer modules (C) and shared memory modules (M), interconnected by a cluster bus. A system bus interconnects several clusters into a node. Each node has a system controller (SC), an inter-node communication module (IN), and a local area network controller (LAN). A detailed description of these modules is presented in Chapter 2. A converter (CNV) interconnects the system bus of the EMPS system and the PhyBUS of the PhyDAS system.

Goals for the design of PhyDAS were:

1. **Real-time processing**: The system meets real-time demands by guaranteeing that worst-case response times are bounded.

2. **Modularity, extendibility**: Different interface modules can be used to construct the required hardware configuration for a given experiment.

3. **Multiprocessor configuration**: To increase the performance and reliability of the system, multiple processors can be used for controlling
the interface modules and processing the data.

4. **Cost-effectiveness**: The life cycle of computers is generally much shorter than the life cycle of the interface modules. By separating the computer bus and the measuring bus, the use of new microprocessor architectures is very cost-effective since the same interface modules can be used.

EPEP is the name of the operating system and the programming language. Goals for the design of EPEP [1] were

1. **Object-based**: The implementation of EPEP is object-based for reasons of modularity and extendibility. The programming language, which is not object-based, is a Pascal-like language.

2. **Interpreter language**: An interpreter language is preferred rather than a compiler-language because it is often necessary that the settings of interface modules can be adapted while running an experiment, without stopping and re-compiling the program.

3. **Multitasking operating system**: The motivations for structuring EPEP as a multi-process system are:

   - In a real-time system, processes must respond to asynchronous events. Asynchrony can be reflected in the structure of the control system using multiple processes: one handler process for each kind of event. EPEP provides the necessary interprocess communication facilities.
   - Programs can be structured with multiple processes of different priorities reflecting different priorities of activities in the system.
   - Concurrent execution may reduce the completion time of a program and also improve the utilization of program resources.

4. **Real-time facilities**: The operating system provides real-time facilities, viz. priority scheduling, real-time clock support, primitives to delay processes for a fixed time, and timeouts [2].

5. **Distributed computing**: The performance and reliability can be improved by distributing processes across the available resources (processors). In future versions, EPEP will provide primitives for interprocess communication via the network.

PhyDAS and EPEP are currently used in a single-processor environment for data-acquisition and control of approximately 50 experiments
within the Department of Physics, e.g. plasma physics, fluid dynamics, solid state physics, and atomic spectroscopy. Triggered by the need for increasing computing power of the users of the single-processor system, a research project for the development of a multiprocessor executive for distributed computing was initiated in 1988.

1.1.2 Dependable Distributed Computing

The Department of Mathematics and Computer Science of the Eindhoven University of Technology initiated a research project for the study of distributed real-time systems. Goal of the project is the research on and development of paradigms and formalisms for the correct construction of embedded distributed systems. Not only the functionality of the system is modeled, but also methods are developed to model the reliability and timeliness aspects of a system. One of the first stages in this project is the development of a DEpendable Distributed Operating System (DEDOS) [3]. DEDOS focuses on operating system architectures and techniques to support reliability and real-time requirements for application software. The intended application areas of DEDOS are in the domain of embedded systems and process control. The driving forces behind the DEDOS project are (1) the external demand for dependable distributed control systems, and (2) the necessity to increase drastically the productivity and quality of application programming for distributed control.

Some of the main issues of DEDOS are

- **Transparency:**
  The key concept in DEDOS is transparency, which comes in several forms, viz. *access transparency*, which allows the access of resources by logical names and independent of their physical location, *execution transparency*, which means that a user is not aware of whether his executions are performed locally or remote, *fault transparency*, which means that faults within a given fault hypothesis are hidden, *replication transparency*, which means that the distributed operating system maintains the consistency of replicated resources and chooses the shortest access path, and finally *scaling transparency*, which means that the performance of the system increases almost linearly with its size.

- **Dependability:** DEDOS is based on the dual-dependability model for the efficient support of timeliness and reliability.
  
  - *Timeliness:* Two types of executions are distinguished in DEDOS, viz. hard real-time (HRT) and soft real-time (SRT) executions.
1.1 Application areas

HRT executions have to adhere to strict deadlines. The missing of a deadline is a catastrophic failure, which results in a system crash. DEDOS guarantees these deadlines by executing programs deterministically at times specified by an off-line calculated schedule. SRT executions are not as time critical. The missing of a deadline is acceptable, and causes only performance degradation. SRT executions can therefore be scheduled dynamically (on-line) using a preemptive priority scheduling policy.

- **Reliability**: Reliability of DEDOS applications is improved by tolerating hardware faults. Two types of fault tolerance can be distinguished for DEDOS, viz. fault masking and fault recovery. Fault masking guarantees that an uninterrupted service is delivered in spite of hardware faults by static replication of service providers. Fault recovery provides dynamic reconfiguration and forward or backward recovery, which results in graceful degradation of a service. Fault tolerance is supported in two DEDOS layers, viz. (1) in the communication layer, where messages are reliably transported, and (2) at the application level, where fault-masking is used for HRT executions and fault recovery is used for SRT executions.

- **Object oriented**: The programmers’ perspective to DEDOS is based on the object oriented programming paradigm for the following reasons:
  - Well known software engineering advantages, viz. modularity, reusability, and extendibility.
  - Fine grained concurrency control is achieved by the concurrent execution of objects. This enhances the possibility to meet the timeliness constraints.
  - Objects are elementary units with respect to mutual exclusion, atomicity, and recoverability.
  - An object is more amenable to the analysis of timing properties since it is a small building block that unifies different but related functionalities.

The DEDOS system has a layered structure (Fig. 1.2). The kernel provides the basic abstractions through which the underlying hardware is manipulated. It is essentially based on the EMPS kernel described in this thesis. Two communication paradigms are used in DEDOS, viz. Remote Procedure Call (RPC) [4] and Atomic Multicast (AM) [5]. The communication layer assures that all correct processes have the same view of the
correct nodes, processors, and processes. Messages from processes, processors, or nodes that are not belonging to a group of correct and consistent members are neglected.

The timely execution of actions and especially of a distributed precalculated HRT schedule requires the availability of synchronized clocks. The clock synchronization algorithm (CSA) for DEDOS is based on the probabilistic CSA [6]. Clock synchronization messages can be piggybacked on RPC messages, thus avoiding an extra overhead. Each processor compares its clock rate to the rate of the global clock and adjusts its own clock’s drift rate accordingly. The scheduler layer contains the dispatcher for both HRT and SRT threads.

The C++ language has been extended to support DEDOS specific features. Language constructs related to concurrency, real-time and reliability are added in DEAL (DEDOS Application Language). Future work will include the design and implementation of a high-level object-oriented application language that integrates object-management, real-time, and reliability.

In order to test and evaluate the DEDOS concepts, they will be implemented on top of the EMPS kernel described in this thesis.
1.2 Distributed operating systems

Because of the recent advances in micro–electronic technology, viz. the development of powerful microprocessors at a moderate price level, and the development of high–speed local area networks (LANs), it is attractive to construct a computer system consisting of many parallel running microprocessors. These systems are called distributed systems, in contrast to centralized systems consisting of a single CPU, its memory, peripherals, and terminals. Distributed systems potentially have a much better price/performance ratio than a large centralized system would. A distributed system can also yield an absolute performance that no single centralized system can achieve at any price [7]. Another potential advantage of a distributed system over a centralized one is higher reliability: by distributing the workload over many machines, the crash of one machine does not have to lead to the crash of the system as a whole. Another advantage of distributed systems is expandability: Computing power can be increased by adding more machines to the system. A potential disadvantage of distributed systems is that they need radically different software compared to centralized systems. The required operating systems for distributed systems are only beginning to emerge. Other potential disadvantages include the interconnection network, which can saturate or malfunction, and security, because easy access also applies to secret data. Despite these disadvantages, it is expected that distributed systems will become increasingly important in the next decade.

The software for multiple computer systems can be divided into three classes, viz. network operating systems, distributed operating systems, and multiprocessor operating systems. Network operating systems allow users at independent machines to access shared resources and to communicate via a shared file system, but otherwise leave each user as the master of his own machine. The main goal of distributed operating systems is to take a collection of computers, and have them behave like one centralized system, yet keeping intact the advantages of distribution, e.g. performance and fault tolerance. Shared–memory multiprocessor systems also offer a single system image, but do so by centralizing everything. Therefore, these systems are not truly distributed systems. In this thesis, we focus on the class of multiprocessor operating systems, however, with the mechanisms and handles necessary for the extension to a distributed operating system. Examples of well–known distributed operating systems are Amoeba [8, 9, 10], Mach [11, 12, 13], V [14, 15, 16], Chorus [17], and Clouds [18]. A comprehensive summary of distributed operating systems can be found in [19].

The main goal of Amoeba was to develop a capability–based, object–
based distributed operating system, which is transparent, reliable, and performs well. The Amoeba hardware architecture is based on the processor pool model. A processor pool consists of a substantial number of CPUs, each with its own local memory and network connection. Amoeba has been designed to deal with multiple architectures and heterogeneous systems. The Amoeba software architecture consists of two basic pieces, viz. a microkernel, which runs on every processor, and a collection of servers that provide most of the traditional operating system functionality. The Amoeba microkernel supports process management, low-level memory management, interprocess communication, and low-level I/O. Amoeba supports multiple threads within a single address space. Threads can allocate and deallocate blocks of memory, called segments. The microkernel supports point-to-point and group communication. Point-to-point communication is based on the client/server model: The client sends a request message to a server, and is blocked until it receives a reply message back from that server. This request/reply communication is the basis on which almost everything is built. The Amoeba kernel is small, simple and very fast. The basic unifying concept underlying all Amoeba servers and services they provide is the object. An object is a encapsulated piece of data on which well-defined operations can be performed by authorized users, independent of the user’s and object’s location. Objects can be managed by server processes and they are named and protected by using capabilities. Operations on objects are implemented as remote procedure calls.

Similar to EMPS, Mach was designed with the intention of integrating both distributed and multiprocessor functionality. The Mach microkernel has been built as a base upon which other operating systems (e.g. UNIX) can be emulated. The emulation is done by a software layer that runs outside the kernel. The Mach kernel provides process management, memory management, communication, and I/O services. The idea behind the kernel is to provide the necessary mechanisms to make a working system, but leaving the policy to user-level processes. The Mach kernel supports five basic abstractions, viz. process, thread, memory objects, ports, and messages. These abstractions were chosen for simplicity and performance reasons. A process is an environment in which execution can occur. A process has an address space containing the program code and data, and usually one or more stacks. A thread in Mach is an executable entity which can run concurrent with other threads and has a program counter and a set of registers associated with it. A concept that is unique to Mach is the memory object, a data structure that can be mapped into a process’ address space. Memory objects form the basis of the Mach virtual (secondary) memory system. Interprocess communication in Mach is based on message passing. To receive messages, a user process asks the kernel to create
1.2 Distributed operating systems

A **port**, which is stored inside the kernel and has the ability to queue an ordered list of messages. A comparison between Amoeba and Mach can be found in [20].

The V system consists of a distributed kernel and a distributed collection of server processes. A functionally identical copy of the V kernel resides on each processor. The V kernel is small and provides high performance communication and can be used by servers which offer most operating system facilities. The communication protocol uses a fixed-length message and a process identifier that uniquely identifies the destination process. The V kernel uses blocking message transactions: A client sends a request message to a server and is blocked until it receives a reply message back from that server. The V kernel also supports group communication. All input and output operations use a uniform, block-oriented interface, called UIO. The V kernel provides the concept of **process groups** and **teams**. Process groups are arbitrary sets of processes that are grouped using kernel support so they can be addressed using a single group identifier. For instance, one can send a message to a process group, or one can kill a process group. A team is a set of processes sharing the same address space. Teams are used for efficient data-sharing. The V system has no special security issues like capabilities in Amoeba and Mach. The V system provides a migration facility for team address spaces.

The Chorus system has been designed for building new generations of open, distributed, scalable operating systems. Chorus has a communication-based architecture, relying on a minimal nucleus which integrates distributed processing and communication at the lowest level, and which implements generic services used by a set of subsystem servers to provide standard operating system interfaces (e.g. UNIX). The Chorus kernel structures distributed processing into **actors**. Actors are resource capsules. The executing units in Chorus are threads, which are encapsulated in actors. Threads communicate and synchronize within an actor through shared memory, or between actors through exchange of messages. The kernel supports both asynchronous exchange of messages and remote procedure calls. Threads send and receive messages through **ports**, which provide the basis for point-to-point communication. A port group is a collection of ports. Multicast and broadcast communication are provided through port groups. The Chorus kernel provides three basic entities, viz object, actor, and port. Each of these entities is uniquely identified with a globally unique name. Objects can be accessed as soon as the actor has a capability of the object.

The Clouds distributed operating system is a general purpose distributed computing environment for a wide variety of users. The Clouds system structure is based on the object-thread model, which is based on
the object-oriented programming model. Clouds objects provide long-term storage for persistent data and associated code. The object-thread model treats storage and computations as orthogonal concepts. Clouds objects are large-grained encapsulations of code and data, which respond to invocation. An invocation results from a thread of execution entering the object to execute an operation. Threads are the only active entities in the model. The Clouds implementation has three levels, viz. (1) a microkernel, which provides mechanisms for managing the basic resources processor and memory, (2) a set of system objects, which are software modules that provide essential system services, and (3) a set of user objects that provide noncritical services.

1.3 Design philosophy

In 1988, the Department of Physics of the Eindhoven University of Technology initiated a research project for the development of a multiprocessor executive for distributed computing. The Eindhoven Multi-Processor System (EMPS) project includes the design of both the hardware architecture and the software architecture of the multiprocessor executive.

The main goal of the EMPS project was to develop a (single-user) testbed, consisting of a large number of computers, that can be used for a large variety of application and research areas, e.g. real-time data-acquisition and control of physics experiments (Sec. 1.1.1) as well as dependable distributed computing (Sec. 1.1.2).

The hardware architecture consists of loosely-coupled nodes. Each node is a tightly-coupled multiprocessor system. The presence of both multiprocessor and distributed aspects permits the investigation of the behavior of e.g. real-time applications under either architecture. Because the EMPS testbed will be used for a large variety of applications, flexibility and scalability of the hardware architecture are required. Therefore, the EMPS hardware architecture is designed to provide a collection of general purpose modules (e.g. computer and memory modules) that can be used to construct the desired distributed configuration for a certain application simply by adding or removing modules. In order to achieve efficient interprocessor communication, the EMPS hardware architecture provides common memory and dedicated hardware facilities. The EMPS hardware architecture provides multiple, independent communication paths between processors to support reliability.

The first stage in the development of the software architecture was the design and implementation of a distributed real-time operating system kernel. For reasons of flexibility, which is for instance required for fun-
1.3 Design philosophy

damental research on dependable distributed computing, a new kernel has been designed from scratch rather than modifying an existing kernel. Although each node of the EMPS system has shared memory, the EMPS kernel was designed to be a distributed kernel primarily for performance reasons: Accesses to shared memory are less efficient than accesses to local memory. For efficiency reasons, the kernel was based on a microkernel [20] directly on bare hardware rather than as a monolithic kernel.

The main goals in the design of the EMPS kernel were:

- **Simple and efficient**: By keeping the kernel small and simple, not only its reliability is enhanced, but also many parts of the operating system can be run as user processes, thus providing for flexibility and extendibility. The kernel exploits the facilities provided by the hardware architecture in order to establish efficient interprocess communication.

- **Process migration**: In order to support failure recovery and load balancing, the kernel has been designed to provide efficient process migration. This is implemented by using a simple process structure and by providing location-transparent interprocess communication using mailboxes.

The new issues that we address in this thesis are:

- **Object-oriented design**: In the literature, many papers on operating system kernels can be found. These papers describe the kernel interface, i.e. the system calls and their parameters. Although these descriptions usually provide sufficient information to use the kernels, they do not provide any insight in their internal structure: How do different parts of the kernel interact? What data structures are used? In this thesis, we use a different approach to describe the design of the EMPS operating system kernel. We describe both the kernel interface and its internal structure, using the object-oriented paradigm. Although the design of the kernel is based on the object-oriented paradigm, its implementation is done in C for efficiency reasons.

- **Location-transparent communication via mailboxes**: In this thesis, we describe the implementation of a new protocol for interprocess communication using mailboxes. The protocol uses these mailboxes not only for buffering, but also for location-transparent addressing. The protocol has been designed to support both efficient communication and efficient process migration.
1.4 Development stages

The development of the multiprocessor executive can be divided into several stages (Fig. 1.3). The first stage consists of the design and the development of the various modules of the hardware architecture. In the second stage, the EMPS real-time multiprocessor operating system kernel that exploits the facilities provided by the hardware architecture is designed and implemented. This kernel is the software testbed which forms the base for research on dependable distributed computing and on real-time data-processing and experiment control. In the third stage, this testbed is used for the development of DEDOS and for the development of a distributed version of EPEP. In the final stage, distributed applications are developed on top of DEDOS and EPEP.

<table>
<thead>
<tr>
<th>Stage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Distributed applications</td>
</tr>
<tr>
<td>3</td>
<td>EPEP</td>
</tr>
<tr>
<td></td>
<td>DEDOS</td>
</tr>
<tr>
<td>2</td>
<td>EMPS kernel</td>
</tr>
<tr>
<td>1</td>
<td>Hardware architecture</td>
</tr>
</tbody>
</table>

Figure 1.3: Design stages

1.5 Overview

In this thesis, we describe the design of the EMPS multiprocessor executive. Research has concentrated on a number of different areas, e.g. the hardware architecture, the software architecture, interprocess communication, and process migration. The structure of this thesis is as follows:

In Chapter 2, we present the motivations that have led to certain choices in the design of the hardware architecture for the EMPS multiprocessor
executive. We also describe the different modules that can be used to construct an EMPS hardware configuration, viz. computer module, memory module, system controller, local area network controller, and inter-node communication module.

Chapter 3 is devoted to the design of the software architecture of the EMPS multiprocessor executive. This chapter describes the executive services provided by the distributed real-time operating system kernel using the object oriented paradigm.

In Chapter 4, the implementation of interprocess communication via mailboxes is described using the object oriented paradigm. The EMPS kernel supports location-transparent communication between application processes, which is required for process migration.

Chapter 5 describes the design and implementation of the process migration facility in the EMPS system. The performance of the process migration facility is compared to process migration facilities in other distributed systems.

In this thesis, we use a graphical representation for classes and their relationships [21]. A summary of this representation is presented in Appendix A. The Eiffel object oriented programming language [22] is used to describe the class definitions for the different parts of the EMPS kernel. A summary of used class interfaces from the Eiffel library is described in Appendix B. The Eiffel implementation of the class definitions, presented in chapter 3 and chapter 4, is described in Appendices C and D, respectively.
Design of the Hardware Architecture

In this chapter, we describe the design of the hardware architecture of the EMPS multiprocessor executive. Although, in this thesis, we focus on the design of the software architecture, a description of the design of the hardware architecture is indispensable. The EMPS multiprocessor executive will be used as a testbed for research on dependable distributed computing (by means of the DEOS system), real-time data-acquisition, and real-time process control, among them the control of physics experiments. Using commercial hardware generally imposes undesirable limitations, e.g. in modifying the architecture of the system, in improving the performance of the system, and in designing the operating system software. Moreover, the absence of standards in multiprocessor systems makes it hard to combine hardware of different suppliers. It is therefore desirable to develop an integral system, i.e. both hardware and software. In 1988, triggered by the need for increasing computing power of the users of the single-processor PhyDAS system, the Department of Physics of the Eindhoven University of Technology initiated a research project for the development of a multiprocessor executive for distributed computing.

The main issues in the design of the EMPS hardware architecture are flexibility, scalability, efficiency, and reliability. Because the EMPS testbed will be used for numerous application and research areas, flexibility and scalability of the hardware architecture are required. Efficiency and scalable performance are required for real-time data-acquisition and control. In order support dependable computing in general and fault tolerance in particular, the hardware architecture must be able to support redundancy.
In Sec. 2.1, we present an overview of parallel processing architectures. Two basic architectural issues, viz. the organization of system memory and the interconnection network are described.

Sec. 2.2 discusses the motivations that have led to certain choices in the design of the hardware architecture using the information presented in Sec. 2.1. Also the different modules that can be used to construct an EMPS hardware configuration, viz. computer module, memory module, system controller, and the local area network controller are described in detail.

2.1 Parallel processing architectures

Three basic approaches to parallel processing architectures can be distinguished [23], viz. von Neumann–based, dataflow, and reduction approaches (Fig. 2.1). The von Neumann–based approach consists of interconnecting two or more von Neumann–type uniprocessors. In a von Neumann–type processor, the sequence of executions is dictated by the ordering of program instructions. The dataflow approach is based on the concept of executing program instructions as soon as there operands are ready (data–driven). An example of an implementation of a dataflow architecture is the Manchester dataflow computing system [24]. The reduction approach consists of carrying out instructions when results are needed for other computations (demand–driven).

The majority of parallel processing systems are based on processors built on the von Neumann principle. According to Flynn’s taxonomy [25], these systems can be further classified according to how they process their instructions and data streams, viz. Single Instruction Single Data (SISD), Multiple Instruction Single Data (MISD), Single Instruction Multiple Data (SIMD), and Multiple Instruction Multiple Data (MIMD).

SISD computers are the traditional von Neumann uniprocessors. An MISD processor works according to the principle of pipelining, which implies the partitioning of instructions into simpler computational steps that can be executed sequentially by independent computational units. In SIMD architectures the same instruction is performed on all items of a given data structure. This category includes e.g. vector/array processors, pipelined array processors, and the Connection Machine [26]. The MIMD category of parallel processing architectures provides flexibility. Therefore, most multiprocessor systems, including the EMPS multiprocessor system, can be classified in this category.

Two basic architectural issues in parallel processing systems can be distinguished, viz. the organization of system memory and the interconnection network.
2.1 Parallel processing architectures

![Classification of parallel processing architectures](image)

Figure 2.1: Classification of parallel processing architectures [23]

2.1.1 Organization of system memory

If the system has a global memory that is shared by and accessible from all the processors, the system is referred to as a tightly coupled parallel system. If each processor has only local memory and inter-processor communication is based on messages, the resulting system is loosely coupled.

Tightly coupled multiprocessor systems have the advantage of sharing code and data structures among processes comprising the parallel application. Processes can communicate efficiently by exchanging information through shared variables. The tightly coupled multiprocessor organization, however, has three disadvantages [27] that contribute to increased memory access times and hence degrade the overall system performance:

1. **Memory contention**: Several requests from different processors are serialized because a memory module can handle only one request at a time.

2. **Communication contention**: Contention for individual links in the interconnection network can result.

3. **Latency time**: Multiprocessor systems with a large number of processors tend to have a complex and slow interconnection network.

The average memory access times can be reduced by associating a cache
with each processor to satisfy most of the memory references made by the processors.

