Technical Report, GMD FIRST, 1989

Architecture and Rationale of the GENESIS Family of Distributed Operating Systems[*]

Wolfgang Schröder-Preikschat

GMD FIRST
Hardenbergplatz 2, 1000 Berlin 12, FRG

Michel Gien

Chorus systemes
6, Avenue Gustave-Eiffel, 78180 Montigny-Le-Bretonneux, FRANCE

ABSTRACT

GENESIS is a multicomputer system with distributed control and, thus, requires a distributed operating system for global control of system resources. The paper discusses the organization of the distributed operating system. A rationale for a family of distributed operating system is given and various system configurations (i.e., family members), which meet different hardware as well as user requirements, are introduced and discussed.

1. Introduction

GENESIS is a multicomputer system with distributed control, i.e., its nodes are autonomous, cooperating computers that communicate through message passing [Giloi 1989]. Consequently, each node has its own operating system controlling local resources; whereas global functions of the multi-node system are performed jointly by the collective of operating systems. This necessitates *function replication* as well as *data replication* among the nodes, leading to a global *distributed operating system*.

Inter-node cooperation takes the form of a multitude of client-server relationships among system processes residing on different nodes. For this reason the communication model will be based on *inter-process communication*, rather than inter-node communication. Furthermore, basic inter-process communication will be synchronous and supported by dedicated communication hardware.

1.1. Basic GENESIS System Architecture

There are two major classes of nodes the GENESIS operating system has to deal with, *processing nodes* (PNs) and *i/o nodes* (IONs). A large, configurable, number of *clusters* groups a couple of PNs, thus building the GENESIS *core system*. Each of these nodes is supplied with a high performance vector floating-point unit and primarily used for numerical

[*] This work was supported by the European Commission under the ESPRIT program, grant no. 2447.

processing. In addition to these user-dedicated nodes, IONs are used for attaching peripherals directly or indirectly, via server stations (i.e., hosts), to the GENESIS machine. The actual configuration of the GENESIS machine will consist of up to 32 clusters and up to 32 PNs per cluster, thus giving a total of 1024 PNs. The number of IONs depends on the number of peripherals going to be attached to the core system. Interconnection of nodes/clusters is achieved by a high-speed network system. Figure 1 depicts the global system organization in terms of global hardware building blocks.

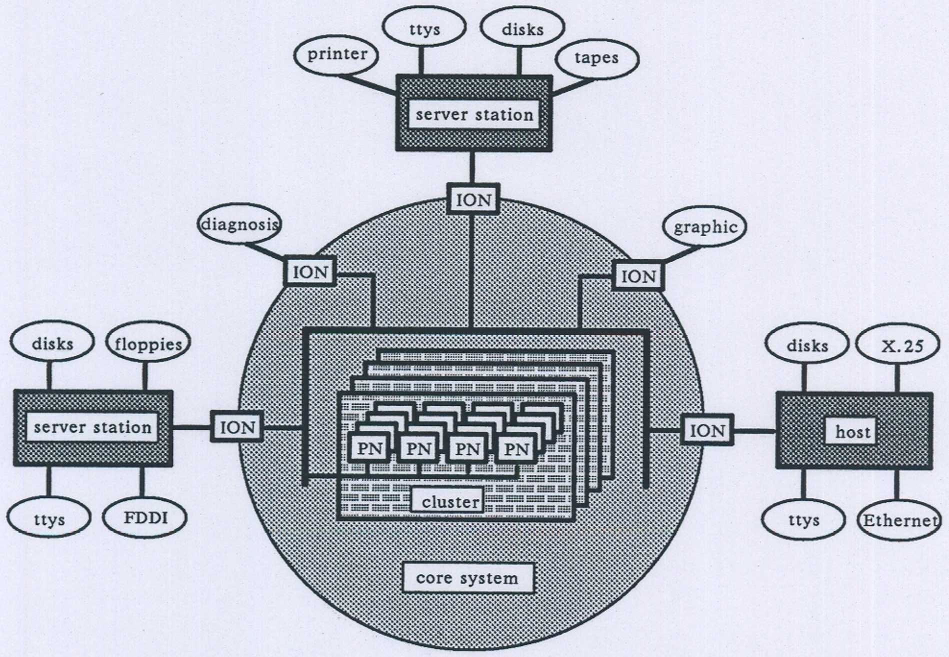

Figure 1: Global Hardware System Organization

Several types of IONs must be managed by the operating system. Generally, different peripherals are controlled by different hardware units, whereby these hardware units are interfaced to the GENESIS core system by a dedicated ION. The basic view is to use an ION to access a *server station*. This server station then provides access to mass storage devices (disks, tapes, etc.), hardcopy devices (printer, plotter, etc.) terminals, workstations as well as local-area and/or long-haul networks. In this sense, a server station represents a traditional *host machine*. There may be several server stations attached to the core system, each one made accessible by a different ION.