Processors in tightly coupled multiprocessor systems communicate through shared variables. This can result in several copies of a shared block in one or more caches at the same time. Although processor caches can significantly improve system performance, they introduce a coherence problem. A special protocol is necessary to ensure that changes made to shared memory locations by an arbitrary processor are visible to all other processors.

Processors in loosely coupled systems communicate through message passing: A processor requests another processor to modify or transmit a variable by sending a message via a network. Message-passing is slow compared to shared memory communication. Loosely coupled systems are most efficient when the interactions between tasks are minimal, whereas tightly coupled systems can tolerate a higher degree of interactions between tasks.

A potential advantage of a loosely coupled system over a tightly coupled multiprocessor is higher reliability. In a tightly coupled multiprocessor, the failure of the common memory can lead to a failure of the whole system. By allocating redundant processors and communication links, the failure of one machine in a loosely coupled system does not have to lead to the failure of the whole system.

The EMPS hardware architecture consists of a mixture of tightly and loosely coupled subsystems: Tightly coupled nodes are interconnected by means of a local area network. The EMPS system benefits from the advantages of both tightly coupled systems, viz. efficiency and ease of programming, and loosely coupled systems, viz. expandability and reliability.

### 2.1.2 Interconnection network

The interconnection network links processors, memory modules, and I/O controllers, and has a profound impact on the performance, size, and cost of the parallel system. Hence MIMD systems can be further classified depending on how processors and memories are interconnected. MIMD architectures can be distinguished that use e.g. a bus, crossbar, multistage network, or direct interconnection between processors and memories.

The full crossbar provides a separate path between any two system components at all times and therefore has the best performance of all interconnection systems but at the expense of complexity, size, and cost.

In multiprocessor systems that have a direct interconnection between the processing elements, generally a direct path is only provided between two
2.1 Parallel processing architectures

adjacent system components because of the limited number of communication links of each processing element. Examples of such systems are processor arrays and hypercubes. The hypercube network is used in multiprocessor systems because it allows many processors to communicate relatively quickly using a reasonable number of wires.

A multistage network consists of multiple stages of switches capable of providing at least one direct path between any two system components with minimum complexity and cost. Although system performance may degrade as a result of contention for one or more switches, it has been demonstrated that properly designed switches achieve nearly full crossbar switch performance at a small fraction of the hardware cost [28]. Examples of tightly coupled multiprocessor systems that use a multistage interconnection network are Monarch [29], RP3 [30], and Ultracomputer [31].

The simplest interconnection for MIMD systems is a common communication bus connecting all system components. A shared bus, however, can only provide one path between two system components at a time and therefore has the lowest performance. Especially in this type of systems, the system performance can be improved considerably by using processor caches.

Parallel systems based on a bus interconnection are attractive for several reasons, viz. low system cost, standard interconnection allowing independent board design by different vendors, and incremental computing power by additional boards. A number of standard buses have been introduced, viz. VME bus, Multibus, and IEEE Fastbus. None of these buses, however, is considered fully adequate for powerful multiprocessor systems with local caches and a shared global memory. Therefore, IEEE has set up the Futurebus Standards Committee and many manufacturers have decided to develop their own high performance buses. Examples of bus-based multiprocessor systems are Encore Computer’s Multimax, Sequent Computer Systems’ Symmetry, and Alliant Computer Systems’ FX/8.

2.1.2.1 Hierarchical cluster systems

The shared bus is the potential bottleneck in a bus-based multiprocessor system that prevents the physical expansion of the system beyond a certain limit. Therefore, extensions to the single-bus architecture are required to increase the capacity of bus-based parallel systems. Processors can be organized in groups (clusters) with a local cluster memory shared by each group of processors. Communication between processors within the same cluster is done through the cluster memory. Processors in different clusters communicate through global memories.

For many applications it is possible to assign tasks in such a manner
that interprocessor communication occurs mainly within small groups of processors. For these applications a hierarchical cluster system yields an improved performance. It is clear that efficient utilization of resources in a multiprocessor system requires a match between the architecture and the application.

Examples of systems that use a hierarchy of buses are Cedar [32] and Cm [33].

2.2 EMPS Hardware architecture

Because of the recent advances in micro-electronic technology, powerful microprocessors have become available at a moderate price level, so that it becomes attractive to construct a computer system consisting of many parallel running microprocessors. In 1979, the Department of Physics developed the PhyDAS system [34], that has been used satisfactorily for real-time data-processing and control of physics experiments. The single-processor PhyDAS system uses the Motorola MC68000 microprocessor that is interconnected to a memory module using the standard VME-bus interconnection. In the past decade, the PhyDAS system has been used in approximately 50 physics experiments. Not only hardware know-how and experience have been obtained, but also a lot of software has been developed [1, 35]. For the reasons mentioned above, the software compatible Motorola MC68030 microprocessor has been chosen for the hardware architecture of the EMPS multiprocessor executive.

The MC68030 microprocessor offers facilities for multiprocessor synchronization. Therefore, it was possible to choose a tightly-coupled multiprocessor organization that can tolerate a high degree of interactions between tasks. The VME-bus interconnection has been chosen for reasons of standardization, low cost, flexibility, scalability, and already existing experience.

As already mentioned in Sec. 2.1.2, the shared bus is the potential bottleneck in a bus-based multiprocessor system, that prevents the physical expansion of the system beyond a certain limit. Therefore, a hierarchy of communication networks has been chosen. At the lowest level, a computer bus connects the processor to its on-board memory, forming the computer module. Several computer modules and shared memory modules are interconnected by a cluster bus (VSB) to form a cluster. Thus, each cluster forms a tightly coupled multiprocessor. In this thesis, the shared memory modules will also be referred to as common memory. The hardware system configuration can easily be modified by adding or removing several modules. To prevent saturation of the shared bus, the number of computer and
memory modules that can be interconnected by the cluster bus is limited to five. At the next level, a system bus (VME) interconnects several clusters (a maximum of four) into a node. A possible node configuration is sketched in Fig. 2.2. By storing local information in the fast on-board memory of a computer module, the number of accesses to the common memory via the buses is decreased, thereby increasing the effective bus bandwidth of those buses. Each node contains a local area network controller (Sec. 2.2.4) that is used for the downloading of programs from a central file server into common memory. In addition, each node also has a system controller (Sec. 2.2.3) that performs bus arbitration and monitors the behavior of the multiprocessor node. Although the number of processors in a node is limited, in order not to degrade the overall system performance by bus contention, the scalability objective is not violated, since nodes can be interconnected (Sec. 2.2.5) by means of a local area network (LAN) or by means of fast point-to-point communication channels using direct links (Fig. 2.3). In this way, the system can, in principle, be infinitely extended. A controller for point-to-point communication channels requires less complex hardware than a controller for broadcast LANs, because the latter has to provide additional (complex) hardware for addressing and arbitration [36]. Therefore, direct links provide high-performance, reliable commu-
Figure 2.3: Nodes can be interconnected by means of a local area network or by means of fast point-to-point communication channels.

cation, and can be used for time-critical communication. The LAN can be used e.g. for communication with the development system.

In the remainder of this section, the computer module, the memory module, the system controller, the local area network controller, and the inter-node communication module of the Non-Uniform Memory Access (NUMA) EMPS system are described.

2.2.1 The computer module

The basic building block of the computer module (Fig. 2.4) is the state-of-the-art 32-bit MC68030 microprocessor [37]. This general purpose CISC ("Complex Instruction Set Computer") microprocessor integrates a memory management unit that supports a demand-page virtual memory operating system with a 4 Gbyte linear address space, a high-performance integer processing unit, and two independent caches for instructions and data, respectively. The MC68882 floating point coprocessor provides full support of the IEEE floating-point format. The instruction set of the MC68030
Figure 2.4: The block diagram of the computer module. The module contains two communication registers for interprocessor communication via the VME system bus and the VSB cluster bus, respectively. The debug register is used for the debugging of data items.

The processor offers facilities for multiprocessor synchronization. These consist of special instructions that use read-modify-write cycles to ensure uninterrupted updating of memory. The TAS (Test and Set) instruction can be used to test and set a common memory flag like a semaphore. The CAS/CAS2 (Compare and Swap) instructions can be used to manipulate single and double linked lists respectively.

Each computer module has 2 Mbyte of static RAM memory (cycle time 150 ns), the private memory. This private memory can only be accessed by the processor residing at the same computer module. The private memory can be used for storing local information, e.g. a local copy of the operating system kernel, the tree structures of the translation tables used by the memory management unit, and the process stacks.

An on-board 512 kbyte PROM memory contains a monitor program that executes at system start-up and provides facilities for initialization of the multiprocessor node and for downloading the operating system kernel into the private memory.

Each computer module has the appropriate interfacing to both the
32-bit VME system bus [38] and the 32-bit VSB cluster bus [39]. The system bus includes a high speed (typical bandwidth 100 Mbyte/s at 25 MHz) asynchronous parallel Data Transfer Bus (DTB). The cluster bus includes a high speed time-multiplexed DTB. The cluster bus includes three geographical address lines. Each module drives these lines according to its position in the cluster. Both the system bus and the cluster bus allow for read-modify-write cycles. Arbitration prevents the simultaneous use of a bus by more than one module and schedules multiple requests. The system bus is assigned by a centralized arbiter (the system bus interface controller at the system controller) to a requesting module according to a serial round-robin scheme. In this way fair arbitration between contending computer modules can be guaranteed. In contrast to the system bus, the cluster bus has two arbitration modes: parallel and serial. If parallel arbitration is used for the assignment of the cluster bus, all contending bus requesters participate in an arbitration cycle to establish the allocation of the cluster bus (in contrast to one centralized arbiter for serial arbitration). Parallel arbitration has the advantage over serial arbitration that a bus master does not need to compete again for bus mastership after each bus cycle if no bus requests from other processors are pending.

The set of processors in a multiprocessor system must have efficient means for interprocessor communication. In such systems, generally bus interrupts are employed to attract the attention of a target processor. The on-board programmable interrupt handler can handle up to a maximum of six interrupts from local devices (i.e., devices directly connected to the computer module) and a maximum of seven system bus interrupts (Cluster bus interrupts are not supported). Since the number of interrupt levels of the system bus is limited and the effective bus bandwidth is decreased by interrupt handling, interprocessor communication using system bus interrupts is not very efficient. Therefore, the computer module of the EMPS system is equipped with two hardware communication registers for interprocessor communication via the system bus and via the cluster bus, respectively. These communication registers serve as input registers of a computer module for data arriving via the system or cluster bus. (Chapter 4). A processor can write a longword (32 bits) directly into the communication register of the target computer module. A local interrupt is generated to attract the attention of the target processor. Data blocks can be exchanged through common memory by means of an appropriate protocol that synchronizes the processors by using the communication registers.

A Dual Universal Asynchronous Receiver/Transmitter (DUART) is used to interface with slow devices, e.g. terminals and printers. Especially in the development phase of the multiprocessor system, it is essential that a terminal can be connected to each computer module independently. The
2.2 EMPS Hardware architecture

DUART also integrates a programmable system clock (3.6864 MHz). This system clock can be programmed to periodically generate interrupts (e.g. each 20 ms). A detailed description of all possible modes of the system clock is beyond the scope of this thesis.

Extra hardware facilities are provided on the computer module that are used for debugging purposes, viz. debug registers. The MC68030 microprocessor supports breakpoints and single instruction execution through exceptions [37]. Although these facilities are sufficient for the debugging of code, additional tools for the debugging of data structures (variables) are desirable in more complex systems. The address of a variable can be written into a debug register. This corresponds to the setup of a watchpoint. The debug registers can monitor the address buses of the system, viz. the computer bus, the cluster bus, and the system bus, respectively. If a watchpoint is set, a local interrupt is generated every time that the corresponding address is accessed.

Each computer module is equipped with four 20 Mbit/s serial point-to-point communication channels (Transputer links [40]), and a four-channel Direct Memory Access (DMA) controller. These links of a computer module can be used for high-performance inter-node communication (Sec. 2.2.5). The DMA controller transfers data directly between the private memory and the links. After completion of a transfer, a local interrupt is generated. The DMA controller can also be used to transfer large blocks of data between private and common memory.

2.2.2 The memory module

The memory module of the EMPS multiprocessor system is a high-performance dual-ported DRAM memory module, providing 4 Mbyte (extendible to 64 Mbyte) of storage with byte parity check. The module has the appropriate interfacing to both the cluster bus and the system bus. The memory module contains a three level arbiter for refresh, cluster bus requests, and system bus requests. The cycle times for the system bus and cluster bus are 500 ns and 600 ns, respectively. Read-modify-write cycles are supported via both buses to ensure uninterrupted updating of memory. Each memory module has separate address spaces for cluster bus and system bus. The address space of the memory in the cluster bus is determined uniquely by the geographical address of the module. For this purpose, a memory module contains two switches that are set to the number of the cluster in which the module is located within the node. The address space of the memory module in the system bus is therefore determined uniquely by a combination of the setting of the switches and the geographical address in the cluster bus.
2.2.3 The system controller

The hardware architecture of the system controller is identical to that of the computer module except for the presence of additional facilities for the arbitration of the system bus. In addition, the system software for the EMPS system is designed in such a way that interrupts on the system bus, e.g. from the local area network controller, are handled exclusively by the system controller.

2.2.4 The local area network controller

Each node in the EMPS system has a local area network controller that is used for the downloading of programs from the file server of the development environment into common memory via the local area network PhyLAN. PhyLAN is also used for inter-node interprocess communication (Chapter 4). PhyLAN is a real-time time-division multiple-access network, that connects up to 255 computer systems by means of a bus topology, and has a transmission rate of 2.5 Mbit/s.

Allocation of the transmission medium of PhyLAN is based on the concept of cable master. This cable master can communicate directly with any slave station. Software determines which station is the cable master through a bit in the control and status register of the PhyLAN controller.

PhyLAN is also used for inter-node interprocess communication. A mastership-passing protocol [41, 42] establishes direct communication between any two nodes that are connected to PhyLAN.

2.2.5 Inter-node communication

A local area network controller (Sec. 2.2.4) or a computer module, provided with four high-speed links (Sec. 2.2.1), can be used for inter-node communication. The protocols for inter-node interprocess communication via PhyLAN are described in Chapter 4. In the near future, protocols will be developed for inter-node communication via a different local area network, viz. Ethernet. Protocols for communication via the serial links of a dedicated computer module will also be developed in the near future.
3

Design of the Software Architecture

In this chapter, we describe the design of the software architecture of the EMPS multiprocessor system using the object-oriented paradigm. As already mentioned in Chapter 1, the EMPS system will be used as a testbed for research on various fundamental issues in distributed computing and their applications like dependable distributed computing for process control, real-time data-acquisition and control of physics experiments. Therefore, flexibility in modifying the software architecture of the multiprocessor system is required. The goal of the software design of the EMPS multiprocessor system is to provide a software testbed that has efficient and transparent facilities for interprocessor communication and resource management.

The first stage in the development of the software architecture of the multiprocessor executive is the design and implementation of the EMPS real-time distributed operating system kernel (cf. Chapter 1). Characteristics of most uniprocessor real-time kernels typically include the availability of fast context-switching, small size, multi-tasking facilities, process synchronization by semaphores, memory management without using secondary memory, priority scheduling, real-time clock support, primitives to delay processes for a fixed time, and timeouts [2]. Our distributed real-time operating system kernel not only includes the features for uniprocessor real-time kernels, but also features are included for interprocess communication. The EMPS kernel is referred to as distributed because its facilities are available uniformly and transparently across all computer modules.
Figure 3.1: The EMPS operating system.

The EMPS operating system can be represented by three layers (Fig. 3.1). Each layer provides a set of services dependent only on the lower layers.

The interface between the basic machine hardware and the operating system is provided by the nucleus, which is the core of the system. The nucleus (actually a microkernel [20]) provides low-level, non-distributed services for memory management (MM), process management (PM), process synchronization (PS), time management (TM), interprocess communication (IPC), and interrupt handling (INTH). The basic idea behind the design of the nucleus was to keep it small and fast. In this way, not only its reliability is enhanced, but also many operating system services can be provided by processes running on top of the nucleus (system processes), thus providing for flexibility and experimentation. System processes can invoke all services provided by the nucleus.

The second layer provides the transparent implementation of distributed process management services (PMS), mailbox management services (MMS),
and device management services (DMS). These services are provided by IPC via the system processes process server (PSRV), mailbox server (MSRV), and interrupt service processes (ISP.1 ... ISP.N), respectively. In addition to these distributed services, the kernel interface provides the non-distributed nucleus services for MM, PS, and TM via system calls. In Fig. 3.2, the general kernel interface of the EMPS operating system is sketched. Services provided by system processes are invoked by means of IPC: The client process sends a request message to the system process and waits for the reply message. The system process receives the request message, invokes the requested nucleus service, and responds with a reply message back to the client process. The EMPS kernel also provides device-independent I/O (Sec. 3.2.3). Nucleus services execute in privileged (supervisor) mode. Therefore, they can only be invoked via system calls. Kernel services and system processes execute in non-privileged (user) mode.

Figure 3.2: General kernel interface of the EMPS operating system. A number of nucleus services can be invoked locally via system calls. Other nucleus services can be invoked by means of IPC via a system process, either locally or remotely.

The third layer provides additional services (utilities), e.g. a file access
(FA) facility (Sec. 3.3.1), a loader, and a command interface. These utilities, which are used by the run–time (RT) system, have been implemented by processes that run on top of the kernel.

An important issue in the design of the real–time EMPS kernel is efficiency. In order to achieve this, the EMPS kernel is built directly on bare hardware, instead of building it on top of an existing operating system, e.g. UNIX. The design of the EMPS kernel was inspired by the Thoth [43, 44] real–time operating system which was designed to be portable over a large set of machines. The Thoth system achieves efficiency in the kernel by using a simple model of processes and messages.

Another important issue in the design of the kernel is support for process migration. In order to establish dynamic load balancing and failure recovery, application processes can migrate between different processors in the system. Migration requires that the communication between application processes does not depend on their physical location. Therefore, the kernel provides location transparent primitives for communication between application processes.

As already mentioned in Chapter 1, a number of operating system kernels have been described in the literature, e.g. [14, 45]. Although these descriptions present a discussion of all available system calls, they do not give a detailed overview of the internal structure of the kernels. In this chapter, we present a detailed description of both the programming interface and the internal structure of the EMPS kernel using the object oriented paradigm. Classes and their relationships are graphically represented using the method described in [21]. This graphical representation is summarized in the Appendix of this thesis. The Eiffel object oriented programming language [22] is used to describe the class definitions for the different parts of the EMPS kernel. This description also uses basic building blocks from the Eiffel library like lists and queues. The implementation of the EMPS kernel is done in C++.

### 3.1 The EMPS Nucleus

The EMPS nucleus interface defines the class MEMORY for memory management, the class PROCESS for process management, the class SEMAPHORE for process synchronization, the class TIMER for time management, the classes MAILBOX and PORT for interprocess communication, and the class INT_HANDLER for interrupt handling (Figs. 3.3 and 3.4). These class definitions are also used in the run–time library of application processes. Multiple instances of these classes can exist at run–time. The nucleus classes will be subsequently described in detail.
3.1 The EMPS Nucleus

Interface classes

<table>
<thead>
<tr>
<th>Memory management</th>
<th>Nucleus</th>
<th>Kernel</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROCESS MANAGEMENT</td>
<td>MEMORY</td>
<td>PROCESS.REFERENCE</td>
<td>PROCESS.INFO</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PROCESS.ADDRESS</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PROCESSOR.ADDRESS</td>
<td></td>
</tr>
<tr>
<td>PROCESS SYNCHRONIZATION</td>
<td>SEMAPHORE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TIME MANAGEMENT</td>
<td>TIMER</td>
<td>TIME</td>
<td></td>
</tr>
<tr>
<td>DEVICE MANAGEMENT</td>
<td>INT.HANDLER</td>
<td>DEVICE.REFERENCE</td>
<td>DEVICE.ADDRESS</td>
</tr>
<tr>
<td></td>
<td>MAILBOX</td>
<td>MAILBOX.REFERENCE</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>INITIATOR.PORT</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RESPONDER.PORT</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>KEY</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MESSAGE BUFFER</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.3: The interface classes of the EMPS nucleus/kernel and their parameter classes.

The routines provided by the nucleus classes PROCESS, MAILBOX, and INT.HANDLER can only be invoked by system processes (Fig. 3.4). Application processes can invoke the routines provided by the kernel classes PROCESS.REFERENCE, MAILBOX.REFERENCE, and DEVICE.REFERENCE, respectively, which provide distributed services for process management, mailbox management, and device management (Sec. 3.2).

The EMPS nucleus uses global objects (shared attributes) for the implementation of the interface classes. These global objects and their scopes are summarized in Fig. 3.5.
Figure 3.4: Graphical representation of the interface classes of the EMPS nucleus and kernel (Fig. 3.3). The classes above the dotted line provide nucleus services that are not present at the kernel interface. The classes below the dotted line provide either nucleus services that are also present at the kernel interface or distributed kernel services. Only the routines provided by the classes of the kernel interface are sketched.

3.1.1 Memory management

Real-time applications require that dynamic memory allocation is predictable. Virtual memory techniques using secondary memory can not be used for these applications, since swapping pages of memory to secondary storage can produce unpredictable response times. Also variable-block allocation schemes can produce unpredictable response times. Therefore, the EMPS system only supports dynamic memory allocation in fixed-size blocks.

The MC68030 microprocessor on the computer module integrates memory management hardware. The EMPS nucleus exploits this hardware for the following reasons:

1. Protection: Physical memory is shared between processes. Memory management hardware can be used to protect address spaces in order to prevent that a process overwrites the address space of other
3.1 The EMPS Nucleus

Scope

<table>
<thead>
<tr>
<th>Memory Management</th>
<th>free_private_memory</th>
<th>node</th>
<th>free_common_memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Management</td>
<td>process_table</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>scheduler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time Management</td>
<td>system_time</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>clock_queue</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interrupt Handling</td>
<td>int_vector_table</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interprocess Communication</td>
<td>service_manager</td>
<td>mailbox_table</td>
<td>lan_table</td>
</tr>
<tr>
<td></td>
<td>mac_manager</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>conreg_table</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.5: Global nucleus objects and their scopes.

processes.

2. Efficiency/Location independence: Programs can be loaded into physical memory at arbitrary locations. With memory management hardware, programs can use absolute addressing, which is more efficient than position-independent program code.

Memory management in the EMPS nucleus consists of two levels. The first level manages two arrays of available physical memory. The second level supports dynamic memory allocation for the process heap, and will be described in Sec. 3.1.2.

Each computer module in the EMPS system has private memory. Common memory is shared between all computer modules within the same node. Physical memory is subdivided into pages of 1 kbyte (Pagesize). Free physical memory is described by the class FREE_MEMORY (Fig. 3.6). At each processor, the nucleus maintains an array of free pageframes of private memory, free_private_memory. An array of free pageframes of common memory, free_common_memory, is maintained for each node. The Get
routine results (the physical address of) a pageframe of either private or common memory. The Free routine adds the pageframe to the corresponding array of free pageframes. Although the attribute free_pageframes is modeled as an array of pageframes, it is implemented (for efficiency reasons) as an array of bits, where each bit corresponds uniquely to a page frame.

Figure 3.6: At each processor, the nucleus maintains an array of free pageframes of private memory. An array of free pageframes of common memory is maintained for each node.

Because multiple processes may be allocating and releasing memory (either directly or indirectly by starting/killing other processes), mutual exclusion is required. The array free_private_memory is stored in the private memory of the computer module. Mutual exclusion is achieved by disabling interrupts to guarantee indivisibility of execution. Since the array free_common_memory is stored in common memory, interrupt disabling is not sufficient to ensure that only one process modifies the list. Mutual exclusion is achieved using the TAS and CAS instructions: The processors, that want to access that list, perform TAS instructions (busy-waiting loop) to test and set a flag. If the setting of the flag succeeds, the processor is allowed to access the list of free pages.
3.1.2 Process management

In the EMPS nucleus, processes are sequential programs, that can execute concurrently on a processor. Process management has the task to add new processes to the system, to start the execution of a process, and to remove a process from the system.

One of the main goals in the design of the EMPS kernel is the support for process migration (Chapter 5). A detailed description of different naming services that support process migration in e.g. V [14, 46], LOCUS [47, 48], and Sprite [49] can be found in Sec. 5.1. The V system uses global process identifiers that consist of a logical host number and a local identifier. LOCUS and Sprite use process identifiers that consist of the node number at which the process was created plus a local identifier within that node. The destinations of messages in these systems are specified using these process identifiers. In contrast, messages in the EMPS system are transmitted and received to/from a port, which contains a local reference to a mailbox (Sec. 3.1.6). Mailboxes exclusively manage the location dependent information of processes, viz. the process addresses (Sec. 3.1.2.1). These process addresses contain the physical location of the process plus a locally unique identifier. When a process moves to a different processor, its process address changes. The communication structure can be updated efficiently (Chapter 5) by changing the process address in those mailboxes that are used for interprocess communication with/by the migrating process.

In order to ensure a rapid response to real-time events [45], the EMPS nucleus schedules processes according to a preemptive priority-based scheduling strategy. When a process is created, it is assigned a priority that reflects its relative urgency. When more than one process is ready to run, the EMPS kernel always selects the highest priority process. The preemptive priority scheduling policy is suitable for EPEP (Sec. 1.1.1) and for SRT executions in DEDOS (Sec. 1.1.2). DEDOS HRT executions are executed deterministically at times specified by an off-line calculated schedule.

3.1.2.1 Process address

Processes in the EMPS nucleus are identified by a 32-bit unique process address (Fig. 3.7). Process addresses are described by the class PROCESS_ADDRESS (Fig. 3.8).

A process address consists of a 16-bit processor address and a 16-bit locally unique identifier. The processor address (described by the class PROCESSOR_ADDRESS) consists of an 8-bit node number (NN), a 2-bit cluster number (CN), and a 3-bit processor number (PN). The processor address contains the physical location of a process (Chapter 2). The locally
unique identifier is an index in the process table. The process table provides the mapping from process address to process. The use of explicit fields for node number, cluster number, and processor number allows for the distributed generation of globally unique process addresses, and an efficient mapping from process address to system address. When a process migrates to a different processor, its process address changes (Chapter 5).

3.1.2.2 Process states

All processes in the EMPS system are memory resident. At any moment, a process is in one of three states, viz. READY, CURRENT, or BLOCKED (Fig. 3.9). A process eligible for execution is in the READY state. A process that is not ready is said to be BLOCKED. Of all highest priority ready processes, the process that is ready for the longest time is allocated the
3.1 The EMPS Nucleus

Figure 3.9: Three different process states can be distinguished in the EMPS nucleus, viz. CURRENT, READY, and BLOCKED. The BLOCKED state can be subdivided into WAIT_FOR_SEMAPHORE, RECEIVING, BLOCKED_ON_SEND, and DELAY.

CPU, and is referred to as CURRENT.

3.1.2.3 Process switching

The cost of finding the next process to activate is determined by the CPU allocation rule and its software implementation. In this section, low-level process allocation is described. On each process switch, the process switching mechanism must locate the highest priority ready process, that has been ready for the longest time. To find this process efficiently, a data structure called ready queue is maintained by the nucleus. For each priority level, ranging from priority 0 (lowest) to priority 7 (highest), a singly-linked queue of ready processes is maintained (Fig. 3.10). Each priority level corresponds to the hardware priority level of the processor on the computer module (Sec. 2.2.1) it is part of. Lower-level hardware interrupts are thus disabled. For a more detailed description we refer to Sec. 3.1.5.

Processes are always added to the end of the ready queue. A tail pointer is maintained for each queue so this add operation requires a
Figure 3.10: A singly-linked queue of ready processes is maintained for each priority level.

small, constant number of instructions. The process in the queue that is ready for the longest time, is pointed to by the head pointer. A list of the head pointers is maintained in order of decreasing priority. The highest priority ready process is found by searching the list of head pointers until a non-nil pointer is found. This process is then activated (dispatch). It is, however, not removed from the ready queue, only its state is changed from READY to CURRENT. Since only the current process (the first process in the queue) can be suspended, the remove operation from the queue always requires a constant number of instructions.

The efficiency of process switching is improved by examining only two situations that actually result in a process switch. The first situation is that the current process blocks. Since there can be no processes of higher priority than the current process, only the ready queues of the same or lower priorities need to be searched. The second situation that results in a process switch is that a higher priority process becomes ready. This can be the result of either a nucleus operation of the current process or interrupt service routine. In these two cases, the priority of the ready process is compared with the priority of the current process. If the ready
3.1 The EMPS Nucleus

process is of higher priority, it must be the highest priority ready process now (because the current process was previously of highest priority), and can be activated immediately without searching the ready queues. It is obvious that no process switch occurs if the readied process is of lower or the same priority.

Ready queues are never empty because on each processor a system process of lowest priority, called the null process, is always ready. This process is used to measure the idle time of a processor.

![Diagram of SCHEDULER class]

Figure 3.11: The class SCHEDULER maintains the ready queue containing processes that are ready-to-run. SCHEDULER also provides basic process switching routines, viz. Block, AddReady, and BlockAndAddReady.

Process switching is managed by the global object scheduler (Fig. 3.11). The scheduler manages the ready queue and provides three basic process switching operations, viz.

- Block()
- AddReady( process )
• BlockAndAddReady( process )

The Block0 operation blocks (suspends) the current process (i.e. removes it from the ready queue), and activates the highest priority ready process. The AddReady operation changes the state of process to ready (i.e. adds it to the ready queue), and activates this process if it is of higher priority than the current process. The BlockAndAddReady operation blocks the current process, changes the state of process to ready, and activates a ready process according to the process allocation rule. The BlockAndAddReady operation is more efficient than two separate operations, and is invoked frequently for interprocess communication ("blocking send"). Examples of the use of the process switching routines provided by the scheduler are given in the Appendix. The class CPU (Fig. 3.11) provides the abstraction of the processor hardware. For the MC68030 processor (Chapter 2), the class CPU includes e.g. address and data registers, memory management registers, and floating-point registers of the MC68882 coprocessor.

3.1.2.4 Tasks and processes

A task in the EMPS system consists of a set of class definitions and a set of process definitions. A task can have multiple processes that share the same address space for code, data, and heap (task address space). Each process within a task has separate address spaces for user and supervisor stack (process address space). See also Fig. 3.13. EMPS processes are sometimes referred to as threads in other (distributed) systems [20].

When all processes within a task would share the same address space for code, data, heap, and stacks, only task switching (not process switching) would require the switching of address spaces. The EMPS system, however, supports memory management hardware (Chapter 2) that can be used to switch address spaces efficiently, viz. by updating a single pointer to the translation tree structure. The update of that pointer invalidates the address translation cache (ATC) of the memory management unit (MMU). In the EMPS system, the times to switch tasks and processes are thus of the same order of magnitude. In order to provide protection (in hardware) between different processes within the same task, each process has a different translation tree structure, used by the MMU, which is described by the class ADDRESS.SPACE (Fig. 3.14). The translation tree structures of all processes within one task are initialized in such a way that the physical pages for code, data, and heap are shared.

A process definition is described by the class PROCESS.INFO, that contains the following information:
3.1 The EMPS Nucleus

- A reference to the code and the static data segments of the class definitions of the task (Task address space).
- The initial program counter.
- The relative urgency (priority).
- Stack sizes for user and supervisor stack.

The task loader, which is a process that runs on top of the kernel, allocates memory for the task address space and copies the code and static data segments of the class definitions of the task into this address space. Next, the task loader creates the processes using the process definitions of the task.

Both system and application processes in the EMPS nucleus are managed by the class PROCESS (Fig. 3.12). An instance of the class PROCESS corresponds to what is usually called a process control block (PCB).

The Create routine creates a new instance of the class PROCESS. The local identifier of the process address pad is determined by the first free index in the process table of the processor. The Create routine uses the class PROCESS_INFO for interface. The process is created in the BLOCKED state. The state of the process can be changed by the scheduler (Fig. 3.11).

The kill routine removes the process from the system, i.e. it removes the process from the process table and releases all resources held by the process (e.g. memory).

The class PROCESS has the attribute address_space (Fig. 3.14), which contains the virtual address translation map of the process. The routine extend.heap can be used for dynamic memory allocation of the heap. The attribute msg and the routine write are used for the implementation of IPC, described in Chapter 4. The attribute ports maintains a list of ports for IPC, that is used for the implementation of the process migration facility (Chapter 5).

3.1.2.5 Memory management

In the EMPS nucleus, a linear, virtual address space is associated with each process. This address space consists of three parts, viz. a system address space, a task address space, and a process address space (Fig. 3.13). The system address space is shared between all processes running on the same processor, and includes a nucleus code and data segment, a kernel code and data segment, and a segment for memory-mapped I/O. Nucleus code and data segments can only be accessed in supervisor mode. This has been established by marking the page descriptors of these segments
Figure 3.12: The class PROCESS

supervisor-only. In this way, the MMU hardware protects the nucleus code and data segments from accessing it in user mode. The task address space of a process consists of a code segment, a data segment, and a heap segment, that are shared by all processes of a task. The process address space consists of two stack segments, viz. a user and a supervisor stack
segment. System address space, task address space, and process address space can consist of both private and common memory.

![Diagram of memory segments]

Figure 3.13: The 4 Gbyte virtual address space of a process in the EMPS system consists of three parts, viz. a task address space, a process address space, and a system address space.

The virtual address space of a process is managed by the class ADDRESS_SPACE (Fig. 3.14). The routine extend can be used to add a number of page frames to the heap segment of a process. Only the heap segment of the task address space of a process can be extended. Each segment contains an array of allocated page frames (page_table), which is used by the memory management hardware to translate a virtual to a physical address.

The runtime library provides routines for allocating and releasing heap memory (Unix provides e.g. malloc and free). The implementation of these routines requires a system call that allocates a block of memory to the heap segment of a process. Therefore, the EMPS nucleus interface defines the class MEMORY (Fig. 3.15). The Allocate routine removes (size/pagesize)
pages from the corresponding list of free pages, adds these pages to the heap segment of the current process (scheduler.current) by invoking the extend routine on the heap segment. The Allocate routine returns a pointer to the virtual address of the first page.
3.1.3 Process synchronization

The EMPS nucleus interface defines the class SEMAPHORE (Fig. 3.16) that provides routines for synchronizing processes running concurrently on the same processor, e.g. in order to establish mutual exclusion [50].

A semaphore is a shared data structure, consisting of a non-negative counter and a queue, on which two operations are defined, viz. wait and signal. A process that executes a wait operation on a semaphore is suspended in the semaphore queue if the counter has a zero value. Otherwise, the process continues execution after the counter has been decremented. A signal operation either increments the semaphore counter if the queue is empty, or releases the first process from the semaphore queue.

The Create(init.count) routine initializes the semaphore queue, and sets the semaphore counter to init.count.
3.1.4 Time management

Each computer module in the EMPS system supports a real-time clock that periodically generates interrupts (clock tick = 20 ms). The physical clocks on different computer modules are initialized at system startup, but are not synchronized afterwards. The interrupt service process (Sec. 3.1.5) of the real-time clock device is referred to as the clock process. At each clock interrupt, the clock process increments the clock counter, that represents the system time in milliseconds.

The nucleus interface defines the class TIMER (Fig. 3.17) that provides routines to get and set the system time. In addition, the class TIMER provides the delay routine to suspend the execution of a process for an
amount of time $t$, and the waituntil routine to suspend the execution of a process until a specified time $t$ has been reached.

![Diagram of clock queue, SORTED_LIST, ENTRY, TIME, and PROCESS]

Figure 3.17: The class TIMER provides routines for getting and setting the system time. Also routines are provided to delay the execution of processes for a specified amount of time.

The processes, that invoke the routines delay and waituntil, are suspended in the clock queue, which is a monotonously non-decreasing queue of processes in order of wake-up time, maintained by the clock process. At each clock interrupt, the clock process compares the system time and the wake-up time of the first process in the clock queue. If the system time is later than the wake-up time of that process, all processes, that have a
wake-up time equal to or earlier than the system time, are removed from the clock queue and inserted into the ready queue.

3.1.5 Interrupt handling

One of the main goals of device management in the EMPS kernel (Sec. 3.2.3) is transparency, which means that I/O is independent of the physical location of a device. In order to achieve transparency and to keep the nucleus small, interrupts from devices are handled by interrupt service processes (ISP), which are system processes that run on top of the nucleus. A user process invokes an I/O operation (Fig. 3.31) on a device by sending a request message to the corresponding ISP. The ISP initializes the device, and waits for I/O completion (device interrupt). The Interrupt Service Routine (ISR) of the device simply unblocks the ISP, which transmits the result parameters back to the invoking user process.

Interrupt handling by ISPs has the following advantages:

1. Mutual exclusion: Mutual exclusion between processes that perform I/O on the same device is achieved automatically by interprocess communication.

2. Minimal stack usage by ISRs: Interrupts from devices are handled on the supervisor stack of the currently executing process. When an ISR would handle the interrupts from devices instead of an ISP, sufficient stack memory should be allocated to each process in order to handle nested interrupts from devices. By having ISPs that handle interrupts from devices, stack usage by ISRs has been minimized.

The ISP requires indivisibility of execution relative to interrupts from the device it is handling, to prevent that the state of the device changes while the process is executing. This is achieved by assigning the ISP the processor priority of the interrupting device, thus masking interrupts of the same and lower priorities. This implies that the ISP can be activated and readied as soon as the device interrupts, because no processes of equal or higher priority can be ready when the interrupt request is accepted by the processor.

A potential disadvantage of mapping process priorities onto hardware priority levels is that lower-priority interrupts can get lost while processes of high priority are executing. At the moment a different mechanism is implemented that distinguishes between hardware and software priorities. Multiple (eight) software priorities are introduced that are independent from the hardware priority level. A detailed discussion of this mechanism is beyond the scope of this thesis.
The class INT_HANDLER provides basic routines for interrupt handling.

The class INT_HANDLER (Fig. 3.18) provides the abstraction of the device interrupt hardware. The ISP, executing in non-privileged (user) mode, waits for a device to interrupt by invoking the routine AwaitInterrupt. The private attribute isp contains a reference to the ISP. The ISR, executing in privileged (supervisor) mode, unblocks the process awaiting the interrupt by invoking the routine StartISP. Since the ISP always has higher priority than the current process, the current process is preempted and the ISP is activated immediately.

The class INT_HANDLER can only be used/inherited from within system processes, as is the case for all classes in the nucleus interface (Fig. 3.4). An example of the use of the class INT_HANDLER is described in the Appendix (C.1.4). Access to devices is described in Sec. 3.2.3.
3.1.6 Interprocess communication (IPC)

In the EMPS architecture, processors in different nodes do not share primary memory. Therefore, the semaphore construct combined with common memory cannot be used for communication and synchronization. Instead, some form of message-passing via the computer network is needed. Although the processors within a cluster and a node do share primary memory, the semaphore construct is not used in this case for reasons of uniformity.

In the design of the communication primitives for message-passing, the following decisions must be made [20, 36]:

- **Location-transparent versus location-dependent addressing.** In order to send a message to a server, the client must know the server's address. Process addresses that contain embedded machine numbers do not provide location transparency, i.e. when a process moves to another processor its address changes.

- **Reliable versus unreliable primitives.** A reliable communication service guarantees the delivery of a message despite network errors. An unreliable service does not handle lost messages, and reliable communication is left to higher layers.

- **Blocking versus non-blocking primitives.** With non-blocking primitives, the operating system has separate calls for initiating the communication and waiting for its completion [51]. Blocking primitives suspend the execution of a process until the communication is completed [14, 52]. A RPC service always uses blocking primitives.

- **Buffered versus unbuffered primitives.** With buffered communication, the operating system kernel provides a message queue. A sender transmits messages to this message queue, where they are buffered until requested by the receiver. Unbuffered primitives do not have such a mechanism: When the sender has a message for a receiver that is not yet ready to receive the message, the sender is suspended until the message is requested by the receiver (rendezvous [53, 54]).

In the EMPS system, two communication paradigms are supported, based on the client-server model [10]:

- **RPC service:** A client transmits a request message to a server and waits for a reply message from that server [51].
3.1 The EMPS Nucleus

- **Multicast service**: A client transfers a message to a subset of all possible destinations (At the moment only a unicast service [55] has been implemented).

The protocol for interprocess communication in the EMPS system consists of a hierarchy of three layers, viz. the physical layer, the medium access (MAC) layer, and the service layer (Fig. 3.19). In the sequel the protocols and the design decisions for these three layers will be discussed.

<table>
<thead>
<tr>
<th>1</th>
<th>Physical layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Medium access layer</td>
</tr>
<tr>
<td>3</td>
<td>Service layer</td>
</tr>
</tbody>
</table>

*Figure 3.19: The protocol for interprocess communication consists of a hierarchy of three layers.*

The bottom layer is the physical layer which deals with the hardware aspects of the interconnection network and delivers an unreliable bitpipe service.

The second layer, the medium access layer, provides an unreliable datagram service. The MAC layer provides a non-blocking send primitive and a blocking receive primitive. These primitives are described in detail in Chapter 4. Because the EMPS hardware architecture consists of a hierarchy of communication networks, at this layer different communication protocols are used for intra-processor, intra-/inter-cluster, and inter-node communication.

The service layer provides a reliable, buffered RPC service (blocking primitives), and a reliable, buffered unicast service (non-blocking send and blocking receive primitives). The RPC and unicast service use mailboxes for communication between application processes, similar to [56]. These mailboxes are not used exclusively for buffering but also for location transparent addressing. Several other systems provide also location transparent
communication, e.g. DEMOS/MP [55, 57], V [14, 46], LOCUS [47, 48], and Charlotte [51, 58]. A comparison with other mechanisms for location transparent communication can be found in Sec. 5.1.

In the remainder of this section, we describe the IPC interface of the service layer for RPC communication via mailboxes. A more detailed description of the implementation of RPC communication via mailboxes can be found in Chapter 4. Because the unicast service via mailboxes is a system service, that will be used for the future implementation of multicast communication, it is not described in this thesis.

3.1.6.1 Mailbox communication

A mailbox is a data structure that provides location-transparency for communicating application processes. Mailboxes can be created dynamically by application processes. Mailboxes are created in common memory and can be accessed by all processes residing on that node. Each mailbox has a globally unique name, which serves as the access key. A process possessing the key can be connected to a mailbox via a port. A port is a data structure, located in the address space of a process, that contains a local reference to a mailbox, the mailbox address. Two different types of ports can be distinguished, viz. initiator and responder ports. A communication path between two application processes consists of an initiator port and a responder port, connected to the same mailbox. Two application processes can communicate with each other only if a communication path between them exists. Multiple client and server processes can be connected to a mailbox (Fig. 3.20). Applications that need multiple servers connected to the same mailbox are e.g. (1) process replication for fault tolerance, and (2) a service desk.

Characteristics of mailbox communication include:

- **Location transparency.** In order to achieve location transparency, the information about the location (process addresses) of the communicating application processes is stored in the mailbox. Processes transmit messages to or receive messages from a port, which contains a local reference to a mailbox.

- **Efficiency.** In order to provide efficient mailbox communication within a node, mailboxes and messages are located in the common memory of that node. In this way, processes on different processors within the same node can access the mailbox without passing messages. Also the messages can be read and written directly by processes within the same node.
3.1 The EMPS Nucleus

Client processes

Initiator ports

Datagram/RPC

MAILBOX

Responder ports

Server processes

mailbox

server

Figure 3.20: Multiple client processes (P1, P2, and P3) and server processes (P4, P5, and P6) can be connected to a mailbox via their initiator ports (IP) and responder ports (RP), respectively. The arrows indicate the client/server-relationship. For RPC communication, message transfer is bidirectional.

- Interprocess communication can proceed during migration. The process migration facility provides a mechanism that prevents message loss when a process migrates to another processor, while it is transmitting a message to or receiving a message from a mailbox. All operations on that mailbox are delayed until the migration of the process is completed.

- Efficient reconfiguration. When a process moves to a different processor, only the mailbox connections of the migrating process need to be updated. By updating one mailbox connection, the communication paths of the migrating process to all processes connected to that mailbox are redirected.

- Buffered datagram communication. Mailboxes can be extended with buffers for messages. This implies that a sending process is not suspended when no receiving process is ready to accept the message. The sending process continues its execution, thereby improving par-
allelism.

- **Message ordering for multicast messages.** Because multiple receivers can be connected to a mailbox, a multicast facility can be added easily. The mailbox provides multiple source ordering of messages within a multicast group [59].

3.1.6.2 Mailbox address

A mailbox is identified by a 32-bit *mailbox address* (Fig. 3.21), which is described by the class MAILBOX_ADDRESS (Fig. 3.22). The class MAILBOX_ADDRESS has two attributes, viz. a 8-bit node number *nn* and a 16-bit locally unique identifier *id*. Each node maintains a table of mailboxes, viz. the *mailbox table* that provides the mapping from mailbox address to mailbox. The attribute *id* of class MAILBOX_ADDRESS is an index in the mailbox table of the node, described by the attribute *nn*.

When a mailbox migrates to another node, its mailbox address changes. Because all ports connected to the mailbox contain that mailbox address, in all these ports this address has to be changed, which would result in a significant message-passing overhead. For this reason, the EMPS kernel does not support mailbox migration. Mailbox crashes can be tolerated by replicating mailboxes and using fault tolerant protocols [59]. In the near future, these protocols will be implemented on top of the EMPS system.

![Figure 3.21: The mailbox address.](image)

In the remainder of this section, we describe the classes MAILBOX and PORT, that provide the non-distributed services for IPC of the nucleus interface. The distributed mailbox management services provided by the kernel interface are described in Sec. 3.2.2.