Instead of representing a gateway to a server station, an ION may also be equipped with dedicated hardware to directly perform device control. Improving i/o performance will be the major reason to have such an organization. For example, graphic workstations might be directly attached to the core system, thus circumventing the server station in case of graphic i/o. It is sensible to have a similar organization for support of fault-tolerance, i.e., using separate disks, controlled by a dedicated ION, to improve the performance of checkpointing and recovery.

Because the GENESIS machine is based on a *distributed memory* organization, each node has only access to its local on-board memory. Communication between the nodes, i.e., between processes executed by different nodes, is exclusively based on *message-passing*. One of the most important user requirements is to guarantee a message setup timing of maximal 20 µsec [Mierendorff 1989] [1]. The only chance to meet this requirement is to support software-based message-passing activities by dedicated communication hardware units; more specifically, to migrate specific software functions down to the hardware. Therefore, each GENESIS node is equipped with a hardware *communication processor* (CP) that handles low-level inter-process communication functionalities [Giloi, Schroeder 1989]. Nevertheless, there will be a software package that interfaces higher-level application programs to the CP, for the CP is a device rather than being directly part of the node (either PN or ION).

1.2. Rationale for the Family Approach

The motivation to build a family of operating systems primarily grows out of the necessity to manage hundreds or thousands of PNs, a large number of IONs and several server stations, i.e., hosts. Although the nodes are based on the same processor unit, the operating system is faced with the management of a heterogeneous computer system, for the variety of functionally dedicated GENESIS nodes. Considering the hosts, different processing units may be given. Consequently, different local operating systems are required and, most importantly, the PNs must not be overloaded with operating system functions. Without a common design philosophy we will have a set of heterogeneous operating systems and, because having designed artificial boundaries, loss of system performance will be the agenda.

Not only the GENESIS hardware organization calls for a variety of dedicated system services, different (numerical) application programs require different operating system support, too [Thole 1989]. Being based only on a single, monolithic, operating system will have several disadvantages for user programs because, in this case, the nodes are always equipped with a superset of system services. Besides the potential of overall poor performance, one obvious fact is that a large operating system for a single node will require a large memory space, which is to the debit of user programs. The concept of virtual memory generally solves this resource problem, however, because of paging overhead, this solution implies loss of performance for memory-bounded user programs. Typical representatives of these application programs can be found within the numerical programming area, the major application field of the GENESIS machine.

Another important fact is that computer system development in general is an evolutionary course of hardware/software design and implementation. *Flexibility* is the keyword in order to meet future user requirements and to optimally support forthcoming hardware architectures.

[1] The message setup time is determined by the system activities needed to start the transfer of user-level data objects. This includes PN hardware as well as software activities such as performing context switches and process dispatching, handling of interrupts, setting up message descriptors and communication with the CP. It is the time during which the PN is not able to further process the application program.

This especially is of significant importance for the design of operating systems, whose task is, besides other functions, to bring user programs onto the actual hardware. Therefore, an operating system has to be open with respect to both aspects, user requirement and hardware architecture.

1.3. Node Operating System vs. Host Operating System

The MIMD programs are executed on the GENESIS core system. These programs require a distributed and problem-oriented *process execution and communication environment*, ensuring utmost low system overhead on each PN. This environment as well as additional core system services will constitute the GENESIS *node operating system*. In contrast to that, there will be the necessity to have traditional operating system services available, such as file management, global job control, i/o management, etc. Although they can be invoked by the programs being executed on the core system, these services are not provided on the core system itself. Rather, a server station or a host machine is used. It will be the major function of the GENESIS *host operating system* to manage resources of server stations and host machines. The *symbiosis* of both types of operating systems will constitute the GENESIS distributed operating system.

In order to cover a wide application area, the GENESIS distributed operating system will be a symbiosis of PEACE® [Schroeder 1988] and CHORUS® [Rozier et al. 1988]. Common to both systems is that they are *process-structured*, *distributed* and *message-based*. The difference basically is in the distribution of operating system services over a network of tightly coupled nodes, i.e. the actual structure and the granularity of modularization. The work of PEACE highly depends upon *networked system services* provided by a large set of dedicated system processes in an *object-oriented* manner. The major portion of these services may be arbitrarily distributed over the system and globally control system resources. In contrast to that, a CHORUS system is much more monolithic, although it is organized in terms of co-operating processes as in PEACE, and can be classified as an *autonomous system* that trades resources with its peer while trying to keep local control.

Concerning GENESIS, the advantages found in both system models will be combined into a single homogeneous distributed operating system. The fundamental approach is to apply the PEACE model to PNs and the CHORUS model to the server stations as well as hosts. Dependent on the functionality required, both models will be applied to IONs. Thus, PEACE will be the *node operating system* and CHORUS will be the *host operating system* of a GENESIS machine. In the following, we will use PEACE as a synonym for the node operating system and CHORUS as a synonym for the host operating system.

Nevertheless, the boundaries between both types of operating systems are not rigid, i.e., PEACE services may be executed by the host operating system in a similar way as CHORUS services may be executed by the node operating system. This is of significant importance if software originally having been developed for PEACE as well as CHORUS is going to be ported onto GENESIS. Especially in case of PEACE, several software packages are available to control

® PEACE is a registered symbol of GMD.

® CHORUS is a registered trademark, licensed exclusively to Chorus systemes.

a distributed memory MIMD machine, because PEACE is the process execution and communication environment which has been developed for SUPRENUM [Giloi 1988].

2. Program Distribution Support by the Node Operating System

In order to understand the way program distribution will be supported by the node operating system, a short discussion about the basic internal structure of programs has to take place. From the node operating system point of view, a program is organized as a *PEACE entity*. This generally holds for user programs as well as for system programs, both in the following referred to as application program. A distributed application program will be organized as a couple of entities interconnected by some communication system. Figure 2 gives a rough outline of the PEACE entity structure.

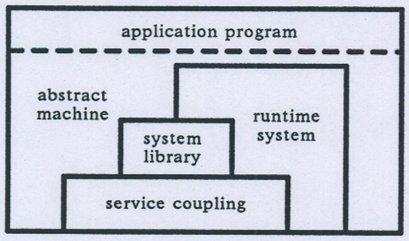

Figure 2: Structure of a PEACE Entity

A single PEACE entity for GENESIS consists of five major building blocks. Dependent on the application point of view, the internal organization of the entity is of different importance. Therefore, in the following a distinction between the top-down user view and the bottom-up system view is made.

2.1. User Point of View

The GENESIS machine is designed to support large scale numerical and distributed MIMD programs. For these programs the MIMD machine is specified in terms of the *GENESIS abstract machine*. This machine specification hides the physical characteristic of the underlying system hardware/software architecture. The idea is that MIMD programs can be written independent of the actual hardware organization/configuration and that these programs will benefit from forthcoming MIMD hardware architectures.

Typically, GENESIS programs are written in terms of primitives provided by this abstract machine. In the first place, these primitives are directly derived from traditional FORTRAN constructs and from fundamental message-passing operations for distributed computing. In addition to that, a dynamical process model is provided and communication between processes is based on typed data objects, such as scalars and vectors, exchanged via messages.

The broken line in the figure indicates that the boundary between abstract machine and real machine (i.e. *runtime system*, *system library* and *service coupling*) is not fixed. The precise position of the boundary depends on the semantics of high-level abstract machine primitives and on the functionality of low-level system services in order to implement the abstract machine. There is always a *semantical gap* between these two levels and the optimal solution for programs being based on the abstract machine definition will be if this gap is only of conceptual nature. That is to say, in order to avoid performance limiting mapping overhead at runtime, a functional specification of the abstract machine always has to reflect technical facts of the underlying system architecture. With respect to forthcoming MIMD hardware architectures, there is a natural *loss of transparency* [Parnas, Siewiorek 1972] because the ideal GENESIS abstract machine will be GENESIS hardware dependent.

2.2. System Point of View

For system programs, the primitives provided by the abstract machine interface are not relevant, at least because system programs surely are not written in FORTRAN. Merely the system program (or compiler) that implements this interface is concerned with abstract machine primitives. The GENESIS system programmer typically is faced with services provided by the three building blocks constituting the real machine.

Proceeding top-down, the *runtime system* is closely related to the programming language or the compiler being used. Typical services are formated i/o and local memory management (e.g., *scanf/printf* and *malloc/free* in C). In terms of programming support for MIMD programs, dedicated mailbox services are provided. This, for example, includes the implementation of abstract machine primitives for inter-process communication.

The *system library* basically provides an interface to the underlying operating system. Part of this interface are functions found in most UNIX™ installations, namely the *UNIX system calls*. These system calls are provided on a compatibility library basis and they are derived from a standardized interface definition such as X/OPEN and POSIX. This UNIX interface is extended by functions which are closely related to the GENESIS distributed operating system. By this way, *lightweight processes* are provided just as *naming* and *checkpointing*, to give some examples. Binary compatibility with UNIX will be achieved by system call emulation.

The building blocks discussed so far all together provide services that can only be efficiently implemented if the corresponding modules share the same address space with the original application program. However, the bulk of operating system services may be located elsewhere. A design that significantly reduces system overhead in terms of memory utilization of a single node makes traditional operating system services available on a *remote procedure call* basis. Moreover, in case of the overall GENESIS hardware architecture it is natural to invoke operating system services in terms of remote procedure calls and, finally, in terms of message-passing. For this purpose a PEACE entity provides for some means of *service coupling* for message-based system architectures. The primary task is to achieve topology transparency, with respect to the actual system configuration, and network transparency, with respect to communication.