3.1.6.3 Mailbox

Mailboxes in the EMPS nucleus are described by the class MAILBOX (Fig. 3.23). The *Create* routine creates a new instance of the class MAILBOX. The local identifier of the mailbox address *mad* is determined by the first free index in the mailbox table. The *Create* routine uses the class
3.1 The EMPS Nucleus

Figure 3.22: The class MAILBOX_ADDRESS has two attributes, viz. a node number nn and a locally unique identifier id.

KEY for interface. This class describes the access key of the mailbox, which must be globally unique. Because the nucleus provides only non-distributed services for mailbox management, the kernel has to provide additional services that check whether the access key is globally unique.

The Connect routine connects the initiator or responder port of a process to the mailbox by inserting the process address in the list of clients or servers, respectively. The Disconnect routine disconnects the port of a process from the mailbox by removing its process address from the list of clients or servers, respectively.

As described in the introduction of this section, mailboxes are used for both buffering and location-transparent addressing. Therefore, the mailbox data structure defines a queue of class MAILBOX_QUEUE, which not only buffers request messages that are waiting to be processed, but also manages the process addresses of the communicating processes. When a process migrates to a different processor, its process address changes. The process migration facility can redirect the communication path of the migrating process efficiently by changing the old process address into the new process address in the mailbox queue. The class MAILBOX_QUEUE defines the routines put, get, and remove. A detailed description of these routines can be found in Chapter 4. The process migration facility is described in detail in Chapter 5.
3.1.6.4 Port

A port is a data structure that contains a local reference to the mailbox, viz. the mailbox address. The parent class PORT has two descendant classes, viz. L.PORT and R.PORT (Fig. 3.24) that provide nucleus routines for RPC communication. In order to create a new instance of the class L.PORT or R.PORT, the mailbox address must be known. The Create routines of these classes create a new instance of the corresponding class and assign a mailbox address to the mad attribute.

The client process transmits a request message and waits for the re-
Figure 3.24: The class PORT provides routines for RPC message communication

ploy, by invoking the Call service on its initiator port (class L.PORT). The
server process receives the request message by invoking the Get operation
on its responder port (class R.PORT). After performing the requested
service, the server process transmits a reply message back to the client
process by executing a Reply operation on its responder port. A detailed
description of the implementation of the Call, Get, and Reply routines on
initiator and responder ports is presented in Chapter 4. Also the class
MESSAGE BUFFER, that provides the data interface with the application,
is described in detail in Chapter 4.
3.2 The EMPS Kernel

The EMPS nucleus provides the basic mechanisms for transparent distribution: The nucleus provides routines that can be used to create locally a new mailbox with a specific access key. When a process knows the address of a mailbox, it can use the nucleus routines to create an initiator port (an instance of class L.PORT) or a responder port (an instance of class R.PORT) and to perform interprocess communication.

The EMPS kernel provides distributed services for process management, mailbox management, and device management using dedicated mailboxes, viz. system mailboxes. A system mailbox is also an instance of the class MAILBOX (Sec. 3.1.6). The only difference between system mailboxes and other mailboxes is that the mailbox addresses of all system mailboxes are known by all processes in the system. In this way, any process can setup IPC via a system mailbox using the routines provided by the EMPS nucleus (Fig. 3.25): A client process creates an instance of the class L.PORT using the address of the system mailbox at the destination processor. Next, the process sends a message to that mailbox using the call routine on its initiator port. At each processor, a system process, which is connected to that mailbox via an R.PORT, receives the request message from the client process by invoking the get routine. The system process invokes the requested nucleus service and responds by sending a reply message via its responder port using the reply routine. System processes can use the classes defined by the nucleus interface, whereas application processes can only use the classes provided by the kernel interface.

The general mechanism for IPC via system mailboxes is sketched in Fig. 3.25. A client (application) process uses an instance of the template class KERNEL.SERVICE provided by the kernel interface. The class KERNEL.SERVICE defines a number of routines $f_1, f_2, \ldots, f_N$. Each routine of this service has a corresponding message class, viz. MSG.1, MSG.2, ..., MSG.N, which are heirs of the template class SERVICE.MANAGEMENT.MESSAGE. The Create routines of MSG.1, MSG.2, ..., MSG.N create a new instance of the corresponding class (a message object), and pack the parameters for the distributed service into the message by inserting them into the attributes of the object. The execute routines of MSG.1, MSG.2, ..., MSG.N implement the requested nucleus service. These execute routines can only be invoked by the system process at the destination processor. The root class of the system process at the destination processor is described by the template class SYSTEM.PROCESS.ROOT.CLASS. The implementation of this class uses an instance of the class R.PORT to receive message from the system mailbox.

The implementation of the routines $f_1, f_2, \ldots, f_N$ creates an instance
of the class LPORT (initiator port) using the mailbox address of the system mailbox at the destination processor. Next, these routines create a message object of the corresponding class and transmit this message object to the system mailbox by invoking the call routine on its initiator port. The system process receives a message object of the class SERVICE_MANAGEMENT_MESSAGE by invoking the get routine on its responder port. The system process executes the requested nucleus service by invoking the execute routine. The dynamic form of the message object determines which version of the execute routine is applied (dynamic binding). The system process packs the parameters of the reply message into the same message object and transmits the reply message back by invoking the reply routine on its responder port.

The class PROCESS_REFERENCE for process management services (Sec. 3.2.1), the classes MAILBOX_REFERENCE, INITIATOR_PORT, and RESPONDER_PORT for mailbox management services (Sec. 3.2.2), and the class DEVICE_REFERENCE for device management services (Sec. 3.2.3)
<table>
<thead>
<tr>
<th>KERNEL_CLASS</th>
<th>SERVICE_MANAGEMENT_MESSAGE</th>
<th>SYSTEM_PROCESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROCESS_REFERENCE</td>
<td>PROCESS_MANAGEMENT_MESSAGE</td>
<td>process server</td>
</tr>
<tr>
<td>MAILBOX_REFERENCE</td>
<td>MAILBOX_MANAGEMENT_MESSAGE</td>
<td>mailbox server</td>
</tr>
<tr>
<td>INITIATOR_PORT</td>
<td>MAILBOX_MANAGEMENT_MESSAGE</td>
<td>mailbox server</td>
</tr>
<tr>
<td>RESPONDER_PORT</td>
<td>MAILBOX_MANAGEMENT_MESSAGE</td>
<td>mailbox server</td>
</tr>
<tr>
<td>DEVICE_REFERENCE</td>
<td>IOORB</td>
<td>EIP</td>
</tr>
</tbody>
</table>

Figure 3.26: An overview of kernel classes, service management messages, and system processes for process management, mailbox management, and device management.

can be substituted for the template class KERNEL_SERVICE (Fig. 3.26). On each processor, a system mailbox exists for process management services. The system process that receives messages from this system mailbox is referred to as process server. Another system mailbox for mailbox management services exists on each processor. The mailbox server receives messages from that system mailbox. A system mailbox exists for each hardware device. The system processes that receive messages from these system mailboxes are the Interrupt Service Processes (ISP), which are described in detail in Sec. 3.1.5.

The semantics of sending a message object and executing the corresponding routine of a system process is the following: The client process transmits a request message that contains an identifier and the parameters for the requested nucleus service to the system process via the system mailbox. The system process receives the message and invokes the execute routine of the corresponding object using a case statement. The result parameters are packed into the reply message, which is transmitted back to the corresponding client process. The net result is the same as for dynamic binding. The implementation, however, is different since the routines are invoked via the network: the usual pointer mechanism of dynamic binding is replaced by communicating different messages and invoking the routines of the corresponding object via a “receive case of message” statement. We therefore refer to this mechanism as “virtual dynamic binding”.

Local services for process management, mailbox management, and device management can be implemented more efficiently, viz. by directly invoking the corresponding nucleus services without using IPC. In this way, however, process migration and mutual exclusion would be more difficult to support. Therefore, both local and remote services in the EMPS system are implemented using mailboxes to provide transparency.
3.2 The EMPS Kernel

3.2.1 Process management

The goal of process management is to make the location of processes transparent to the user. Because a process can be located on a remote machine, it cannot be referenced directly. Therefore, the EMPS kernel interface defines the class PROCESSREFERENCE (Fig. 3.27) that provides routines that can be used transparently for the creation (new), activation (start), and deletion (kill) of processes.

![Diagram of process management](image)

Figure 3.27: The class PROCESSREFERENCE provides routines for creating, starting, and killing processes

The Create routine creates an instance of the class PROCESSREFERENCE, and assigns a process address to the attribute pad.

The New routine adds a new process to the system at the processor specified by location (of class PROCESS_ADDRESS). The class PROCESSINFO defines the information needed to create a new process (Sec. 3.1.2.4).

The Start routine starts the execution of the process identified by pad.

The Kill routine removes the process, identified by pad, from the system, and releases all resources held by the process (e.g., memory and devices).

At each processor, a system mailbox exists for process management.
The system process that receives messages from this mailbox is referred to as the process server. The process management routines provided by the nucleus can only be invoked by means of IPC to the process server.

### 3.2.2 Mailbox management

The EMPS kernel provides some additional services for mailbox management, viz. the creation and deletion of mailboxes on a remote node, performing a check whether the access key of a new mailbox is globally unique, and static name binding by transparently connecting and disconnecting the initiator and responder ports of application processes to a mailbox with a specific key. At each processor, a system mailbox exists for mailbox management. The system process that receives messages from this mailbox is referred to as the mailbox server.

![Diagram of mailbox management](image)

*Figure 3.28: The class MAILBOX_REFERENCE provides routines for creating, and removing mailboxes from the system*

The EMPS kernel interface defines the classes MAILBOX_REFERENCE, INITIATOR_PORT, and RESPONDER_PORT that can be used by application processes for IPC.
3.2 The EMPS Kernel

The class MAILBOX_REFERENCE (Fig. 3.28) provides routines to add a new mailbox to and to remove a mailbox from the system. The Create routine creates an instance of the class MAILBOX_REFERENCE and assigns an access key to the key attribute of the class. The new routine creates a new mailbox with the access key at the node identified by the class PROCESSOR_ADDRESS. In order to check the uniqueness of the key, the new routine sends a message of class CHECK_KEY_MSG containing the access key to all mailbox management mailboxes. If the key is unique, the mailbox is created at the corresponding node by sending a message of class NEW_MAILBOX_MSG to the system mailbox for mailbox management of that node. Otherwise an exception occurs.

The remove routine removes the mailbox with the access key from the system by sending a message of class REMOVE_MAILBOX_MSG to all mailbox management mailboxes. The mailbox server that finds the key in its mailbox table checks whether all ports are disconnected from the mailbox and removes the mailbox from the mailbox table. If the key is not found, an exception occurs.

The kernel interface also defines the classes INITIATOR_PORT and RESPONDER_PORT (Fig. 3.29) for initiator and responder ports of application processes, respectively, that provide routines to setup and break communication paths.

Client or server processes establish a connection to a mailbox with a specific access key by invoking the Connect routine on their initiator port or responder port, respectively (static name binding). The access key is transmitted to all mailboxes for mailbox management using the class CONNECT_MSG. The mailbox server that finds the access key in its mailbox table connects the process to that mailbox and returns the mailbox address. After the connection has been established, the attribute mad (Fig. 3.24) contains the mailbox address.

A process can break a communication path by disconnecting its port from the mailbox using the Disconnect routine. A message of class DISCONNECT_MSG is transmitted to the mailbox server at the node contained in the mailbox address of the mailbox to which the process was connected. The mailbox server disconnects the process from the corresponding mailbox.

3.2.3 Device management

The objectives in the design of I/O in the EMPS kernel are:

- Device independence: A program should be as much as possible independent of the device type used for its I/O. E.g., it should not
Figure 3.29: The classes INITIATOR_PORT and RESPONDER_PORT provide routines to setup and break a communication path with a mailbox that has a specific access key.

make any difference to a program whether its output is sent to a terminal or to a storage device.

- **Transparency**: I/O of a program should be independent of the physical location of the device.

- **Efficiency**: Since I/O-operations often form a bottleneck in a computing system, it is desirable to perform them as efficiently as possible.

- **Uniform treatment of devices**: For simplicity reasons, all devices are implemented in a uniform manner [60].

Device independence implies that programs do not operate on actual devices but on virtual devices [61]. In order to achieve transparency, devices
are managed by system processes (I/O–processes). Each operation on a device is implemented as IPC to its I/O–process.

Figure 3.30: The class DEVICE.REFERENCE provides routines for transparent device access.

The class DEVICE.REFERENCE (Fig. 3.30) provides distributed routines for transparent device access. The Create routine creates an instance of the class DEVICE.REFERENCE, and assigns a device address to the attribute dad. The routine do_iq can be used to transmit device–dependent messages (DEV.MSG.1, DEV.MSG.2, ..., DEV.MSG.N) to the I/O–process that manages the device, identified by its device address dad. The execute routines of these messages, invoked by the I/O–process of the device, implement the I/O–operations.

A system mailbox exists for each hardware device. The system processes that receive messages from these mailboxes are the interrupt service processes ISP.1...ISP.N (Fig. 3.1). These ISPs provide the abstraction of the device hardware.

I/O is performed in the following way (Fig. 3.31): A user process invokes the I/O operation on a virtual I/O–device by means of IPC to the ISP (Sec. 3.1.5) of the I/O–device. This is done by invoking the appropriate do_iq routine of the device reference class. The I/O–Request Block (class IORB) is packed into the request message, that is transmitted to the ISP via the corresponding device management mailbox (Sec. 3.2). The ISP unpacks the IORB and performs error checks. When the IORB is valid, the ISP initializes the device, and waits for completion by invoking the Await.interrupt operation. When the I/O–operation is finished, the device interrupts. The ISR of the device starts the ISP by invoking the operation
StartISP. The ISP performs error checks, packs the result parameters into the reply message, and sends this reply message back to the user process. Finally, the user process checks the status of the I/O-request.

Device Management

<table>
<thead>
<tr>
<th>User Process</th>
<th>Kernel ISP</th>
<th>Nucleus ISR</th>
</tr>
</thead>
<tbody>
<tr>
<td>pack IORB Call</td>
<td>→ Get unpack IORB</td>
<td>store device registers into DCB</td>
</tr>
<tr>
<td></td>
<td>→ perform error checks</td>
<td>... StartISP</td>
</tr>
<tr>
<td></td>
<td>→ initiate I/O</td>
<td></td>
</tr>
<tr>
<td></td>
<td>→ Await interrupt</td>
<td></td>
</tr>
<tr>
<td></td>
<td>→ unpack reply</td>
<td>pack reply</td>
</tr>
<tr>
<td></td>
<td>← Reply</td>
<td></td>
</tr>
<tr>
<td>unpack reply check status</td>
<td>←</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.31: The EMPS I/O system

I/O-access to local devices can be implemented more efficiently, viz. by directly accessing the device registers without using IPC. In this way, however, process migration and mutual exclusion would be more difficult to support. Therefore, each operation on a device in the EMPS system is implemented using mailboxes to provide transparency.

3.3 The EMPS operating system

In the remainder of this section, the file access facility will be described in detail. A description of the command interface and the loader is beyond the scope of this thesis.

3.3.1 File access

In the EMPS system, nodes are interconnected via the local area network PHYLAN to a central file server (Chapter 2). The file access facility is
3.3 The EMPS operating system

built on top of device management, described in Sec. 3.2.3. File access is performed via messages (remote procedure calls) to the file server, which performs directory and disk management.

The EMPS system provides a set of low-level, UNIX-like services for file access, that are used by the run-time system. In order to provide easy error recovery, a stateless file server has been implemented. It should be noted that multiple processes can access the same file simultaneously without notice. State information of files is managed locally and is transmitted as part of the remote procedure calls to the central file server.

The following services are provided for file access:

- \( \text{fid} = \text{open}(\text{filename}, \text{mode}) \)
- \( \text{fid} = \text{creat}(\text{filename}, \text{prot}) \)
- \( \text{n.read} = \text{read}(\text{fid}, \text{buf}, n) \)
- \( \text{n.written} = \text{write}(\text{fid}, \text{buf}, n) \)
- \( \text{close}(\text{fid}) \)
- \( \text{iseek}(\text{fid}, \text{offset}, \text{end}) \)

The open service opens the file filename in the indicated mode. A file can be opened for reading (R), writing (W), or reading & writing (R/W). The semantics of the mode parameter depends upon the file system at the file server. The open service returns a file identifier fid. The creat service creates a new file filename with protection specified by prot, and opens it for writing. The syntax of prot is Unix-like (see [62]), and its semantics depends upon the file system at the file server. The creat service returns also a file identifier fid. The read service reads n bytes from the file associated with fid into buf, and returns the “number of bytes read” (n.read). The write service writes n bytes from buf into the file fid, and returns the “number of bytes written” (n.written). The close service close the file associated with fid. The Iseek service seeks a new position within the file, identified by fid. If end is 0, the new position is offset bytes from the beginning of the file; if end is 1, the new position is offset bytes from the current position in the file; if end is 2, the new position is offset bytes from the end of the file.

If an error occurs, the services for file access return the value -1, and set the variable errno to the corresponding error number.
3.4 System initialization

The code and data segments of the EMPS operating system, including nucleus, kernel, and utilities, are stored on the storage device of the file server (Chapter 2). After the operating system has been downloaded into the private memory of the computer module, its initialization routine is executed.

This initialization routine

- determines the node number, which is the network address of the local area network controller.
- determines the node configuration, i.e. the number of computer and common memory modules and their physical location within the node (cluster number $CN +$ processor number $PN$).
- initializes the memory subsystem, viz. the MMU mapping registers and their translation tables.
- initializes the system bus interface controller that includes the interrupt controller, and initializes the interrupting devices, e.g. timer, keyboard, terminal, communication registers, and local area network controller.
- initializes the preemptive priority scheduler.
- issues system calls to create processes that are needed to establish the initial execution environment.

The initial execution environment consists of:

- The NULL-process.
- Interrupt service processes for timer, keyboard, terminal, communication registers, and local area network controller.
- The system processes process server, mailbox server, and migration server, that provide distributed services for process management, mailbox management, and migration management, respectively.
- A command interpreter.
- A loader.

The command interpreter provides commands that can be used to load and execute distributed applications in the EMPS system.
4

Implementation of Interprocess Communication (IPC)

In this chapter, we describe the implementation of interprocess communication (IPC) in the EMPS multiprocessor system using the object oriented paradigm. A detailed description of the issues in the design of IPC has been presented in Chapter 3. The protocol for interprocess communication, which is implemented in the nucleus, consists of a hierarchy of three layers, viz. the physical layer, the medium access (MAC) layer, and the service layer. The service layer corresponds to the nucleus interface.

Mailboxes in the EMPS system provide both buffering and location transparent addressing. Multiple client processes can transfer a request message to the mailbox. The mailbox buffers the incoming request messages (First-Come-First-Served), until a server process is ready to accept the message. The mailbox also queues server processes that are ready to accept a request message. In order to establish location-transparency for communicating application processes, the mailbox exclusively manages location-dependent information, viz. the process addresses of the communicating processes. In this way, not only location transparency is achieved but also the communication structure can be updated efficiently when a process migrates to a different processor: By changing the process address of the migrating process in the data structure of the mailbox, the communication paths of all processes, connected to that mailbox are redirected (Chapter 5). In order to support the migration of client processes while waiting for the reply message from a server process, each mailbox maintains a list of
process addresses of client/server-pairs during RPC communication. This list is updated when a process migrates. When the server process transmits the reply message back to the client process, it accesses the mailbox in order to find the (new) process address of the client process that performed the request. A detailed description of the mailbox data structures and their operations is presented in Sec. 4.1.2.3.

In this chapter, we present a detailed description of the implementation of RPC communication via mailboxes. The unicast service via mailboxes is a system service, that is used for the implementation of multicast communication. The classes of the unicast service are similar to the classes of RPC communication, described in this chapter. Also the unicast protocol is similar to the message protocol of RPC communication. A detailed description of unicast and multicast services is beyond the scope of this thesis. Sec. 4.1 describes the service layer interface and the implementation of the service layer protocol. In Sec. 4.2, unreliable datagram communication provided by the MAC-layer is described. An example of the message flow for RPC communication is presented in Sec. 4.3. Some implementation issues are discussed in Sec. 4.4. Finally, in Sec. 4.5, the performance of RPC mailbox communication is evaluated.

4.1 Service layer

The service layer provides a reliable, buffered RPC service, and a reliable, buffered unicast service, using mailboxes. In this section, the service layer protocol for RPC communication via mailboxes is described in detail. The services for creating and removing mailboxes (class MAILBOX_REFERER), and the services for connecting/disconnecting ports to/from a mailbox (classes INITIATOR_PORT and RESPONDER_PORT) are described in Chapter 3.

The nucleus interface (Chapter 3) defines the parent class PORT (Fig. 4.1) that contains a local reference to a mailbox, viz. the mailbox address mad. The descendant classes LPORT and RPORT provide routines that can be used for RPC communication between application processes. Because the mailbox can be located within a different node, the routines provided by the class MAILBOX can not always be invoked directly. Therefore, the routines of the classes LPORT and RPORT use the routines provided by the class SERVICE_MANAGER (Sec. 4.1.1.2) for transparent access to both local and remote mailboxes. The class SERVICE_MANAGER uses the class SERVICE_MESSAGE (Sec. 4.1.1.1) for interface.

The call routine of class LPORT creates a new instance of a service layer message (class CALL_MESSAGE), using the process address cpad of
Figure 4.1: The parent class PORT has two descendant classes, viz. L.PORT and R.PORT, that can be used for RPC communication between application processes.

the invoking process (the client process), the mailbox address mad, and the message buffer data, and invokes the service layer call routine, provided by the class SERVICE_MANAGER. The get routine of class R.PORT creates a new instance of a service layer message (class GET.MESSAGE) using the process address spad of the invoking process (the server process) and the mailbox address mad, and invokes the service layer get routine of class SERVICE_MANAGER. The get routine results the message buffer of a
received service layer message. The reply routine of class R.PORT creates a new instance of a service layer message (class REPLY_MESSAGE) using the process address spad of the invoking process (the server process), the mailbox address mad, and the message buffer data, and invokes the service layer reply routine of class SERVICE.MANAGER.

4.1.1 Service layer interface

In this section, the service layer interface for transparent mailbox access is described. The service layer interface defines three routines (call, get, and reply) and three corresponding types of messages (CALL_MESSAGE, GET_MESSAGE, and REPLY_MESSAGE). In the design of the service layer, we require three different types of messages because the execute routines of these messages implement the service layer protocol for mailbox access. Sec. 4.1.1.1 describes the service layer messages. The interface for transparent mailbox access is presented in Sec. 4.1.1.2.

4.1.1.1 Service data

Service layer messages are described by the class SERVICE_MESSAGE (Fig. 4.2). This class defines the attributes mad, and data, which are used to transfer the mailbox address and the message buffer, respectively.