™ UNIX is a registered trademark of A T & T Bell Laboratories.

2.3. Lightweight Processes, Teams and League

The PEACE process model is based on *teams* and *lightweight processes* [Cheriton 1979]. A team defines a common framework for a couple of lightweight processes mapped into the same address space, which then can be physically isolated from other team address spaces by a memory management unit. In this sense, a team is the unit of isolation as well as distribution and not a single process. The basic idea of lightweight processes is to encapsulate and uniquely represent separate threads of control within a single address space. A PEACE entity is represented by a single team and, thus, may be controlled by a couple of lightweight processes.

In order to execute a team, i.e. the corresponding PEACE entity, at least one process is needed at the application program level. If necessary, a PEACE entity additionally uses lightweight processes at the runtime system level and/or at the system library level. Typical examples are to provide support for system-wide (i.e. network-wide) exception handling, checkpointing and asynchronous communication [Giloi, Schroeder 1989]. Having a couple of lightweight processes at the user programming level depends on the model of the GENESIS abstract machine. Extending the machine model accordingly will have advantages for several user application.

As was outlined above, a distributed application program will be organized as a couple of PEACE entities. Because each entity is isolated by a single team, the distributed application is constituted by a group of teams. In PEACE, such a group of related teams is called a *league*, which defines an application-related communication environment. The most important aspect associated with leagues is that they provide a *communication firewall* between different applications. Thus, to support a multi-user mode of operation at least a two-level security mechanism is implemented in PEACE, for different jobs will be mapped onto different leagues and teams constituting a league will be isolated from each other.

2.4. Message Passing

Different PEACE entities are coupled by a communication system that is based on message-passing and implemented by a collective of small message-passing kernels. Each GENESIS node will be equipped with one message-passing kernel. Active elements within the entity are lightweight processes. Consequently, message-passing takes the form of *inter-process communication* rather than inter-node communication.

The PEACE message-passing kernel implements synchronous communication. More specifically, for application-level communication, CP communication semantics is maintained and provided without any performance limiting mapping overhead. The following basic communication services will be provided:

- *remote invocation send* for efficient implementation of *remote procedure calls*, i.e., system service requests
- *high-volume data transfer* as the means for network-wide data transfers between peer address spaces
- *synchronization send* for efficient implementation of user-level communication

These services all are directly supported by the CP; moreover, synchronization send will be the fundamental communication service provided by the CP and directly made available by the kernel. Asynchronous communication then is provided by lightweight processes on a library level and is performed by the entity itself.

2.4.1. Port-Based Addressing Scheme

For performance reasons, PEACE message-passing is non-buffered – therefore synchronous – and directly performed between processes addressed via ports. Each process is associated with a *gate*, which will act as a port-based communication endpoint without buffering capabilities, but rather routing capabilities. A communication endpoint will be represented by a *system-wide unique identifier*. This identifier contains a hint on which node the corresponding process is located. They will be normally associated in one-to-one correspondence with processes, i.e., the creation of a process also results in the creation of a process gate.

Gates can also be associated dynamically with processes, thus allowing dynamic reconfiguration in case of team migration as a result of load balancing. Moreover, there might be several gates associated with the same process, thus having more gates available than processes. Even after team migration, the kernel will automatically route an incoming message to the new destination process. In this sense, the gate takes the form of a *forwarding identifier* associated with a specific process.

2.4.2. Symbolic Naming Functions

In order to keep kernel complexity small and to achieve high-speed inter-process communication, dynamic reconfiguration in case of a node crash will not be exclusively supported by the kernel, but rather in co-operation with global naming functions provided by dedicated system processes. Because gates that have been managed by the kernel of a crashed node can no longer be used to address associated processes and because a system-wide unique gate identifier will have to contain a node hint for the gate object, the communication endpoint identifier for a process changes if the process is migrated from the crashed node onto a running node.

Local caching of gates (i.e., communication endpoint identifiers) associated with remote processes will help to maintain transparency even in the case of a node crash – at least if the cache store does not exhaust –, however only to the expense of communication performance. In this case, hashing strategies must be implemented and, most importantly, executed along with each message-passing activity. This organization would make it impossible to guarantee a 20 μsec message setup time, because all these activities must be executed by a PN (as well as ION) under the control of the sender/receiver process. For all these reasons, the kernel will not cache communication endpoint identifiers, i.e., gates, which normally are managed by remote kernels.

The solution to overcome the crash problem, while maintaining high-performance inter-process communication, can only be to consider a node crash as a system exception. Note, communication endpoints must already be known by the processes in order to address their peers, thus, these processes necessarily will have to cache corresponding communication endpoint identifiers. As a result of exception propagation, the processes itself are directed to get knowledge of the new endpoint identifier in case of a node crash. As part of exception handling, the processes itself will invoke PEACE naming services. These services are provided by system processes, i.e., *name server*, instead of being implemented by the kernel. Consequently, a PEACE entity itself is responsible to recover from a node crash, at least with respect to requesting the resolution of communication endpoint identifiers. In this approach, node crash transparency for the application program can be maintained by the runtime system, the system library or the service coupling.

If local caching on a node is avoided, the actual checking whether a message is to be forwarded, i.e., routed, introduces no loss of communication performance. However, in case of incoming messages, performance loss will be caused by the handling of interrupts and context switches in order to invoke the message-passing kernel. Therefore, routing on the basis of forwarding identifiers will be a typical functionality that is off-loaded from a node and performed by the CP. In order to let the CP perform forwarding of messages, gate objects must be handled accordingly. Thus, directing a CP to cache and flush process gates will be necessary, which is to be controlled by the PEACE message-passing kernel.

2.5. Interfacing to the Communication Processor

Communication will be actually performed by the CP, which takes the form of a *communication device* that is attached to each GENESIS node. Consequently, communication between a node and the CP is necessary in order to perform system-wide inter-process communication. For this purpose, *shared memory* is used to perform data exchanges between a node and the CP without copying overhead. In figure 3 the basic CP organization from the operating system designer viewpoint is illustrated.

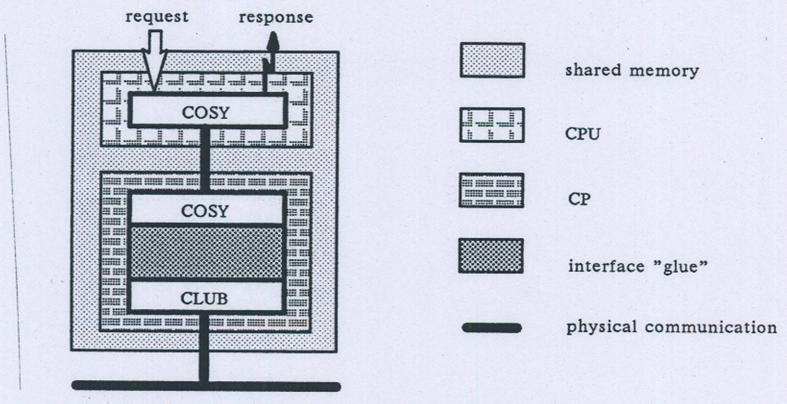

Figure 3: Organization of the Network Interface

Software packages being processed by a GENESIS node are interfaced to the CP via a communication system module, entitled *COSY*. This module contains low-level access routines which control the physical CP interface, i.e., deals with device registers in the traditional sense of system programming. The COSY interface is defined by a set of small, fixed-size request and response packets. For system maintenance purposes, these packets contain CP control and state information. In order to request communication, the packets contain message descriptors which refer to a message buffer located in the address space of a PEACE entity. Additionally, these *inter-process communication packets* must contain also communication endpoint identifiers, to uniquely address the sender and receiver process. The identifiers will be used by the CP to detect a rendezvous between two processes, which enables the CP to automatically setup DMAs in order to "blast" complete data segments between the sender and receiver address space [Giloi, Schroeder 1989].

The CP can be programmed to signal interrupts, indicating either exceptional conditions or the completion of a complete message transfer. A response packet then can be used to further specify the cause of the interrupt. Consequently, a polling mode of operation can be realized by the kernel by selectively switching CP interrupts off.

In order to complete communication between a node and CP, the CP site also has to implement COSY functionalities. Moreover, the actual network interface must be controlled. In the figure, this interface is represented by the *CLUB* module[2]. Thus, the primary function of the CP is to map COSY services onto CLUB services, in the case of sending a message, and to map CLUB services onto COSY services, in the case of receiving a message. In addition to these basic *gateway* functionalities, the CP is capable to perform routing and monitoring.

From the operating system designer viewpoint, the CP is a software processor, rather than being a hardware processor. The capabilities of this processor are defined by the COSY module, i.e., COSY is the most fundamental *abstract machine* on top of which the message-passing kernel of a GENESIS node will be layered. Therefore, in the following CP is considered as a synonym for this abstract machine and does not necessarily stand for the hardware communication processor.

3. Connecting PEACE with CHORUS

The interconnection of two different operating systems usually is realized by a *guest level* implementation of one of these systems on top of the other one. This would mean, for example, to have a PEACE guest level available on top of CHORUS, or vice versa. With this approach it may be difficult for programs to directly access services of both basic systems. In case of a PEACE guest level on top of CHORUS, programs that are directly running on CHORUS are not aware of PEACE and, thus, will not be able to directly apply PEACE services. Unfortunately, the user will be faced with a heterogeneous operating system interface.