Three different types of service layer messages can be distinguished, viz. CALL_MESSAGE, GET_MESSAGE, and REPLY_MESSAGE. The class CALL_MESSAGE is used by a client process to transfer a request message to a server process via a mailbox. The class CALL_MESSAGE defines the attribute cpad, which contains the process address of the client process. The class GET_MESSAGE is used by a server process to receive a request message from a mailbox. The class REPLY_MESSAGE is used by a server process to transfer a reply message to the corresponding client process via a mailbox. The classes GET_MESSAGE and REPLY_MESSAGE define the attribute spad, which contains the process address of the server process.

The execute routines of these messages implement the service layer protocol for accessing mailboxes (Sec. 4.1.2.1). These execute routine can only be invoked by a process that exists on a processor within the mailbox node, i.e. the node at which the mailbox, identified by the mad attribute of the message, is located. When the client or server process exist on a processor within the mailbox node (local mailbox access), the execute routine is invoked directly by that process. When the mailbox is located within a different node (remote mailbox access), the message is transferred to that node. The process that receives the message at the mailbox node
Figure 4.2: Service layer messages are described by the class SERVICE_MESSAGE.

executes the service layer protocol by invoking the execute routine of the received message (Sec. 4.2.2.1).

4.1.1.2 Services

The class SERVICE_MANAGER allows transparent mailbox access via three routines, viz. call, get, and reply. The call routine transmits a request message (an instance of class CALL_MESSAGE) to the mailbox, and waits for a reply message to arrive. The blocking call routine uses the class
CALL_MESSAGE for interface. The get routine is used to receive a request message from a mailbox. If no request message is pending in the mailbox, the get routine waits for a request message to arrive. The get routine uses the class GET_MESSAGE for interface and results a message buffer. The reply routine transmits a reply message back to the corresponding client process. The reply routine uses the class REPLY_MESSAGE for interface. The implementation of these routines is described in Sec. 4.1.2.2.

4.1.2 Service layer implementation

Before reading this section, the reader is advised to take a look at Figures 4.5 and 4.6, and the example in Sec. 4.3, for an overview of the entities involved in a communication and the general flow of control. In Sec. 4.1.2.1, the implementation of the execute routines of the service layer messages is described in detail. Sec. 4.1.2.2 describes the implementation of the class SERVICE_MANAGER and the protocol for transparent mailbox access. The implementation of mailboxes is described in Sec. 4.1.2.3.

4.1.2.1 Messages

The execute routines of the service layer messages (Fig. 4.3) implement the protocol for accessing mailboxes. The execute routine of a service layer message can only be invoked by a process that exists within the mailbox node (mad.mm). The id attribute (Sec. 3.1.6) of the mailbox address mad, contained in the service layer message, is used as the index in the mailbox table of the node to obtain a reference to the mailbox. The implementation of the execute routines uses the classes MAC.PROC.MSG (process message) and MAC.MANAGER defined by the MAC layer interface (Sec. 4.2). The implementation of the execute routines will be described below.

The execute routine of the class CALL_MESSAGE performs the spad := put(msg) operation on the mailbox queue. If a server process is waiting for the request message, its process address spad is returned. An instance of a MAC layer process message (class MAC.PROC.MSG) is created using the process address of the server process spad and the service layer message msg, and this message is transmitted to the server process (which is in the blocked state) using the send routine provided by the class MAC.MANAGER.

The execute routine of the class GET_MESSAGE performs the msg := get(spad) operation on the mailbox queue. If the execute routine is invoked by the server process (local mailbox access), the message buffer (contained in the data attribute of the service layer message msg) is copied to the message buffer of class GET_MESSAGE. If the get operation on the mailbox
results a request message and the routine is not invoked by the server process spad but by an ISF (remote mailbox access, see Fig. 4.6), a MAC layer process message is created using the service layer message msg and the process address of the server process spad, and the message is transmitted back to the server process spad using the send routine provided by the class MAC.MANAGER.

The execute routine of the class REPLY.MESSAGE performs the cpad := remove(spad) operation on the mailbox queue, which results the process address of the client process cpad that performed the request to the server.
process `spad`. The reply message is transferred (as a MAC layer process message) to the corresponding client process `cpad` using the `send` routine provided by the class `MAC_MANAGER`.

### 4.1.2.2 Protocol for transparent mailbox access

The class `SERVICE_MANAGER` (Fig. 4.4) provides transparent access to both local and remote mailboxes. The routines of the class `SERVICE_MANAGER` determine whether the mailbox, identified by the `mail` attribute of the service layer message, is located within the same node (local mailbox) or within a different node (remote mailbox), and execute the corresponding protocol.

![Diagram of SERVICE_MANAGER](image)

**Figure 4.4: Implementation of the class SERVICE_MANAGER.**
Local mailbox  The implementation of the routines of class SERVICE.MANAGER for accessing local mailboxes (Fig. 4.5) can invoke the execute routines of the service layer messages directly, because the mailbox is located in the common memory of the node.

![Diagram](image)

- = process suspended
- = process ready-to-run

Figure 4.5: Local mailbox access for two different situations, viz. (1) the client process transmits the request message before the server process is ready to accept that message (upper figure), and (2) the client process transmits the request message while the server process is waiting for a message (lower figure)

The call routine transfers the request message (class CALL.MESSAGE) to the mailbox by invoking the execute routine of the class CALL.MESSAGE and waits for the reply message by invoking the blocking receive
routine provided by the class MAC.MANAGER.

The get routine invokes the execute routine of the class GET.MES-
SAGE. If a message was already waiting, the attribute data contains the
message buffer. Otherwise, the get routine waits for a request message
to arrive by invoking the blocking receive routine provided by the class
MAC.MANAGER.

The reply routine invokes the execute routine of the class REPLY.MES-
SAGE, which transmits the reply message to the client process that per-
formed the request.

Remote mailbox  The routines of the class SERVICE.MANAGER for ac-
cessing remote mailboxes can not operate directly on the mailbox, be-
cause the mailbox is located within a different node. The implementa-
tion of the routines of the class SERVICE.MANAGER uses the classes
MAC.MBX.MSG and MAC.MANAGER defined by the MAC layer in-
terface (Sec. 4.2). An Interrupt Service Process (ISP) is involved at the
mailbox node, which receives messages via the interconnection network
(Sec. 4.2.2.1). The call, get, and reply routines (Fig. 4.6) transmit an in-
stance of a MAC mailbox message (class MAC.MBX.MSG), that contains
the corresponding service layer message, to the node at which the mail-
box is located (determined from the mailbox address mad contained in
the service layer message) by invoking the send routine provided by
the class MAC.MANAGER. The ISP that receives the message of class
MAC.MBX.MSG invokes the execute routine of the service layer message
(Sec. 4.2.2.1). The call routine waits for a reply message to arrive, and the
get routine waits for a request message to arrive by invoking the blocking
receive routine provided by the class MAC.MANAGER.

In Fig. 4.6, the client process, the server process, and the mailbox are
located on three different nodes. The mailbox can also be located on either
the client node or the server node. In these situations, the message flow
can be derived from a combination of Figs. 4.5 and 4.6.

4.1.2.3  Mailboxes

The class MAILBOX (Chapter 3) consists of the access key of the mailbox,
two lists of process addresses of client and server processes that are con-
ected to the mailbox via their initiator and responder ports, respectively,
and the mailbox queue.

The interface of the class MAILBOX.QUEUE (Fig. 4.7) defines routines
that can be used to put, get, and remove a message to/from a mailbox. The
class MAILBOX.QUEUE has three private attributes, viz. requests, which is
Figure 4.6: Remote mailbox access for two different situations, viz. (1) the client process transmits the request message before the server process is ready to accept that message (upper figure), and (2) the client process transmits the request message while the server process is waiting for a message (lower figure).

a queue of request messages that are waiting to be handled, servers, which is a queue of process addresses of server processes that are waiting for a request message to arrive, and busy, which is a table of process addresses of client/server-pairs.

The put routine has the following results: If no server process is waiting for the request message, the put routine inserts the request message into requests; Otherwise, the put routine removes the first process address of
the server process from servers and inserts the pair of process addresses of both client process (read from the message) and server process into busy.

The get routine has the following results: If no request message is waiting to be processed, the process address of the server process is inserted into servers. Otherwise, the get routine removes (and results) the request message from requests, and inserts the pair of process addresses of both client process (read from the message) and server process into busy.

The busy table contains the process addresses of the communicating client and server process during RPC communication. When e.g. the client process migrates while waiting for the reply message, the process migration facility of the EMPS system updates the process address of the client process. The server process finds the process address of the client process that performed the request by invoking the remove operation on the mailbox queue.

Figure 4.7: The class MAILBOX_QUEUE
4.2 MAC layer

The MAC layer provides unreliable datagram communication. The MAC layer provides a non-blocking send primitive and a blocking receive primitive. The send primitive implements two different protocols viz.

1. A protocol to transfer a message directly from one process to another by passing the mailbox; the other process may reside either on the same processor, on a different processor within the same node, or on a processor within a different node.

2. A protocol to transfer a message to a mailbox on a different node, via the Interrupt Service Process (ISP) of the PhyLAN controller (Sec. 4.2.2.1) of that node.

For these two protocols, the MAC layer provides two different types of messages, described by the classes MAC.PROC.MSG and MAC.MBX.MSG, respectively (fig. 4.9). A MAC layer message of class MAC.PROC.MSG can be transmitted to a process that has previously invoked the blocking receive primitive. A MAC layer message of class MAC.MBX.MSG can be transmitted to the ISP of the node at which the mailbox is located (Fig. 4.8), determined from the mad attribute of the service layer message.

4.2.1 MAC layer interface

In this section, the MAC layer interface for unreliable datagram communication is described. Sec. 4.2.1.1 describes the MAC layer messages. In Sec. 4.2.1.2, the interface for datagram communication is presented.

4.2.1.1 Service data

The class MAC_MESSAGE (Fig. 4.9) defines the attribute msg that contains a reference to the service layer message. Two different types of MAC layer messages can be distinguished, viz. process messages and mailbox messages, described by the classes MAC.PROC.MSG and MAC.MBX.MSG, respectively. A process message can be used to transmit the service layer message to the process, identified by the dpad attribute, that is in the blocked state (Chapter 3). A mailbox message can be used to transmit a service layer message to the node on which the mailbox is located (determined from the mailbox address contained in the service layer message).
4.2.1.2 Services

The *mac_manager* of class MAC_MANAGER (Fig. 4.10) provides datagram communication and is shared between all processes that execute on the same processor. The routines provided by the class MAC_MANAGER use the class MAC_MESSAGE (Fig. 4.9) for interface. The class MAC_MANAGER defines the routine *send*, that can be used to transfer a message of class MAC_PROC_MSG to a process that is in the blocked state, and to transfer a message of class MAC_MBX_MSG to the node at which the mailbox is located. The blocking *receive* routine provided by the class MAC_MANAGER results a MAC layer message.

4.2.2 MAC layer implementation

The EMPS system consists of a hierarchy of communication networks (Chapter 2). Therefore, the MAC layer protocol provides three different protocols, viz. protocols for intra-processor (IP), intra-node (IC), and inter-node (IN) communication. These protocols will be subsequently described in detail in Sec. 4.2.2.1.
4.2 MAC layer

Figure 4.9: MAC layer message format

4.2.2.1 Protocols for datagram communication

The send routine, provided by the class MAC_MANAGER, can be used to transfer a message of class MAC_PROC_MSG to the process, identified by the dpad attribute of the message, that exists either on the same processor (IP protocol), on a different processor within the same node (IC protocol), or on a processor within a different node (IN protocol). The send routine can also be used to transfer a message if class MAC_MBX_MSG to the node at which the mailbox, identified by the mad attribute of the service layer message, is located (IN protocol).

Intra–processor (IP) protocol The IP protocol is executed when the process that performs the send routine and the destination process, identified by the dpad attribute, exist on the same processor. The process dpad should be in the blocked state. A reference to an instance of the class PROCESS (Chapter 3) is obtained from the process table using the process address dpad as an index in this table. Next, the service layer message is transferred to the process dpad by invoking the write routine provided by the class PROCESS (Fig. 3.12), which inserts the message in the msg attribute of that class, and the process is added to the ready queue by invoking the
addready routine provided by the scheduler (Chapter 3).

Intra-cluster (IC) protocol  Each computer module in the EMPS architecture has been equipped with dedicated hardware communication registers. A processor attracts the attention of another processor by writing a 32-bit message-pointer into the communication register of that processor. This generates an on-board interrupt at the target processor. Interrupts from devices are handled by interrupt service processes (Chapter 3). The interrupt service process (ISP) from a communication register is referred to as the Communication Register Server Process (CRSP).

The IC protocol is executed when the process that performs the send routine and the destination process, identified by the pad attribute, exist on a different processor within the same node. Messages are stored in common memory and transferred via pointers. The implementation of the send routine writes a pointer to the message (of class MAC.PROC.MSG) into the communication register of the processor at which the destination process exists (determined from the comreg table using the processor address (prad attribute) of the process address dpad as an index).
4.2 MAC layer

The root class executed by the CRSP is the class CRSP_ROOT (Fig. 4.11). When the CRSP starts executing, it creates a new instance cr of the class COMMUNICATIONREGISTER, which is an heir of class INT_HANDLER (Fig. 3.18), and adds cr to the table of communication registers, comreg_table, which is maintained for each node. Next, the CRSP invokes the routine loop, which consists of the following instructions: (1) Wait for a message to arrive in the communication register by invoking the awaitinterrupt routine provided by the class INT_HANDLER. (2) Execute the IP protocol because the CRSP and the process, identified by dpad exist on the same processor.

![Diagram of CRSP_ROOT class](image)

**Figure 4.11: The class CRSP_ROOT**

Inter-node (IN) protocol Nodes are interconnected by means of the local area network PhyLAN. Each node has a PhyLAN controller and an ISP that handles interrupts from the network (PhyLAN.ISP). The IN protocol is executed when the process that performs the send routine and the destination process, identified by the pad attribute of a message of class MAC.PROC.MSG, exist on different nodes. The IN protocol is also executed when the process that performs the send routine and the mailbox, identified by the mad attribute of a message of class MAC.MBX.MSG, exist on different nodes. The implementation of the send routine writes the message via the local area network to the destination node (determined
from the node number of either the process address \textit{dpad} or the mailbox address \textit{mad} contained in the service layer message.

The root class executed by PhyLAN.ISP is the class \textit{PHYLAN\_ROOT} (Fig. 4.12). When PhyLAN.ISP starts executing, it creates a new instance of the class \textit{PHYLAN}, which is an heir of class \textit{INT\_HANDLER} (Fig. 3.18), and adds it to the table of LAN controllers \textit{lan\_table}, which is maintained for the system. Next, PhyLAN.ISP invokes the routine \textit{loop}, which consists of the following instructions: (1) Wait for a message to arrive via the LAN by invoking the \texttt{awaitinterrupt} routine provided by the class \textit{INT\_HANDLER}. (2) If the received message is a message of class \textit{MAC\_PROC\_MSG}, the message is forwarded to the destination process \textit{dpad} using the IP or IC protocols. (3) If the received message is of class \textit{MAC\_MBX\_MSG}, the execute routine of the service layer message (attribute \textit{msg}) is invoked, which implements the routines for (remote) mailbox access (Sec. 4.1.2.2).

![Diagram](Image)

\textbf{Figure 4.12: The class PHYLAN\_ROOT}
4.3 Example of communication

In Fig. 4.13, the message flow is sketched for RPC communication between a client process and a server process via a mailbox. The client process, the server process, and the mailbox are located on different nodes, viz. the client node, the server node, and the mailbox node, respectively.

![Diagram of message flow]

- = process suspended
- = process ready-to-run

Figure 4.13: RPC message flow

The client process invokes the call routine on its initiator port of class L_PORT. The implementation of the call routine creates an instance of a service layer message of class CALL.MESSAGE (using the mailbox address mad, the process address of the client process cpad, and the message to be transmitted), and invokes the call routine provided by the class SERVICE.MANAGER. The implementation of this call routine creates an instance of class MAC_MBX(MSG), packs the service layer message into it, and invokes the send routine provided by the mac.manager, which transfers the message to the mailbox node. At the mailbox node, the message is received by phylan.isp, which invokes the execute routine of the CALL_MESSAGE. The execute routine performs the put operation on the mailbox queue. Because the server process is not yet ready to receive the
message, the message is inserted into the mailbox queue.

The server process invokes the get routine on its responder port of class R.PORT. The implementation of the get routine creates an instance of a message of class GET.MESSAGE (using the process address of the server process spad and the mailbox address mad), and invokes the get routine provided by the class SERVICE.MANAGER. The implementation of this get routine creates an instance of class MAC.MBX.MSG, packs the service layer message of class GET.MESSAGE into it, and invokes the send routine provided by the mac.manager, which transfers the message to the mailbox node. At the mailbox node, the message is received by phyIan.isp, which invokes the execute routine of the GET.MESSAGE. The execute routine performs the get operation on the mailbox queue. The get operation yields the request message from the server process, which is transmitted back to the server process in the following way: The execute routine of class GET.MESSAGE creates an instance of class MAC.PROC.MSG (using the spad attribute of the class GET.MESSAGE), and invokes the send routine provided by the mac.manager, which transfers the message to the server node. At the server node, the message is received by phyIan.isp, which transfers the message to the server process by invoking the write routine, and adds the server process to the ready queue (IP protocol).

After the server has handled the request message from the client process, it transmits a reply message back by performing the reply operation on its responder port. The implementation of the reply routine creates a message of class REPLY.MESSAGE (using the process address of the server process spad, the mailbox address mad, and the reply message), and invokes the reply routine provided by the class SERVICE.MANAGER. The implementation of this reply routine creates an instance of class MAC.MBX.MSG, packs the reply message into it, and invokes the send routine provided by the mac.manager, which transfers the message to the mailbox node. At the mailbox node, the message is received by phyIan.isp, which invokes the execute routine of the REPLY.MESSAGE. The execute routine performs the remove operation on the mailbox queue, which yields the process address of the client process. The reply message is forwarded to the client process as follows: The execute routine creates an instance of class MAC.PROC.MSG, packs the reply message into it, and invokes the send routine provided by the mac.manager, which transfers the message to the client node. At the client node, the message is received by phyIan.isp, which transfers the message to the client process by invoking the write routine, and adds the client process to the ready queue (IP protocol). This concludes the RPC.
4.4 Implementation issues

The service layer protocol for remote mailbox access has been modeled using the object-oriented techniques of inheritance and dynamic binding. Each service layer message has an execute routine which implements the protocol for remote mailbox access. At the mailbox node, the service layer message is received and the execute routine of the corresponding service layer message is invoked. The implementation of this mechanism of "virtual dynamic binding" is discussed in Sec. 3.2.

Communication via the local area network PhyLAN is established by a mastership-passing communication protocol. In order to maintain consistency of this software protocol, interrupts from the PhyLAN controller are handled exclusively by the system controller (Chapter 2). The interrupts from the PhyLAN controller are handled by the PhyLAN Interrupt Service Process. At the system controller of each node, the system process XmitPhyLAN takes care of the transmission of messages via PhyLAN. A message, that is destined for a process within a different node, is forwarded to XmitPhyLAN via a system mailbox (Chapter 3).

Messages are located in common memory of a node, thus they are shared between processes that exist in the same node. For intra-node communication, the pointer to the message buffer is exchanged between the communicating processes. On the receipt of a message via PhyLAN, memory is allocated for that message. A detailed description of allocating and releasing memory is beyond the scope of this thesis.

4.5 Performance evaluation

In this section, performance measurements of RPC communication via mailboxes are presented. After the Initiator port of the client process and the responder port of the server process had been connected to the same mailbox, statistics were collected (Fig. 4.14) by performing a large number of RPC's in a tight loop. The total elapsed time divided by the number of RPC's yields an estimate for the time to perform a remote procedure call.

In table 4.1 timing figures are presented from measurements of RPC communication between a client process and a server process both running on the same processor and on different processors within the same node for two different message lengths. The timing figures for these two different message lengths are equal because only a pointer to the message (located in common memory) is transferred between the client process and the server process.
Client()
{
    int i;
    TIME StartTime,EndTime,Time;
    TIMER t;       // (See Chapter 3)
    MESSAGE_BUFFER msg;
    INITIATOR_PORT ip;
    KEY k;

    ip.Connect(k);  // Connect Port to mailbox
    StartTime = t.get();
    for( i = 0; i < N; i++ ){
        ip.call(msg);
    }
    EndTime = t.get();
    Time = (EndTime - StartTime) / N;
}

Server()
{
    MESSAGE_BUFFER msg;
    RESPONDER_PORT rp;
    KEY k;

    rp.Connect(k);  // Connect Port to mailbox
    while( TRUE ){
        msg = rp.get();
        rp.reply(msg);
    }
}

Figure 4.14: The program used for the performance measurements of RPC communication via mailboxes. N is the number of remote procedure calls. It is assumed that the mailbox with the access key k exists.

In table 4.2 more general timing figures are presented from measurements of RPC communication between a client process and a server process for five different situations including inter-node communication.

A collection of performance measurements of remote RPC in other distributed systems can be found in [63]. The performance measurements of
4.5 Performance evaluation

<table>
<thead>
<tr>
<th>client processor</th>
<th>server processor</th>
<th>0 kbyte</th>
<th>1 kbyte</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. X</td>
<td>X</td>
<td>0.90 ms</td>
<td>0.90 ms</td>
</tr>
<tr>
<td>2. X</td>
<td>Y</td>
<td>1.18 ms</td>
<td>1.18 ms</td>
</tr>
</tbody>
</table>

Table 4.1: Performance measurements for two different situations of intra-node mailbox communication. X and Y represent different processors within the same node.

<table>
<thead>
<tr>
<th>client node</th>
<th>mailbox node</th>
<th>server node</th>
<th>0 kbyte</th>
<th>1 kbyte</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. X</td>
<td>X</td>
<td>X</td>
<td>0.90 ms</td>
<td>0.90 ms</td>
</tr>
<tr>
<td>2. X</td>
<td>Y</td>
<td>Y</td>
<td>10.56 ms</td>
<td>19.06 ms</td>
</tr>
<tr>
<td>3. X</td>
<td>X</td>
<td>Y</td>
<td>13.40 ms</td>
<td>21.72 ms</td>
</tr>
<tr>
<td>4. X</td>
<td>Y</td>
<td>X</td>
<td>22.86 ms</td>
<td>39.22 ms</td>
</tr>
<tr>
<td>5. X</td>
<td>Y</td>
<td>Z</td>
<td>not measured</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Performance measurements for five different situations of inter-node mailbox communication. X, Y, and Z represent different nodes.

a null–RPC in Amoeba, V, and Sprite for a 10 Mbits/s Ethernet interconnection network yield latencies of 1.2 ms, 2.5 ms, and 2.8 ms, respectively. These latencies are much smaller than the latencies for mailbox communication measured on the 2.5 Mbits/s PhyLAN interconnection network (Table 4.2). By means of a detailed analysis of the performance of the communication protocol of PhyLAN, we will show in the remainder of this section that these latencies are mainly determined by the overhead of the mastership–passing protocol of PhyLAN.

As already mentioned in Sec. 2.2.4, allocation of the transmission medium of PhyLAN is based on the concept of cable master. Software determines which station (node) is the cable master. The cable master is the only station that can transmit data messages [41]. All slave stations can receive data messages. Synchronization messages regulate the passing of the mastership from one station to another.

A data message consists of two parts, viz. a fixed–size header and a variable–size data field. The size of the data field is contained in the header. The time to transmit a data message $T_S$ [42] is represented by

$$T_S = T_H + T_D$$

(4.1)

where $T_H$ is the time to transmit the header, and $T_D$ is the time to transmit the data field of a data message. The time to pass the mastership between two stations is represented by $T_M$. 
In [42], a detailed analysis of the performance of the mastership–passing communication protocol of PhyLAN can be found. The performance measurements presented in that paper yield a value $T_M = 2.9$ ms for the time to pass the mastership between nodes, and a value $T_H = 1.35$ ms for the time to transfer the header of a data message. Although these timing figures have not been measured using the EMPS hardware architecture, they can serve as an estimate of the overhead of the mastership–passing communication protocol of PhyLAN.

In the remainder of this section, we will analyze the different situations for inter–node communication in order to determine the overhead of the mastership–passing protocol of PhyLAN. The overhead is calculated using the times $T_M$ and $T_H$ (Table 4.3).

<table>
<thead>
<tr>
<th>client node</th>
<th>mailbox node</th>
<th>server node</th>
<th>Inter–node messages</th>
<th>Mastership transfers</th>
<th>Protocol overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0 ms</td>
</tr>
<tr>
<td>2.</td>
<td>X</td>
<td>Y</td>
<td>Y</td>
<td>2</td>
<td>8.5 ms</td>
</tr>
<tr>
<td>3.</td>
<td>X</td>
<td>Y</td>
<td>Y</td>
<td>3</td>
<td>9.85 ms</td>
</tr>
<tr>
<td>4.</td>
<td>X</td>
<td>Y</td>
<td>X</td>
<td>5</td>
<td>19.35 ms</td>
</tr>
<tr>
<td>5.</td>
<td>X</td>
<td>Y</td>
<td>Z</td>
<td>5</td>
<td>22.25 ms</td>
</tr>
</tbody>
</table>

Table 4.3: The number of inter–node messages and the number of times that the mastership is passed between the nodes for inter–node mailbox communication.

In situation 2 (Fig. 4.15), the server process and the mailbox are located within the same node. After the mastership has been passed to the client node, the client process transfers a MAC layer mailbox message (containing a service layer CALL_MESSAGE) to the mailbox node. The server process receives the request message from the mailbox and performs the requested service. Mastership is passed to the server node, and the server process transfers a MAC layer process message (containing a service layer REPLY_MESSAGE) to the node at which the client process is located. The overhead of the protocol consists of $2 \cdot T_M + 2 \cdot T_H = 8.5$ ms.

In situation 3, the client process and the mailbox are located within the same node. After the server node has obtained mastership, the server process transfers a MAC layer mailbox message (containing a service layer message of class GET_MESSAGE with no data) to the mailbox node. Mastership is passed to the client node, and a MAC layer process message containing the request message from the client process is transferred to the server process. After the server process has performed the requested service, mastership is transferred to the server node, and a MAC layer mailbox message (containing the reply message of class REPLY_MESSAGE) is trans-
4.5 Performance evaluation

Figure 4.15: Message flow for situation 2

mitted to the mailbox node. In the following iteration of the measuring loop, the server node already has the mastership of the network such that the server process can directly transfer the corresponding MAC layer mailbox message. Mastership is passed two times and three headers of data messages are transferred. Therefore, the measured response times include $2 \cdot T_M + 3 \cdot T_H = 9.85$ ms.

In situation 4, the client and server process are located within the same node, while the mailbox is located in a remote node. Five inter-node messages are transferred in this situation, which requires the mastership to be passed four times between the nodes. The data fields of both the request and the reply message are transferred two times for each RPC. The message transfers in situation 5 (Fig. 4.13) can be compared to those in situation 4. The only difference is that situation 5 requires the mastership to be passed five times.

Although we have not been able to measure the performance using a
10 Mbits/s Ethernet interconnection network, a comparison of the figures given in Tables 4.2 and 4.3 suggests that the latencies of RPC communication using mailboxes are comparable to the latencies of RPC communication in other distributed systems.
5

Efficient Process Migration

In this chapter, we describe the process migration facility in the EMPS system [64] that will be used for dynamic load-balancing and fault-recovery.

A process migration facility provides for the transfer of a process between processors in a distributed system. It supports mechanisms to suspend the execution of a process at one processor, to transfer the state of the process to another processor, and to resume the execution of that process. The overall performance of a distributed system can be improved by distributing the load as evenly as possible over the set of available resources in order to maximize the parallelism. The system performance can also be improved by clustering processes on the same processor dependent on their communication patterns, thus reducing the inter-processor communication traffic. Communication costs are reduced by moving a process to a processor that has a resource this process is using frequently. Such load-balancing is difficult to achieve with a static assignment of processes to processors. A system may achieve better overall throughput if the system load can be balanced dynamically by redistributing processes during their lifetimes.

The mechanisms for process migration can also be very useful for dynamic reconfiguration and recovery after crash failures. A fault-tolerant system can e.g. be achieved by checkpointing the process state on stable storage. After a processor crash, the recovery mechanism recreates the process on another processor, restores a consistent process state from stable storage, and continues the execution of the process, thus effectively migrating the process from a processor that crashed to a working one. Note that this backward recovery mechanism only works when the error can be detected and the fault can be traced down to a recoverable component.

Process migration requires that the communication between processes
does not depend on their physical location. The EMPS kernel was designed to support efficient process migration by providing a simple process model (Chapter 3) and by providing location-transparent interprocess communication using *mailboxes* (Chapter 4). As already mentioned in Chapter 3, process migration is only supported for *application processes*, *system processes* (e.g. process server, mailbox server, migration server, and interrupt service processes) cannot migrate.

The organization of this chapter is as follows: In Sec. 5.1, facilities for process migration in several multiprocessor systems are discussed. In Sec. 5.2, the kernel interface and the implementation of the process migration facility in the EMPS multiprocessor system are described. In Sec. 5.3, performance measurements of process migration are presented and evaluated. Finally, in Sec. 5.4, the conclusions of the paper are summarized.

### 5.1 Related work

Several other systems provide a process migration facility, e.g., DEMOS/MP [57, 55], V [14, 46], LOCUS [47, 48], Sprite [49] and Charlotte [51, 58]. The naming services supporting these migration facilities are explained below.

In DEMOS/MP, messages are sent to *links*, which are uni-directional communication paths managed by the kernel. A link contains a globally unique identifier for the recipient of the message. This globally unique identifier consists of an identifier of the node at which the process was created, a unique local identifier for the process within the node, and a last-known location of the process. Messages are always transmitted to the last-known location. When a process moves to another node, the previous node forwards the messages sent to the process and notifies the senders so they can update the link with the new process location.

In the V-kernel, the destinations of messages are specified by using global process identifiers. A global process identifier indicates a *logical host* for the process, which is mapped to the physical node using a cache of mappings maintained by the kernel. If a message to a process fails, because the process is no longer at that location, the kernel sending the message broadcasts to obtain the logical host's new location and updates its cache.

In the LOCUS distributed system, process identifiers are composed of the node on which the process was created (the *origin site*) plus a process identifier within the origin site. The origin site keeps track of the current location of the process. Other machines communicate with the origin site to find out the current location of the process for message forwarding. The Sprite operating system [49] uses a *home node*, similar to the origin site in
LOCUS.

In Charlotte, links are capabilities for duplex communication channels. At each end of the duplex link, information is maintained about the other end. When a process moves to another processor, messages are transferred via all communication links of the process in order to update the information at the stationary link ends.

In order to reduce the time that the execution of the migrating process is suspended, V uses address space pre-copying. Pre-copying is done by an initial copy of the complete address space followed by repeated copies of the pages modified during the copy action until the number of modified pages is small. Then, the process is suspended and the rest of its state is transferred, along with the remaining modified pages. Another mechanism is used in Mach [65]. Mach performs a logical transfer of the address space to the new host, leaving the physical memory of the process in the old host. The virtual address space of the migrated process is copied-on-reference, i.e. an attempt to reference memory pages that have not yet been transferred will result in the generation of requests to copy the desired memory pages from the old host. The copy-on-reference approach makes it difficult to achieve a fault-tolerant system because the physical memory pages of a process can be located in multiple hosts after a number of migrations and hence the crash of one of these hosts will lead to a crash of the process. Pre-copying imposes less limitations with respect to fault-tolerance because only two processors are involved during a limited period. In the EMPS system, the execution of a process is suspended until the process is completely migrated, i.e. its address space is copied to the new machine. This mechanism is also used e.g. in DEMOS/MP, LOCUS, and Sprite. The EMPS system does thus not attempt to reduce the time that a process is suspended during migration (e.g. by using address space pre-copying).

Although location-transparent communication is provided in many distributed systems, it is not always very efficient after the migration of a process to a different node. In DEMOS/MP, the forwarding of messages requires extra overhead, especially after a large number of migrations. Also every machine at which the process was located has to keep a reference to the new process location. This makes it very difficult to achieve a fault-tolerant system. The migration costs in the V-kernel include the time to broadcast the new process location and the time to update the caches. Communication with the origin site in the LOCUS system and the home node in Sprite degrade the overall system performance. In addition, the crash of the origin site or home node will lead to the crash of the migrated process. In Charlotte, every process that has a link connected to the migrating process has to update this link to the new location. In the EMPS
system, mailboxes provide efficient reconfiguration. When a process moves to a different processor, only the mailbox connections of the migrating process need to be updated. By updating one mailbox connection, the communication paths of the migrating process to all processes connected to that mailbox are redirected. In contrast to DEMOS/MP, LOCUS, and Sprite, no extra messages are required after process migration in the EMPS system.

As already discussed in Sec. 3.1.6.2, mailboxes can not migrate. This implies that, after the crash of the memory module in which the mailbox is located, all communication paths via that mailbox are broken. Mailbox crashes can, however, be tolerated by replicating mailboxes and using fault tolerant protocols [59]. Because these protocols are still evolving, a detailed description is beyond the scope of this thesis.

5.2 Process migration

Process migration is defined as the relocation of a process from the processor on which it is executing (the source processor) to another processor (the destination processor) in a distributed system. For failure transparent systems, process migration is an involuntary operation that may be initiated without the knowledge of the migrating process, or any process interacting with it. All processes continue execution with no apparent changes in their computation or communications. The migration facility must reliably and efficiently detach a process from its source environment, transfer it with its context, and attach it to a new environment on the destination machine. The mailbox facility makes the new location of the process transparent to other processes and ensures that no messages are lost. In this way, the transition can be performed without affecting the correctness of the execution of the process.

In this section, the process migration facility in the EMPS system is described. In Sec. 5.2.1, the kernel interface for process migration is presented. Sec. 5.2.2 describes the implementation of the process migration facility. In Sec. 5.2.3, an example of process migration illustrates the use of the routines for process migration.

5.2.1 Interface

The kernel interface of the EMPS system defines the class PROCESS.REFERENCE (Sec. 3.2.1) that provides the routines New, Start, and Kill, that can be used transparently for the creation, activation, and deletion of processes, respectively.
5.2 Process migration

For process migration, the class PROCESS_REFERENCE has been extended with the routine Migrate(newprad) (Fig. 5.1). This routine can be used to migrate the process, identified by the pad attribute, to the processor, identified by the newprad parameter.

![Diagram of process migration](image)

Figure 5.1: The EMPS kernel interface defines the class PROCESS_REFERENCE. The interface of this class has been extended with the routine migrate for process migration.

5.2.2 Implementation

A process migration facility has to provide facilities to transfer the process and its context to a different processor, such that the correctness of the execution of the process is not affected. In addition, facilities are required to establish a reconfiguration of the communication paths (Sec. 3.1.6.1) to all processes interacting with the migrating process such that connectivity is not affected. In the remainder of this section, the implementation of these facilities will be described in detail.

The EMPS kernel provides distributed services for migration management using system mailboxes (Sec. 3.2). On each processor, a migration server exists that receives messages from the system mailbox for migra-
tion management. A surveillance process initiates process migration by invoking the migrate(newpro) routine provided by the class PROCESS.REFERENCE (Fig. 5.1). The implementation of this routine creates an instance of the class L.PORT using the address of the system mailbox for migration management at the processor at which the migrating process exists (determined from the proad attribute of the process address pad). The surveillance process creates an instance of class MIG.MSG using the pad attribute of the class PROCESS.REFERENCE and the parameter newpro, and transfers this message to that mailbox by invoking the call routine on its initiator port. The migration server at the source processor receives the message and invokes the execute routine, which implements the remainder of process migration (Sec. 5.2.3).

5.2.2.1 Process context

In the EMPS system, processes are described by the class PROCESS (Sec. 3.1.2.4). The class PROCESS contains all relevant information required for process migration. This information consists of two parts, viz. (1) a fixed–sized part, which is the same for all processes and contains e.g. the process state, the process priority, the process identifier, the processor state (program counter, register contents, and interrupt masks), a reference to the message buffer, and the mailbox connections of a process, and (2) a variable–sized part, the address space, which contains information of the memory allocated for code, data, heap, and stack segments of the process.

5.2.2.2 Additional mailbox services

The client and server lists in the mailbox data structure (Fig. 5.2) contain the process addresses of all processes that are connected to that mailbox. The mailbox queue contains the process addresses of server processes that are waiting for a request message to arrive. In addition, the mailbox queue contains the process addresses of the client/server pairs of processes in order to be able to route the reply message (Chapter 4). The process address contains the physical location of a process. Because the process address changes when a process migrates to another processor, the process migration facility provides a routine that updates the process address of the migrating process in the mailbox data structure.

Mailboxes are located in the common memory of a node and are thus shared between processes within a node. In order to establish mutual exclusion between processes that want to modify the mailbox data structure, the data structure contains a flag. Each operation on the mailbox is preceded by a lock–operation on the mailbox flag. This lock–operation
uses Test-And-Set instructions to establish exclusive access to the mailbox. After the mailbox operation is completed, an unlock-operation clears the mailbox flag. Because mailboxes are located in common memory, a semaphore (Sec. 3.1.3) can not be used for the implementation of this flag.

In order to support the reconfiguration of the communication structure, the class MAILBOX (Fig. 5.2) has been extended with three routines, viz. lock, unlock, and update. The lock routine is used to acquire exclusive access to the mailbox by setting the flag attribute. The unlock routine releases the lock by clearing the flag. The update( pad,newpad ) routine changes the old process address pad into the the new process address newpad in the client/server lists and the mailbox queue.

Because the mailbox can be located within a different node, the rou-
Figure 5.3: Message for additional mailbox services.

tines provided by the class MAILBOX can not always be invoked directly. As already mentioned in Sec. 4.1.1.2, the class SERVICE.MANAGER (Fig. 4.4) provides transparent access to both local and remote mailboxes. Thus, also the class SERVICE.MANAGER has been extended with three routines, viz. lock, unlock, and update, which use the (service layer) messages of classes LOCK_MSG, UNLOCK_MSG, and UPDATE_MSG (Fig. 5.3) for interface, respectively. The execute routines of these messages implement the protocol for accessing mailboxes (Sec. 4.1.2.1). The routines of class SERVICE.MANAGER determine whether the mailbox, identified by the mad attribute of the message, is located within the same node (local mailbox access) or within a different node (remote mailbox access). For local
5.2 Process migration

mailbox access, the corresponding mailbox routines are invoked directly. For remote mailbox access, an instance of a MAC layer mailbox message is created, and this message is transferred to the mailbox node by invoking the Send routine provided by the class MAC_MANAGER (Sec. 4.2). The process that receives the message at the mailbox node invokes the execute routine of the received message, thus performing the requested mailbox service.

5.2.2.3 Implementation issues

As already mentioned in Sec. 3.1.4, the execution of a process can be suspended on a timer. In that case, the migration facility removes the process from the clock queue of the old processor, and transfers its wake-up time as part of the message to the new processor. If the system time on the new processor is later than the wake-up time of the migrating process, the process is inserted into the ready queue. Otherwise, the process is inserted into the clock queue of the new processor.

The environment information contains the memory blocks of the address space of a process. Therefore, the message that transfers this information (a message of class TRANSFER_MSG) can become very large. At the moment, the process migration facility in the EMPS system only supports the transfer of messages that are smaller than the maximum length of a message that can be transferred via the local area network. A future extension of the migration facility will support larger messages by transferring them in smaller units.

5.2.3 Example of process migration

The migration of a process from the source processor to the destination processor consists of the following consecutive steps (Fig. 5.4).

1. Process migration is started by the initiator by invoking the migrate routine provided by the class PROCESS_REFERENCE. The migrate routine inserts the process address and the destination processor into a message of class MIG_MSG, and transmits this message to the migration server at the source processor via the corresponding system mailbox.

2. The migration server on the source processor receives the message of class MIG_MSG, and invokes its execute routine. The execute routine performs the following operations: (1) The migrating process is halted; (2) if the process was blocked during IPC (waiting for either a
request message or a reply message), the lock routine is invoked on the corresponding mailbox; (3) a message of class TRANSFER_MSG is initialized with the relevant environment information of the process, and this message is transferred to the migration server at the destination processor via the corresponding system mailbox.

3. The migration server on the destination processor invokes the execute routine of the class TRANSFER_MSG, which creates a new instance of the class PROCESS using the information from the received message, allocates memory for the code, data, heap, and stack segments, and copies the contents of the process address space from the message into these segments. The migration server redirects the mailbox connections of the old process to the new process by invoking the update routine for each mailbox connection (from the list of ports of class PROCESS). If the process was blocked during IPC, the lock is released by invoking the unlock routine on the corresponding
mailbox. Finally, the execution of the new process is resumed, and a reply message is transmitted back to the migration server at the source processor.

4. After the migration server at the source processor has received the reply message, it removes the old process from the system, and transmits a reply message back to the process that started the migration. This concludes process migration.

5.3 Performance evaluation

The total cost of process migration in the EMPS system is composed of (1) the cost to transfer the process state (i.e., a process without memory allocated for its process address space), (2) the cost to transfer the process address space, and (3) the cost to update the communication structure.

The time to migrate a process from a source processor to a destination processor in the EMPS system, $T_{MIG}$, is represented by

$$T_{MIG} = T_0 + T_1 \cdot S + T_2 \cdot P$$

(5.1)

where $T_0$ is the time to transfer the process state, $T_1$ is the time to transfer a page of the process address space, $S$ is the process size, i.e., the number of pages (page size is 1 kbyte) of memory allocated for the process address space, $T_2$ is the time to redirect a mailbox connection from one process to another, and $P$ is the number of mailbox connections of the migrating process.

In our experiments, the process address space was located in the private memory of the computer module. Statistics were collected for simple programs (Fig. 5.5) that perform a large number of migrations in a tight loop. The total elapsed time divided by the number of migrations yields an estimate for the time to migrate a process from one processor to another. The program is executed by an initiator process at the source processor. The measurements are obtained using Motorola MC68030 processors running at 20 MHz.

In our first experiment, processes of different sizes $S$, that had no mailbox connections ($P = 0$), were migrated. The processes were migrated within a node, which means that the common memory of the node is used to transfer the data between private memories of the source and destination processor. In Fig. 5.6, the results of these measurements are presented. A
main()
{
    int i;
    TIME StartTime,EndTime,MigrationTime;
    TIMER t;  // (See Chapter 3)
    PROCESS_ADDRESS spad,dpad;
    PROCESS_REFERENCE sp,dp;

    StartTime = t.get;
    sp.Create( spad );
    dp.Create( dpad );
    for( i = 0 ; i < N/2 ; i++ ) {
        sp.migrate( dpad.prad );
        dp.migrate( spad.prad );
    }
    EndTime = t.get;
    MigrationTime = ( EndTime - StartTime ) / N;
}

Figure 5.5: The program used for the measurements of process migration. spad and dpad are the process addresses of the process at the source processor and destination processor, respectively. N is the number of process migrations.

least-squares fit of the experimental data points for process sizes, varying from 8 to 128 kilobytes, yielded

\[
T_0^{[\text{INTRA-NODE}]} = 9.24 \pm 0.05 \text{ ms}
\]

\[
T_1^{[\text{INTRA-NODE}]} = 2.76 \pm 0.01 \text{ ms}
\]

In the second experiment, processes of different sizes \( S \), that had no mailbox connections \( P = 0 \), were migrated between different nodes. The data is transferred via the local area network PhyLAN. In Fig. 5.6, the results of these measurements are presented. A least-squares fit of the experimental data points for process sizes, varying from 8 to 128 kilobytes, yielded

\[
T_0^{[\text{INTER-NODE}]} = 29.94 \pm 0.05 \text{ ms}
\]

\[
T_1^{[\text{INTER-NODE}]} = 6.84 \pm 0.01 \text{ ms}
\]

In the third experiment, the time to migrate a process between processors within the same node was measured, varying the number of mailbox connections. In Fig. 5.7, the results of the measurements are presented. A
5.3 Performance evaluation

Figure 5.6: Performance measurements of intra-node and inter-node process migration for different process sizes (8 - 128 kbyte) and no mailbox connections. The measurements of intra-node and inter-node process migration in the EMPS system are indicated by the full curve and the dashed curve, respectively. The dot-dashed curve indicates process migration in the Charlotte system. Process migration in the Sprite system is indicated by the dotted curve.

The least-squares fit of the experimental data points for 5 different numbers of mailbox connections, yielded

$$T_2^{\text{[INTRA-NODE]}} = 0.49 \pm 0.01 \text{ ms}$$

In the fourth experiment, the time to migrate a process between processors in different nodes was measured, varying the number of mailbox connections (Fig. 5.7). The mailbox is located in the common memory of the source node, i.e., the node in which the source processor is located. Note that in each cycle of the program, the migration server of the source processor invokes the UpdateMailbox service at the source node (intra-node) and the migration server of the destination processor invokes this service at the source node (inter-node) for each mailbox connection. A
Figure 5.7: The measured migration times for a process of constant size (40 kbyte), varying the number of mailbox connections (1 – 15). The measurements for intra-node and inter-node migration are indicated by the dashed curve and the full curve, respectively.