The approach for the GENESIS distributed operating system is to interconnect PEACE and CHORUS directly at the lowest possible level of inter-process communication. The idea is that CHORUS processes and PEACE processes shall be able to directly communicate with each other and, most importantly, that PEACE processes shall be able to directly invoke CHORUS system services, and vice versa. In figure 4 this interconnection principle is illustrated.

In this organization, no guest level implementation is realized, but rather the message-passing kernels of PEACE and CHORUS are directly coupled. The fundamental interface to the network hardware is provided by the CP[3]. This low-level software module will be present on

[2] CLUB stands for cluster bus. It is derived from SUPRENUM, in which the cluster bus represents the basic interconnection media within a cluster. In the PEACE context, the CLUB module hides network device specific details.
[3] Remember, CP actually will be represented by a low-level, problem-oriented abstract machine interface.

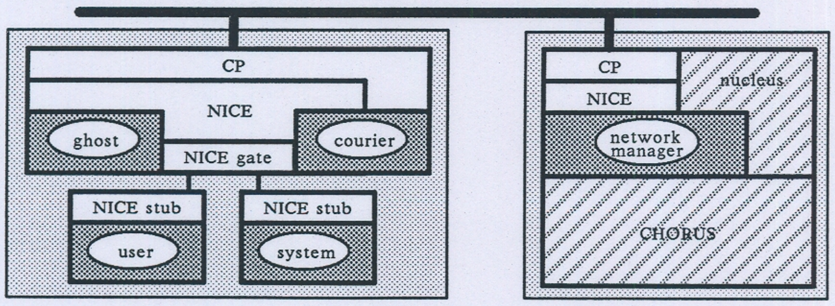

Figure 4: Symbiosis between PEACE and CHORUS

each GENESIS node and interfaces high-level software modules directly to the network, i.e., the hardware CP. The basic function of CP is to provide for some means of *inter-node communication*. Above CP either PEACE or CHORUS modules will be layered. Moreover, user programs may be directly attached to this module if inter-node communication completely meets the user requirements, rather than having system-wide inter-process communication. The following two subsections explain the low-level organization of PEACE and CHORUS nodes in some more detail.

3.1. The PEACE Site

Usually, above CP merely the PEACE message-passing system is layered. This system implements a network independent communication executive, called NICE, that offers network-wide synchronous inter-process communication facilities. In PEACE, traditional operating system services are exclusively provided by corresponding system processes. By this way, NICE merely enables the interconnection between the processes in terms of message-passing by the means of the PEACE *dispatching protocol*, and nothing else.

Each PEACE process has to use NICE primitives in order to interact with its outside world. In case of user mode processes, NICE primitives for sending and receiving messages are invoked via the *NICE gate* module. This module implements a system call interface in the traditional sense, namely it handles *system call traps*. The issuing of these system calls is made hidden by means of the *NICE stub* module which is a traditional system call library. Either user processes or system processes will use the NICE stub to switch from user to supervisor mode, i.e., to invoke the message-passing kernel.

The NICE module represents the heart of the PEACE message-passing kernel. Other functionalities are provided by lightweight *kernel processes* on a remote procedure call basis. A standard representative of these kernel processes is the *ghost*, which provides node-bounded system services related to the message-passing system. This process, together with NICE, builds up the *kernel team*.

Dependent on the kernel configuration, lightweight processes may be added to the original kernel team, such as the *courier*. The task of this process is to interface PEACE processes (i.e., teams) to the physical CP attached at each GENESIS node, thus acting as a *device deputy*. By means of a CP process representation, control of the CP is made feasible on a message-passing basis; more specifically, on the basis of remote procedure calls. For this purpose, the courier needs direct access to the CP and, thus, basically acts as a gateway between PEACE (i.e., CHORUS) and the CP. However, note that system-wide communication is directly supported by the CP, without any intervention of the courier. Merely NICE as well as the CP determine the software overhead accounted with a message setup, i.e., with the communication between a PN and its CP.

Both processes, ghost and courier, control a PEACE kernel entity (highlighted by the shaded boxes). With respect to the courier, the entity gives PEACE processes a complete illusion of the CP. Both processes are encapsulated by the same team, which is always executed in supervisor mode.

3.2. The CHORUS Site

Concerning CHORUS, the CP takes the form of a low-level device driver that has to be included in a host operating system release. This modul will be used by the CHORUS *network manager* to attach the CHORUS message-passing kernel, i.e. the *nucleus*, to the network. By this way, the mechanisms for inter-process communication of the CHORUS operating system are extended to support distributed GENESIS application programs.