The least-squares fit yielded

\[ T_2^* = \frac{T_2^{[\text{INTRA-NODE}]} + T_2^{[\text{INTER-NODE}]} }{2} = 8.49 \pm 0.01 \text{ ms} \]

Thus

\[ T_2^{[\text{INTER-NODE}]} = 16.49 \pm 0.02 \text{ ms} \]

The total time (in ms) to migrate a process from a source processor to a destination processor in the EMPS system is represented by

\[ T_{\text{MIG}}^{[\text{INTRA-NODE}]} = 9.24 + 2.76 \cdot S + 0.49 \cdot P \]
\[ T_{MIC}^{[\text{INTER-NODE}]} = 29.94 + 6.84 \cdot S + 16.49 \cdot P \]

It is difficult to compare the performance of the process migration facility in the EMPS system with results published for other systems, because each system uses different processors and different process structures. To provide some form of comparison, however, some expressions for migration times in other systems have been presented. The results of performance measurements of the process migration facility in Sprite (Sun–3 workstations) [49], the V–system (Sun–2 workstations, with 68010 processors, connected via a 10 Mbit/s Ethernet LAN) [46], and Charlotte (20 VAX–11/750 machines, connected by a Pronet token ring) [58] are (in ms)

\[ T_V = 80 + 6 \cdot S \]
\[ T_{Sprite} = 190 + 3.63 \cdot S \]
\[ T_{Charlotte} = 54.9 + 6.1 \cdot S + 1.7 \cdot M \]

where \( S \) is the process size in kilobytes, and \( M \) is the number of duplex communication channels. Note that \( M \) in Charlotte is the same as \( P \) in EMPS if a mailbox establishes precisely one communication path, viz. one initiator port and one responder port connected to the mailbox. The total cost of process migration in the V–system has been published [46], but no separate measurements were presented of the time needed to broadcast the new process location, and the time to update the name caches. In the Sprite system, the communication structure is not updated. Instead, all messages are forwarded to the home node, which yields an extra cost of 3 ms per message. In Charlotte, the update of a non–local communication link requires 1.7 ms.

### 5.4 Discussion

A process migration facility has been successfully implemented in the EMPS operating system. The EMPS kernel was designed to support efficient process migration by providing a simple process model and by providing location–transparent interprocess communication using mailboxes. When a process moves to another processor, the reconfiguration of the communication paths can be done very efficiently: by redirecting the communication ports of the migrating process to the mailboxes the process is connected to, the communication paths of all processes to the migrating process are redirected.

As compared to the origin site in LOCUS and the home node in Sprite, mailboxes in the EMPS system are a single point of failure. Also the
forwarding of messages in DEMOS/MP also imposes limitations with respect to fault-tolerance. In the EMPS system, mailbox crashes can be tolerated by replicating mailboxes and using fault-tolerant protocols. V and Mach reduce the time that a process is suspended during process migration using the techniques of pre-copying and copy-on-reference, respectively. The copy-on-reference technique makes it, however, difficult to achieve a fault tolerant system. On the other hand, pre-copying could be a future extension of the process migration facility in the EMPS system.

In spite of the low bandwidth of PhyLAN (2.5 Mbits/s), the performance measurements for inter-node process migration yield results that are comparable to other distributed systems. This has been achieved by using a simple process model. The time to redirect a remote mailbox connection is mainly determined by the protocol overhead of PhyLAN. In the present implementation of the process migration facility in the EMPS system, one message is transmitted via PhyLAN for each remote mailbox connection. A detailed description of the overhead of the communication protocol of PhyLAN is presented in Sec. 4.5. A future implementation of a 10 Mbits/s Ethernet interconnection network will reduce the latencies of process migration because of higher bandwidth and lower protocol overhead. Also the communication paths can be reconfigured more efficiently by exploiting the broadcast facilities of Ethernet. Intra-node process migration uses common memory to transfer a process to another processor and to update the mailbox data structure. Therefore, the performance measurements for intra-node process migration yield significantly better results than process migration in other systems. Also the reconfiguration of the communication paths can be done more efficiently in this case.

As described in Sec. 3.2, the kernel provides distributed services for process management, mailbox management, and device management, using system mailboxes. An application process can invoke these services by means of RPC communication. Application processes can be migrated during these blocking RPCs. The migration facility updates the process address in the corresponding system mailbox such that the system process will transfer the reply message to the new process location (Chapter 4). After migration, requests to perform local kernel services will be directed to the corresponding system mailbox at the new processor.

Although the mechanisms for migrating a process have been implemented, strategies, that actually decide when to move a process, have to be studied. Future research involves the implementation of load balancing algorithms, to achieve improved performance by redistributing the system load across the available resources and by minimizing the inter-processor communication traffic. Backward recovery mechanisms, after a processor or communication link crashed, will also be studied.
Summary

In this thesis, we describe the design of the EMPS multiprocessor executive for distributed computing. As already mentioned in Chapter 1, the development of the multiprocessor executive as a whole consists of several stages, viz. (1) the development of the hardware architecture, (2) the design and implementation of a real-time operating system kernel, (3) the development of a DEpendable Distributed Operating System (DEDOS) and a distributed version of the Eindhoven Program Editor and Processor (EPEP), an application program development environment.

The EMPS hardware architecture, described in Chapter 2, consists of a mixture of tightly and loosely coupled subsystems: Several computer and shared memory modules are interconnected by means of a cluster bus to form a cluster. A node consists of several clusters, interconnected by means of a system bus. Thus, each node is a tightly coupled multiprocessor. Nodes are interconnected by means of a local area network into a loosely coupled structure.

The design of the EMPS multiprocessor kernel is based on the object-oriented paradigm and has been described in Chapter 3. For the description of the EMPS kernel we use a graphical method to represent class structures and their relationships. This description not only presents the kernel interface but also provides useful insight into the internal structure of the kernel. It thus supports higher comprehensibility and maintainability. An important issue in the design of the kernel was support for process migration. Process migration requires that communication between processes is independent of their physical location. In this thesis, we describe a new protocol for IPC that uses mailboxes not only for buffering but also for location-transparent addressing. A small and fast nucleus provides low-level, non-distributed services. In addition, the kernel supports dis-
tributed services via interprocess communication (IPC) with system processes residing on other processors. We model distributed services using the mechanism of "virtual dynamic binding". Each message object contains an execute routine which implements the requested nucleus service. A client process transmits the message to the system process via the corresponding system mailbox. The system process performs the requested nucleus service by invoking the execute routine of the received message object. The dynamic form of the message object determines which version of the execute routine is applied.

Chapter 4 describes the implementation of a special protocol for interprocess communication via mailboxes. This protocol consists of a hierarchy of three layers, viz. the physical layer, the medium access (MAC) layer, and the service layer. The physical layer deals with the hardware aspects of the interconnection network and delivers an unreliable bit-pipe service. The MAC layer provides an unreliable datagram service. The service layer provides (1) a reliable buffered unicast service, and (2) a reliable, buffered RPC service, using mailboxes. Mailboxes are not only used for buffering but also for location transparent addressing, which, in turn, is required for process migration. A detailed analysis of performance measurements of mailbox communication shows that the latencies for mailbox communication are comparable to the latencies of interprocess communication in other distributed systems.

A process migration facility has been successfully implemented. It is useful for dynamic load balancing and for dynamic reconfiguration and recovery after crash failures. The design and implementation of that facility is described in Chapter 5. The process migration facility provides mechanisms to suspend the execution of a process, to transfer the state of the process to another processor, and to resume the execution of that process, without affecting the correctness of the execution. The communication structure can be updated very efficiently when using mailboxes, because only the mailbox connections of the migrating process have to be redirected. This also means that IPC can proceed during process migration and no messages are lost. Performance measurements of process migration demonstrate that, in spite of the low bandwidth of the interconnection network, the timing figures are comparable to those for other distributed systems. A future extension of the process migration facility can be the implementation of a pre-copying mechanism which reduces the time that a process is being suspended during migration.

The development of the hardware architecture was completed in 1990. At the moment, a working prototype of the kernel is realized. We have designed and implemented the low-level nucleus services and distributed kernel services, described in this thesis. On top of the kernel, we have im-
implemented a file access facility, a simple command interface, and a loader. For efficiency reasons, these services have been implemented using the C programming language.

Future work will include the development of the dependable distributed operating system DEDOS and the design and implementation of a distributed version of the program development environment EPEP. Also planned is the implementation of an Ethernet interconnection network, which will enhance the performance of both inter-node interprocess communication and inter-node process migration. Fault-tolerance and availability enhancements will be tackled in the context of the DEDOS project. This also includes the development of fault-tolerant protocols that can tolerate mailbox crashes by replicating mailboxes.
Graphical representation

Classes and their relationships are graphically represented. Fig. A.1 shows the symbols that represent classes, attributes, routines, shared attributes, and deferred routines. The class symbol is not used for predefined classes, e.g. INTEGER. The scopes of shared attributes, viz. processor, node, and system (Chapter 3), are represented by different gray-levels (Fig. A.2). The graphical notation for the class relationships “uses for interface” and “uses for implementation” is shown in Fig. A.3. If a class, that is used for interface, is the result of a function, an arrow is used to denote the relation. A class A is said to be a client of class B, and B a server of class A, whenever A uses class B for interface or implementation. Inheritance relations are represented as solid arrows (Fig. A.4). Genericity is represented by a dashed arrow to the generic class and a solid line to the parameter class (Fig. A.5). The representation of “nested” genericity (the private attribute ready of the class SCHEDULER in Chapter 3) is sketched in Fig. A.6.

**SYMBOLS**

| class | attribute | shared attribute | routine | deferred routine |

Figure A.1: The symbols that represent classes, attributes, routines, shared attributes, and deferred routines

115
Figure A.2: The scopes of shared attributes are represented by different gray–levels

Figure A.3: "Uses for interface" and "uses for implementation" relationships
Figure A.4: The inheritance relation is represented by an arrow between two class definitions.

Figure A.5: Genericity is represented by a dashed arrow to the generic class and a solid line to the actual parameter class.
Figure A.6: Representation of nested genericity. The actual parameter of the generic class C refers to the generic class T with actual parameter S.
B

Used class interfaces

In this section, the class interfaces for tables, queues, arrays, and lists, that are used for the design of the EMPS kernel, are summarized. These class interfaces are inspired by the class definitions of the Eiffel library.

B.1 Tables

The class TABLE is a generic class that is e.g. used for the implementation of the process table and the mailbox table.

class TABLE[T  ANY,U  HASHABLE]
export item, has, put, remove
feature
    item(key:U):T is
      -- Item associated with key

    has(key:U):BOOLEAN is
      -- Is key currently used?

    put(v:T;key:U) is
      -- Insert item v with key

    remove(key:U) is
      -- Remove item associated with key

    entry(i:INTEGER):T is
      -- i-th entry in table

end -- TABLE
### B.2 Queues

The class QUEUE is a generic class that is e.g. used for the implementation of semaphore queues.

```plaintext
class QUEUE[T]
export empty, item, put, remove
feature
  empty: BOOLEAN is
    -- Is queue empty ?

  item:T is
    -- Oldest item added to the queue

  put(v:T) is
    -- Add item v to the queue

  remove is
    -- Remove oldest item
end -- QUEUE
```

### B.3 Arrays

The class ARRAY is a generic class that is used for the implementation of arrays.

```plaintext
class ARRAY[T]
export count, item, empty, put
feature
  count:INTEGER is
    -- Available indices in the array

  empty: BOOLEAN is
    -- Is array empty ?

  item(i:INTEGER):T is
    -- Entry at index i

  put(v:T,i:INTEGER) is
    -- Assign item v to i-th entry
end -- ARRAY
```
B.4 Lists

The class LIST is a generic class that is used for the implementation of lists.

```plaintext
class LIST[T]
  export start,count,item,empty,put,remove
  feature
    start is
    -- Move cursor to first position

    count:INTEGER is
    -- Number of items in the list

    empty:BOOLEAN is
    -- Is list empty?

    item:T is
    -- Item at cursor position

    put(v:T) is
    -- Put item v at cursor position

    remove is
    -- Remove item at cursor position

  end -- LIST
```
Used class interfaces
C

Software architecture

C.1 Implementation of EMPS nucleus classes

C.1.1 Low-level memory management

class FREE_MEMORY
export get, free
feature

free.pageframes:ARRAY[PAGE_FRAME];

Create is
do
  free.pageframes.Create;
end

get:PAGE_FRAME is
do
  -- Remove a pageframe from the array of free pageframes
end

free(page:PAGE.Frame) is
do
  -- Add the pageframe to the array of free pageframes
end
end -- FREE_MEMORY
C.1.2 Process management

class PROCESS
export priority,pad,active,change-State,kill,page-table,msg,assign,ports

priority:INTEGER;
pad:PROCESS.ADDRESS;
state:PROCESS.STATE;
msg:MAC.MESSAGE;
ports:LIST[PORT];

address-space:ADDRESS.SPACE;

Create(info:PROCESS.INFO) is
do
  -- Create a new process with specified priority,
  -- and allocate memory for user and supervisor stack
end

change-State(st:PROCESS.STATE) is
do
  -- Change the state of this process
  -- (Used by the scheduler)
  state=st;
end

kill is
do
  -- Release allocated memory for user address space
  -- Perform state-dependent features, e.g. remove the process from the ready queue
  -- if the process is in the READY state
end

write(msg:MAC.MESSAGE) is
do
  -- Assign message reference to attribute msg of this process
  -- (Used by the IPC routines)
  msg=m;
end

extend.heap(size:INTEGER):POINTER is
do
  -- extend the heap segment of process with size bytes
end

end -- PROCESS
C.1.2.1 Memory management

class MEMORY
export allocate
feature
allocate(size:INTEGER):POINTER is
local
  cp:PROCESS; -- The current process
do
  cp:=scheduler.current;
  Result:=cp.extend.hesp(size);
end
end -- MEMORY

C.1.3 Process synchronization

class SEMAPHORE
export wait,signal
feature
  counter:INTEGER;
  queue:QUEUE[PROCESS];
Create(init_count:INTEGER) is
do
  counter:=counter;
  queue:=queue;
  counter:=init_count;
end

wait is
do
  if counter > 0 then
    counter:=counter-1;
  else
    queue.put(scheduler.current);
    scheduler.block;
  end
signal is
local p:PROCESS;
do if queue.empty then
  counter:=counter+1;
else
  p:=queue.item;
  queue.remove;
  scheduler.addready(p);
end
end -- Semaphore

C.1.4 Time management

class TIMER
export get, set, delay, waituntil
feature
get:TIME is
do
  Result:=system.time;
end

set(t:TIME) is
do
  system.time:=t;
end

delay(t:TIME) is
local qt:ENTRY;
do
  qt.Create(scheduler.current.system.time+t);
  clock.queue.put(qt);
  scheduler.block;
end

waituntil(t:TIME) is
local qt:ENTRY;
do
  qt.Create(scheduler.current.system.time);
  clock.queue.put(qt);
  scheduler.block;
end
end -- TIMER
C.1 Implementation of EMPS nucleus classes

C.1.5 Interrupt handling

class INT_HANDLER
export

..lst-code::

    interrupt
feature

isp:PROCESS; -- Reference to interrupt service process

Create(int_number:INTEGER) is --
do
    int_vector_table.put(self,int_number);
end

interrupt is

..lst-code::

    do
    isp=scheduler.current;
    scheduler.block;
end

startisp is --
do
    if not isp.Void then
        scheduler.attach(isp);
    end
end
end -- -- INT_HANDLER

The file "ClockServer" consists of the following program. The clockserver is a system process that handles interrupts from the clock device.

class CLOCK_DEVICE
export

..lst-code::

    getperiod
inhibit INT_HANDLER
feature

period:TIME;

clock:TIME is -- Return clock period
do
    Result=period;
end
end -- -- CLOCK_DEVICE

class CLOCK_SERVER

clock_device::CLOCK_DEVICE;
timer:TIMER;
Create is
  do
    clockserver;
  end

  clockserver is
    local
      qe: ENTRY;
    do
      clock_device.Create(CLOCK_VEC);
      timer.Create;
    from
      until FALSE
        loop
          clock_device.waitInterrupt;
          system.time := system.time + clock_device.getperiod;
          -- remove those entries from clock queue with t > systemtime
        from
        until clock_queue.item.time > systemtime
          loop
            qe := clock_queue.item;
            clock_queue.remove;
            scheduler.addentry(qe.process);
        end
    end
end -- CLOCK_SERVER

C.1.6 IPC

  class PORT
  feature

    mail:MAILBOX_ADDRESS;

  end -- PORT

  class I.PORT
  export call
  inherit PORT
  feature

    call(m:MESSAGE) is
      -- Send a request message to a mailbox (RPC communication)

  end -- I.PORT
C.2 Implementation of EMPS kernel classes

C.2.1 Process management

class PROCESSREFERENCE
end proc, neo, start, kill
feature

pad:PROCESSADDRESS;
Create(ad:PROCESSADDRESS) is
do
    pad>Create;
pad:=ad;
end

neo(info:PROCESSINFO, pad:PROCESSORADDRESS):PROCESSADDRESS is
local
    np:NEWPROCESSMSG;
    np.PORT;
do
    np>Create(info);
    ip.Create(mad); -- mad is address of PM mailbox at processor pad
    ip.call(np);
    Result:=np.pad;
end
start is
local
  sp:START_PROCESS_MSG;
  ip:1.PORT;
  do
    sp.Create(pad);
    ip.Create(mad); -- mad is address of PM mailbox at processor pad.prd
    ip.call(sp);
  end
  
  kill is
local
  kp:KILL_PROCESS_MSG;
  ip:1.PORT;
  do
    kp.Create(pad);
    ip.Create(mad); -- mad is address of PM mailbox at processor pad.prd
    ip.call(kp);
  end
end -- -- PROCESS_REFERENCE

default class PROCESS_MANAGEMENT_MESSAGE
inherit SERVICE_MESSAGE
export execute
feature

execute is
default
end
end -- -- PROCESS_MANAGEMENT_MESSAGE

class NEW_PROCESS_MSG
inherit PROCESS_MANAGEMENT_MESSAGE
feature
  pinfos:PROCESS_INFO;
  pad:PROCESS_ADDRESS;

Create(infos:PROCESS_INFO) is
do
  pinfo = info;
end
C.2 Implementation of EMPS kernel classes

execute is
local
  p:PROCESS;
do
    p.Create(pinfo);
p.id:=processable.index;
processable.put(p,p.pad);
pad:=p.pad; — — Result
end
end -- NEW_PROCESS.MSG

class START_PROCESS.MSG
inherit PROCESS_MANAGEMENT.MESSAGE
feature
  pad:PROCESS_ADDRESS;
Create(ad:PROCESS_ADDRESS) is
do
  pad:=ad;
end
execute is
local
  p:PROCESS;
do
    p:=processable.item(p.id);
scheduler.addready(p);
end
end -- START_PROCESS.MSG

class KILL_PROCESS.MSG
inherit PROCESS_MANAGEMENT.MESSAGE
feature
  pad:PROCESS_ADDRESS;
Create(ad:PROCESS_ADDRESS) is
do
  pad:=ad;
end
execute is
local
   p:PROCESS;
do
   p:=processable.item(p.id);
   processable.remove(p.id);
   p.kill;
end
end -- KILL_PROCESS_MSG

At system initialization, the system process process server is created that executes the program described by the class PROCESS_SERVER:

class PROCESS_SERVER
feature
Create is
do
   processserver;
end

processserver is
local
   rp:R_PORT;
   pms:PROCESS_MANAGEMENT_MESSAGE;
do
   rp.Create(.,);
   -- Create a responder port using the address of system mailbox for PM
   from
   until FALSE
      loop
         pms:=rp.get;
         pms.execute;
         rp.reply(pms);
      end
   end
end -- PROCESS_SERVER

C.2.2 Mailbox management

class MAILBOX_REFERENCE
export key, new, remove
feature
key:KEY;
C.2 Implementation of EMPS kernel classes

Create(k:KEY) is -- Create a mailbox reference
   do
      key=k;
   end

new(location:NODE_ADDRESS) is
   -- Create a new mailbox identified by key at a specified node
   local
      nn:NEW_MAILBOX_MSG;
      ip:IP_PORT;
   do
      nn.Create(key);
      if not key.exists then
         ip.Create(mad); -- mad is address of MM mailbox at node location
         ip.call(nn);
      end
   end

remove is -- Remove the mailbox identified by key
   local
      rm:REMOVE_MAILBOX_MSG;
      ip:IP_PORT;
   do
      rm.Create(key);
      from nn=0
      until nn=MAX_NODES
      loop
         ip.Create(mad); -- mad is address of MM mailbox at node nn
         ip.call(nn);
      end
   end

key.exists:BOOLEAN is -- Check global uniqueness of the key
   local
      ck:CHECK_KEY_MSG;
      ip:IP_PORT;
      exist:BOOLEAN;
   do
      exist=FALSE;
      ck.Create(key);
      from nn=0
      until exist v nn=MAX_NODES
      loop
         ip.Create(mad); -- mad is address of MM mailbox at node nn
         ip.call(ck);
         exist=ck.exist;
      end
   Result=exist;
end

end -- MAILBOX_REFERENCE
class INITIATOR_PORT
export connect, disconnect
inherit I.PORT
feature
connect(key:KEY) is -- connect a port to the mailbox identified by key
local
    com:CONNECT.MSG;
    ip1:PORT;
do
    com.Create(key);
    from m=0
until not com.mac Void \ m=MAX_NODES
loop
    ip.Create(mad); -- mad is address of MM mailbox at node m
    ip.call(com);
end
    mad:=com.mac;
end -- connect

disconnect is -- disconnect a port from the mailbox
local
    dcom:DISCONNECT.MSG;
    ip1:PORT;
do
    dcom.Create(mad);
    ip.Create(mad); -- mad is address of MM mailbox at node mad.m
    ip.call(dcom);
end -- disconnect
end -- INITIATOR_PORT

class RESPONDER_PORT
export connect, disconnect
inherit R.PORT

-- Identical implementation as routines of class INITIATOR_PORT
end -- RESPONDER_PORT

class MAILBOX_MANAGEMENT_MESSAGE
inherit SERVICE.MESSAGE
export execute
feature
execute is
defered
end

end -- MAILBOX_MANAGEMENT_MESSAGE
C.2 Implementation of EMPS kernel classes

class CHECK_KEY_MSG
inheritate MAILBOX_MANAGEMENT_MESSAGE
feature
key:KEY;
exist:BOOLEAN;

create(k:KEY) is
  do
    key:=k;
  end
execute is
donot exist:=mailboxtable.has(key);
end
end -- CHECK_KEY_MSG

class NEW_MAILBOX_MSG
inheritate MAILBOX_MANAGEMENT_MESSAGE
feature
key:KEY;
create(k:KEY) is
  do
    key:=k;
  end
execute is
local
  mailbox:MAILBOX;
  do
    mailbox.Create(key);
    mailboxtable.put(mailbox.key);
  end
end
end -- NEW_MAILBOX_MSG

class REMOVE_MAILBOX_MSG
inheritate MAILBOX_MANAGEMENT_MESSAGE
feature
key:KEY;
create(k:KEY) is
  do
    key:=k;
  end
execute is
require
mailbox:MAILBOX;
do
mailbox:=mailboxtable.item(key);
if mailbox.empty then -- No ports connected to the mailbox
mailboxtable.remove(key);
end
end -- REMOVE_MAILBOX_MSG

The CONNECT service connects a port to a mailbox with a specific key. The CONNECT service returns the mailbox reference (mailbox node plus a local identifier).

class CONNECT.MSG
inhibit MAILBOX_MANAGEMENT_MESSAGE
feature
key:KEY;
pad:PROCESS_ADDRESS;
map:MAILBOX_ADDRESS;
port:PORT;

Create(k:KEY,p:PROCESS_ADDRESS,pn:PORT) is
do
key:=k;
pad:=p;
port:=pn;
end
execute is
local
m:MAILBOX;
do
m:=mailboxtable.item(key);
m.connect(port,pad);
map:=mailboxtable.index; -- Return parameter
end
end -- CONNECT.MSG

The DISCONNECT service disconnects a port from a mailbox.

class DISCONNECT.MSG
inhibit MAILBOX_MANAGEMENT_MESSAGE
feature
C.2 Implementation of EMPS kernel classes

```plaintext
pad:PROCESS_ADDRESS;
mad:MAILBOX_ADDRESS;
port:PORT;

Create(ad:MAILBOX_ADDRESS;p:PROCESS_ADDRESS;po:PORT) is
do
  mad:=ad;
  pad:=p;
  port:=po;
end

eexecute is
local
  m:MAILBOX;
do
  m:=mailboxtable.entry(mad.id);
  m.disconnect(port,pad);
end
end -- -- DISCONNECT.MSG

The file "MailboxServer" consists of the following program. This program is executed by the system process mailbox server.

class MAILBOX_SERVER
feature
Create is
do
  mailboxserver;
end

mailboxserver is
local
  rp:R_PORT;
  mms:MAILBOX_MANAGEMENT_MESSAGE;
do
  rp.Create(.);
  -- Create a responder port using the address of system mailbox for MM
from
until FALSE
loop
  mms:=rp.get;
  mms.execute;
  rp.reply(mms);
end
end -- -- MAILBOX_SERVER
```
C.2.3 Device management

```java
class DEVICEREFERENCE
end dad,do io
feature

dad:DEVICEADDRESS;

Create(ad:DEVICEADDRESS) is
do
  dad:=ad;
end

do io(iorb:IORB) is
local
  ip:I_PORT;
  do
    ip.Create(mad); -- mad is address of DM mailbox of device dad
    ip.call(iorb);
  end
end -- DEVICEREFERENCE
```
D

Interprocess communication architecture

D.1 Implementation of the service layer

\begin{verbatim}

invariant
    \(cp \in \text{PROCESS} \land cp = \text{scheduler.current}\)
    \(-- cp \text{ is a reference to the current process}\)

class PORT
    feature
        mad:MAILBOX_ADDRESS; \text{-- The mailbox address}\n        end -- PORT
\end{verbatim}
class INITIATOR_PORT
export call
inherit PORT
feature

call(m:MESSAGE.BUFFER) is
   -- Send a request message to a mailbox (RPC communication)
require
   not mad.Void -- The initiator port is connected to a mailbox
local
   msg:CALL_MESSAGE;
do
   msg.Create(cp.pad,mad,m);
   service.manager.call(msg);
end
end -- INITIATOR_PORT

class RESPONDER_PORT
inherit PORT
export get,reply
feature

get(MESSAGE.BUFFER) is
   -- Receive a request message from a mailbox
require
   not mad.Void -- The responder port is connected to a mailbox
local
   msg:GET_MESSAGE;
do
   msg.Create(cp.pad,mad);
   Result:=service.manager.get(msg);
end
end -- RESPONDER_PORT

D.1 Implementation of the service layer

D.1.1 Service data

class SERVICE.MESSAGE
export mad.data
feature

data:MESSAGE.BUFFER;
mad:MAILBOX.ADDRESS;

execute is
require
mam.nn=cp.pad.pmad.nn
(This routine can only be invoked by a process that exists on the mailbox node)
defered
end
end -- -- SERVICE.MESSAGE

class CALL.MESSAGE
inherit SERVICE.MESSAGE
export cpad,execute
feature

cpad:PROCESS.ADDRESS;
Create(cpad:PROCESS.ADDRESS,madr:MAILBOX.ADDRESS;
   dat:MESSAGE.BUFFER) is
do
   cpad:=cpadr;
madr:=madr;
data:=dat;
end
execute is
local
   spadr:PROCESS.ADDRESS;
   proc_msg:MAC.PROC.MSG;
   mailbox:MAILBOX;
do
   mailbox:=mailboxstable.item(mad.id);
   spadr:=mailbox.queue.put(self);
   if not spadr Void then
      -- if server process is waiting for a request message to arrive
   proc_msg:=Create(cpadr.self);
   mac_manager.send(proc_msg);
end
end
end -- -- CALL.MESSAGE
class GET_MESSAGE
inherit SERVICE.MESAGE
export spad.execute
feature
spad:PROCESS.ADDRESS;

Create(spadr:PROCESS.ADDRESS,madr:MAILBOX.ADDRESS) is
do
spadr:=spadr;
madr:=madr;
end
execute is
local
m:SERVICE.MESSAGE;
proc.msg:MAC.PROC.MSG;
mailbox:MAILBOX;
do
mailbox:=mailboxtable.item(m.id);
m:=mailbox.queue.get(spad);
if cp.pad=spadr then -- Server process accesses local mailbox
data:=m.data;
else if not m Void then
proc.msg.Create(spad,m);
mac_manager.send(proc.msg);
end
end
end -- GET_MESSAGE

class REPLY.MESSAGE
inherit SERVICE.MESSAGE
export spad.execute
feature
spad:PROCESS.ADDRESS;

Create(spadr:PROCESS.ADDRESS,madr:MAILBOX.ADDRESS;
dat:MESSAGE BUFFER) is
do
spadr:=spadr;
madr:=madr;
data:=dat;
end
D.1 Implementation of the service layer

execute is
local
cpadr:PROCESS_ADDRESS;
proc.msg:MAC.PROC.MSG;
mailbox:MAILBOX;
do
mailbox:=mailbox table item(mad.id);
cpadr:=mailbox queue remove(cpad);
proc.msg.Create(cpadr.self);
mac.manager.send(proc.msg);
end
end -- REPLY.MESSAGE

D.1.2 Services

class SERVICE.MANAGER
feature

call(m:SERVICE.MESSAGE):SERVICE.MESSAGE is
-- Perform a put operation on a mailbox
require not m.cpad.Void
local
mbx.msg:MAC.MBX.MSG;
do
if m.mad.mmbc.pod.prad.nn then
  m.execute;
else
  mbx.msg.Create(m);
  mac.manager.send(mbx.msg);
end
Result:=mac.manager.receive.msg;
end
get(m:GET_MESSAGE) MESSAGE BUFFER is
   -- Perform a get operation on a mailbox
local
   mbx_msg:MAC.MBX.MSG;
do
   if m.ad.mn=cp.pad.prad.mn then
      m.execute;
      if m.data.Void then -- No request message pending
         Result:=mac.manager.receive.msg.data;
      else
         Result:=m.data;
      end
   else
      mbx.msg.Create(m);
      mac.manager.send(mbx.msg);
      Result:=mac.manager.receive.msg.data;
   end
end

reply(m:SERVICE.MESSAGE) is
   -- Perform a reply operation on a mailbox
local
   mbx_msg:MAC.MBX.MSG;
do
   if m.ad.mn=cp.pad.prad.mn then
      m.execute;
   else
      mbx.msg.Create(m);
      mac.manager.send(mbx.msg);
   end
end -- SERVICE.MANAGER
D.1.3 Mailboxes

class MAILBOX
export key, connect, disconnect, queue
feature

key: KEY;
clients: LIST[PROCESS_ADDRESS];
servers: LIST[PROCESS_ADDRESS];
queue: MAILBOX_QUEUE;

connect(pad: PROCESS_ADDRESS) is
  do
  end

disconnect(pad: PROCESS_ADDRESS) is
  do
  end
end -- -- MAILBOX

class MAILBOX_QUEUE
export put, get, remove
feature

requests: QUEUE[MESSAGE];
servers: QUEUE[PROCESS_ADDRESS];
busy: TABLE[PROCESS_ADDRESS, PROCESS_ADDRESS];

invariant
  requests.empty ∨ servers.empty

put(msg: MESSAGE): PROCESS_ADDRESS is
  require msg, pad ∈ PROCESS_ADDRESS
  do
    if servers.empty then
      requests.put(msg);
    else
      msg, pad = servers.item;
      servers.remove;
      busy.put(msg, pad, msg, pad);
      Result = msg, pad;
    end
  end

Interprocess communication architecture

get(spad:PROCESS_ADDRESS):MESSAGE is
local
  msg:MESSAGE;
do
  if requests.empty then
    servers.put(spad);
  else
    msg := requests.item;
    requests.remove;
    busy.put(msg, cpad, spad);
    Result := msg;
  end
end

remove(spad:PROCESS_ADDRESS):PROCESS_ADDRESS is
require busy.has(spad)
local
  cpad:PROCESS_ADDRESS;
do
  cpad := busy.item(spad);
  busy.remove;
  Result := cpad;
end
end -- MAILBOX, QUEUE

D.2 Implementation of the MAC layer

D.2.1 Service data

class MAC.MESSAGE
export msg
feature

  msg:SERVICE.MESSAGE;
end -- MAC.MESSAGE
D.2 Implementation of the MAC layer

```plaintext
class MAC.PROC.MSG
inherit MAC.MESSAGE
export dpad
feature

dpad: PROCESS ADDRESS;

Create(ad: PROCESS ADDRESS, m: SERVICE MESSAGE) is
do
dpad := ad;
    msg := m;
end

end -- MAC.PROC.MSG

class MAC.MBX.MSG
inherit MAC.MESSAGE
feature

Create(m: SERVICE MESSAGE) is
do
    msg := m;
end

end -- MAC.MBX.MSG

D.2.2 Services

class MAC.MANAGER
export send, receive
feature
```
send(m;MAC.MESSAGE) is
   require
      \( m \in \text{MAC.PROC.MSG} \land dp \in \text{PROCESS} \land \)
      \( dp.pad=mp.dp.dpad \land dp\text{state} \in \text{BLOCKED} \)
      \( \lor (m \in \text{MAC.MBX.MS} \land m.msb.mad.nn \neq cp.ppad.prad.nn) \)
      \( \text{--- The mailbox is located in a different node} \)
   local
      p: \text{PROCESS} ;
      cr: \text{COMMUNICATION.REGISTER} ;
      lan: \text{PHYLAN} ;
   do
      if \( m \in \text{MAC.PROC.MSG} \) then
         if \( m.dpad.prad.cp.prad \) then \( \text{--- IP protocol} \)
            \( p := \text{process.table.item}(m.dpad.id) \);
            \( p.\text{write}(m) ; \)
            \( \text{scheduler.addready}(p) ; \)
         else
            if \( m.dpad.prad.nn = cp.prad.nn \) then \( \text{--- IC protocol} \)
               \( cr := \text{comreg.table.item}(m.dpad.prad) ; \)
               \( cr.\text{write}(m) ; \)
            else \( \text{--- \( (m.dpad.prad.nn \neq cp.prad.nn) \) IN protocol} \)
               \( lan := \text{lan.table.item}(m.dpad.prad.nn) ; \)
               \( lan.\text{write}(m) ; \)
         end
      else \( m \in \text{MAC.MBX.MSG} \)
         \( lan := \text{lan.table.item}(m.msb.mad.nn) ; \)
         \( lan.\text{write}(m) ; \)
      end
   end
receive:MAC.MESSAGE is
   do
      \( \text{scheduler.block} ; \)
      \( \text{Result=} \text{scheduler.current.msg} ; \)
   end
end -- MAC.MANAGER
D.2.2.1 Intra-cluster protocol

class COMMUNICATION_REGISTER
inherit DEVICE
export msg,write
feature

msg:MAC_MESSAGE;

write(m:MAC_MESSAGE) is
  do
    msg:=m;
  end
end -- COMMUNICATION_REGISTER

class CRSP_ROOT
feature

cr:COMMUNICATION_REGISTER;

Create is
  do
    cr.Create(...);
    cmreg.table.add(cr.thisprocessor);
    loop;
  end
loop is
  do
    from until FALSE
      loop
        cr.awaitinterrupt;
        mac_manager.send(cr.msg);
      end
    end
end -- CRSP_ROOT
D.2.2.2 Inter-node protocol

class PHYLAN
inherit DEVICE
export msg.write
feature

msg:MAC.MESSAGE; -- The received message

write(m:MAC.MESSAGE) is
do
  msg:=m;
end
end -- PHYLAN

class PHYLAN_ROOT
feature
phylan:PHYLAN;
Create is
do
  phylan.Create(...);
  lan_table.add(phylan,thisnode);
  loop;
end
loop is
do
  from until loop
    phylan.awaitinterrupt;
    if phylan.msg ∈ MAC.PROC.MSG then
      mac_manager.send(phylan.msg);
    else -- phylan.msg ∈ MAC.MBX.MSG
      phylan.msg msg.execute;
    end
  end
end
end -- PHYLAN_ROOT
Bibliography


Samenvatting

Een tweetal verschillende toepassingen hebben aanleiding gegeven tot de ontwikkeling van de EMPS multiprocessor-omgeving voor gedistribueerde applicaties. Deze toepassingen zijn (1) real-time data acquisitie en controle van fysische experimenten en (2) dependable distributed computing. De fysische meetomgeving bestaat uit de real-time PhyDAS hardware en de interpretatieve programmeeromgeving EPEP (Eindhoven Program Editor and Processor). Het onderzoek naar dependable distributed computing omvat de ontwikkeling van DEDOS (een DEpendable Distributed Operating System) waarin architecturen en technieken zullen worden ontwikkeld waarmee betrouwbaarheid en tijdigheid van applicaties kunnen worden ondersteund. De ontwikkeling van het EMPS systeem kan worden onderverdeeld in een viertal fases. De eerste fase omvat het ontwerp en de realisatie van de hardware. Het grootste deel van dit proefschrift is gewijd aan de tweede fase, die bestaat uit het ontwerp en de implementatie van een multiprocessor operating system kernel. In de derde fase zal deze kernel gebruikt worden als een testbed voor de ontwikkeling van een DEDOS en voor de ontwikkeling van een gedistribueerde versie van de EPEP. In de laatste fase zullen gedistribueerde applicaties ontwikkeld worden boven op DEDOS en EPEP.

Hoofdstuk 2 beschrijft de hardware-architectuur van het EMPS systeem. Deze architectuur bestaat uit een aantal nodes die zijn gekoppeld via een netwerk. Iedere node is een multiprocessor-systeem met gemeenschappelijk geheugen, waarin verschillende modules, bijvoorbeeld computer- en geheugen-modules, door middel van een bus met elkaar zijn verbonden. Met behulp van deze "loose" modules kan voor iedere gewenste applicatie de meest geschikte hardware-configuratie worden samengesteld, zodat het systeem gebruikt kan worden voor een verschijnsel aan applicaties.

De eerste fase in de ontwikkeling van de software-architectuur is het ontwerp en de implementatie van een kernel. De belangrijkste doelstellingen van deze kernel zijn:

1. **Eenvoudig en efficient**: De kernel is eenvoudig en biedt alleen die faciliteiten die nodig zijn om een groot deel van alle operating-system services te implementeren m.b.v. processen die boven op de kernel draaien. De kernel maakt op efficiënte wijze gebruik van de onderliggende hardware.

2. **Ondersteunen van process migratie**: De mechanismen die worden gebruikt voor het migreren van processen zijn bruikbaar voor fault
recovery en dynamic load balancing. Om ervoor te zorgen dat ook na het migreren van een process de communicatie van andere processen met het gemigreerde process onveranderd kan blijven doorgaan, biedt de kernel communicatie-faciliteiten die onafhankelijk zijn van de fysieke locatie van processen.

De beschrijvingen van operating-system kernels in de literatuur beperken zich meestal tot de gebruikersinterface, en geven weinig inzicht in de interne structuur van de kernel. In hoofdstuk 3 van dit proefschrift wordt de EMPS kernel op object-georiënteerde wijze beschreven. Met behulp van een grafische methode wordt zowel de gebruikersinterface als de interne structuur van de verschillende onderdelen van de kernel beschreven. Hierdoor wordt de begrijpelijkheid en onderhoudbaarheid van de kernel vergroot. De onderste laag van de EMPS software-architectuur is de nucleus, die niet-gedistribueerde services biedt voor geheugenbeheer, processen, synchronisatie, tijd, interrupt-afhandeling en communicatie. Bovenop de nucleus biedt de kernel transparante, gedistribueerde services voor processen, mailboxes en devices.

Hoofdstuk 4 beschrijft de implementatie van een nieuw protocol voor Remote Procedure Call (RPC) communicatie via mailboxes. Dit protocol gebruikt mailboxes niet alleen voor het bufferen van berichten, maar ook voor het addresseren van processen. Iedere mailbox heeft een unieke naam, die fungeert als toegangssleutel. Ieder process dat de toegangssleutel in zijn bezit heeft kan worden verbonden met de mailbox via een port, die een referentie naar de mailbox bevat. Er zijn twee verschillende typen ports, namelijk initiator en responser ports. Twee processen kunnen alleen met elkaar communiceren wanneer de een via een initiator port en de ander via een responser port verbonden is met dezelfde mailbox. Omdat processen uitsluitend berichten zenden en ontvangen via een port, is de communicatie onafhankelijk van hun fysieke locatie.

Hoofdstuk 5 beschrijft het ontwerp en de implementatie van de faciliteit voor process migratie in het EMPS systeem, die in de toekomst gebruikt zal worden voor het implementeren van algoritmes voor fault recovery en dynamic load balancing. De structuur van de mailboxes zorgt ervoor dat de communicatiestructuur op efficiënte wijze kan worden aangepast wanneer een process migreert: Ein kan volstaan worden met het herconfigureren van de ports van het migrerende process. Een gedetailleerde evaluatie van de gemeten performance van process migratie toont welk gedeelte verschillende onderdelen bijdragen tot de tijd die nodig is om een process naar een andere processor te migreren.
Dankwoord

Vele mensen hebben de afgelopen jaren een belangrijke bijdrage geleverd aan de totstandkoming van dit proefschrift. Allereerst wil ik een aantal mensen in het bijzonder bedanken voor hun wetenschappelijke bijdrage.

Dank aan Ariën van der Wal, die mij de gelegenheid bood om dit promotie-onderzoek te beginnen. Onder zijn begeleiding heb ik ervaring op kunnen doen in het schrijven en presenteren van publicaties.

Dank aan mijn promotor Dieter Hammer voor het overnemen van mijn promotie-onderzoek dat reeds in een vergevorderd stadium was. Zijn vele suggesties en vereenvoudigingen hebben de begrijpelijkheid van dit proefschrift enorm vergroot.

Dank ook aan mijn tweede promotor, Sape Mullender, voor zijn suggesties en opbouwende kritiek over eerdere versies van dit proefschrift.

Dank aan mijn copromotor Klaas Kopinga voor de vele discussies met betrekking tot de toepasbaarheid van het onderzoek in de experimentele fysica, en voor het nauwkeurig lezen en corrigeren van de tekst van dit proefschrift.

Dank aan mijn dagelijkse begeleider Peter van der Stok voor de vele, inspirerende discussies over de onderwerpen die in dit proefschrift zijn beschreven.

Dank aan Onno van Roosmalen voor zijn suggesties met betrekking tot het object-georiënteerde ontwerp van de kernel.

Dank ook aan Mark van Gils voor zijn waardevolle bijdrage aan het onderzoek met betrekking tot process migratie waaruit een publicatie is voortgekomen.

Daarnaast dank aan mijn collega’s van de faculteit Technische Natuurkunde voor de technische ondersteuning en aan mijn collega’s van de faculteit Wiskunde en Informatica voor het aangename gezelschap.

Ten slotte in het bijzonder dank aan mijn moeder en Anneko voor hun liefde en ondersteuning.
Curriculum vitae

7 januari 1963 Geboren te Nijmegen
Stellingen

behorende bij het proefschrift

“The Design of the EMPS Multiprocessor Executive for Distributed Computing”

van Gert-Jan van Dijk

I

Door middel van een rechtsvaardige strategie voor het doorgeven van het mastership zorgt het deterministische communicatie protocol voor het local area netwerk PhyLAN [1] ervoor dat te allen tijde een real-time responsie gegarandeerd kan worden.


II


III

Het ontwerp van een multiprocessor-architectuur bestaande uit een hiërarchie van bussen biedt een oplossing voor het probleem van verzadiging van de gemeenschappelijke bus in single-bus architecturen.

(Dit proefschrift – Hoofdstuk 2)

IV

De in dit proefschrift gebruikte grafische methode [3] voor het object-georiënteerd beschrijven van interfaces en onderlinge relaties van classes vergroot de begrijpelijkheid en onderhoudbaarheid van software en is daarom uitermate geschikt als aanvulling op de documentatie.

V

De object-georiënteerde methode is uitermate geschikt voor het ontwerpen en model-leren van de statische systeemstructuur, maar geeft geen goed inzicht in de control flow en de correctheid van de gebruikte protocollen. Dit levert in het algemeen weinig problemen op bij het ontwerpen van een operating system [4], maar is des te lastiger bij het ontwerpen van applicaties.


VI

Het concept van "mailboxes" wordt vaak gebruikt voor gebufferde communicatie [5]. Dit proefschrift toont aan dat dit concept op eenvoudige wijze kan worden uitgebreid ter ondersteuning van locatie-onafhankelijke communicatie die nodig is voor process migratie. Dit proefschrift toont tevens aan dat dit concept het mogelijk maakt om de communicatiestructuur op efficiënte wijze te herconfigureren na de migratie van een process.


VII

Door de process migratie faciliteit uit te breiden met het "pre-copying" mechanisme voor de adresruimte van een process kan de tijd, dat een process geblokkeerd is tijdens de migratie, worden gereduceerd zonder afbreuk te doen aan fouten-tolerantie.

(Dit proefschrift – Hoofdstuk 5)

VIII

(Mee)-broken schaadde de gezondheid. Het kan hartkloften en longkanker veroorzaken [6]. Desalniettemin stuit het instellen van een rookverbod in ruimtes waarin zich mensen bevinden vaak op meer problemen dan het instellen van een rookverbod in computer-ruimtes.


IX

De ogenblikkelijk tegengestelde semantiek van de begrippen "synchron" en "asynchron" in de jargon van respectievelijk hardware en software ontwerpers leidt in onderlinge discussies regelmatig tot spraakverwarring.

X

De Technische Universiteit Eindhoven is er trots op dat zij haar studenten zeer veelzijdige en uitgebreide sportmogelijkheden kan bieden [7]. De Technische Universiteit zou echter nog trots kunnen worden door deze sportfaciliteiten ook te bieden aan haar promovendi.