In order to couple PEACE and CHORUS, the network manager maps PEACE primitives for inter-process communication onto the corresponding CHORUS primitives, and vice versa. More precisely, the network manager executes the PEACE dispatching protocol as implemented by the PEACE-site NICE. For this purpose, NICE as well as CP representations are required at the CHORUS site such that the network manager can act as a gateway between PEACE and CHORUS.

Having coupled CHORUS to the GENESIS machine, i.e., to PEACE, a complete set of UNIX system services is directly available. User programs being executed on GENESIS nodes can access these services with a minimum of system overhead. These programs are represented as PEACE processes and, thus, will use the PEACE model of inter-process communication. By means of the network manager, a standard CHORUS concept will be used to make UNIX services directly available to PEACE processes.

4. Basic Node Operating System Configurations

As pointed out previously, PEACE represents the GENESIS node operating system and, thus, provides a process execution and communication environment for GENESIS programs being executed on PNs as well as IONs. Therefore, each PN must be supplied with a distinct set of PEACE services. In order to satisfy different user requirements, different PEACE configurations are offered [Behr et al. 1988].

The major aspect of these configurations is whether single-processing or multi-processing is supported. In the former case, the operating system overhead of inter-process communication is negligible and, concerning inter-process communication performance, the applications are supported in the best possible way. However, large distributed applications may suffer from low scalability, for the limited number of nodes. These scalability problems can be solved if multi-processing support is given, however only to the expense of communication performance.

In the following subsection, the major configurations are discussed in detail.

4.1. Single-Processing

The simplest configuration one can think about, at least from the user point of view, is to have only one process, i.e., one program, that is loaded onto a node. This means that on this node no multi-processing support will be necessary and, therefore, no system overhead concerning process scheduling, process dispatching, context switching and interrupt handling is produced. In case of a distributed program, communication between the different processes will be possible on a *communication library* basis.

4.1.1. Instantaneous Loading

Loading the distributed application means to execute the PEACE bootstrap procedure for the corresponding nodes. The entire user job will be instantaneously loaded during this procedure, without the opportunity to load additional program entities at runtime.

The number of processes, i.e., nodes, for the user job must be known in advance, namely at job generation time. A PEACE system configuration utility is used to generate (configure) the user job. This utility enables the user to specify the job distribution over a dedicated set of nodes. The original application program itself is not involved in this configuration specification, rather a separate *configuration language* is used for this purpose. Figure 5 depicts the resulting PEACE configuration.

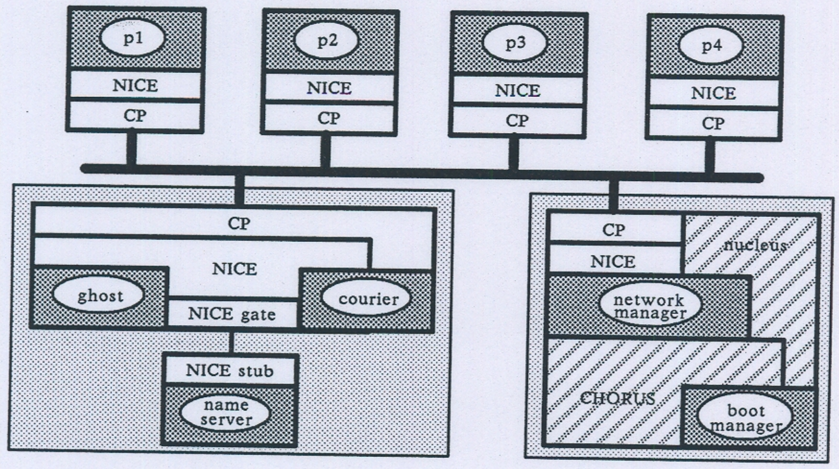

Figure 5: Configuration for Instantaneous Loading

The *boot manager* interprets the configuration specification and controls the bootstrap procedure for the given set of GENESIS nodes. For each node, load images are read from the CHORUS file system and transferred over the GENESIS network. Corresponding to the bootstrap protocol executed by the boot manager, the GENESIS nodes must be equipped with memory resident software, for example on EPROM basis. This software at least implements the low-level bootstrap protocol with the boot manager.

The communication system is represented by a dedicated NICE module and by the original CP. The NICE module merely manages a single process, nevertheless it provides the complete set of PEACE primitives for inter-process communication. Both modules are directly linked as a communication library to the PEACE entity and, thus, are directly interfaced to the CP. In the exemplified configuration, communication between the processes *p1*, *p2*, *p3* and *p4* is supported with a minimum of system overhead.

As outlined in a previous section, the programs are represented as PEACE entities and, thus, are able to invoke operating system services although these services are not locally implemented. In PEACE, each system service will be accessed on a remote procedure call basis. Fundamental PEACE operating system services are provided by the kernel team and by the *name server*. The name server keeps track of logical process and/or service names and their associated real system processes, i.e. the connection endpoints of these processes. By this way, application processes can address each other independently of their actual node addresses. The same mechanism is used to address service providing system processes.

Although the major aspect of the illustrated configuration is to merely support a single-processing mode of operation, multi-processing is provided also. In this configuration, one node of the GENESIS machine is used as a *PEACE server node* that processes the ghost, the courier and the name server. Dependent on the overall system organization, there may be a couple of server nodes, for example associated with each user job, likewise. At least for server nodes a separation is made between supervisor mode execution and user mode execution, as indicated by the broken line. The PEACE message-passing kernel, assisted by ghost and courier, will be executed in privileged supervisor mode as a team of lightweight processes.

4.1.2. Incremental Loading

An obvious solution to avoid a static description of distributed programs is to load additional nodes with program entities upon program request. The initial PEACE bootstrap leads to the execution of at least one user job entity. Merely the initial (*root*) entity must be specified in the configuration description. Dependent on the program state, further entities are loaded at runtime, on demand. For this purpose, additional system services must be introduced, as illustrated in figure 6.

The difference to the previously discussed configuration is given by two additional system processes. The *pool server* basically keeps track about the actual utilization of a specific pool of nodes. If a request for loading another entity is indicated, the pool server allocates a free node for the entity. In the reverse case, if a program entity decides to relinquish control of the node, the pool server is notified accordingly. In order to take advantage of a pool of nodes, the pool server provides proper services and makes these services globally known by the name server. Moreover, several pools may be available, each one managed by its own pool server. By this way *logical clusters*, rather than physical ones, can be provided.

Loading a program entity will be handled by the *boot server*. This system process initiates and controls the bootstrap procedure for a specific node. A load image is read from the CHORUS file system and transferred to the corresponding node. Basically, the selected node is forced to executed the memory resident bootstrap protocol with the boot manager. For this purpose, the GENESIS communication system has to provide low-level mechanisms to

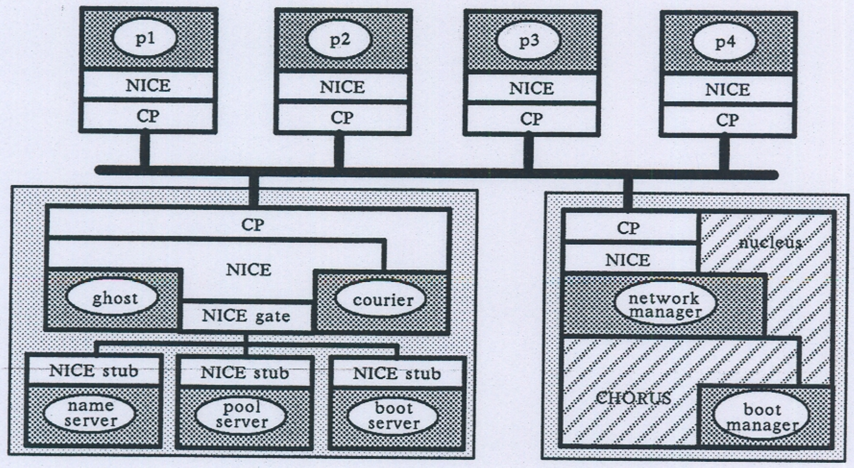

Figure 6: Configuration for Incremental Loading

selectively reset single nodes. Having finished bootstrapping, the boot server is notified, i.e., the process having been loaded invokes a boot server system service during his setup activity. This enables the boot server to transfer environment data from the process having requested the load service to the process having been loaded.

From the system point of view, the application processes itself invoke pool server as well as boot server services. However, from the application point of view requesting these services is made hidden by the runtime system and the PEACE system library.

4.2. Multi-Processing

Support for multi-processing means being based on a NICE module that, roughly spoken, manages a couple of processes. In the previous configurations, multi-processing was already introduced, however it was not associated with PNs but rather with PEACE server nodes. Remember, PEACE system services have been provided by a couple of system processes executed by a server node. Thus, at least multi-processing on this node type was necessary.

For security reasons in a multi-processing mode of operation, the NICE module must be physically separated from untrustworthy user processes. These processes are executed in user mode of the underlying processor, whereas critical system services are executed in supervisor mode. There is also a traditional technical reason for this separation. The NICE modules encapsulates operations and data structures needed to multiplex a single processor between several processes. This multiplexing functionality is shared by all processes of the node. In its simplest form, the NICE module is a *shared library* which is not directly linked to the PEACE entities. In such a organization, the trap system of the underlying processor is the standard mechanism in order to access the services provided by NICE. Figure 7 shows an example configuration.

This configuration focuses on two aspects. The first aspect is that user programs now will be executed in user mode and that the corresponding processes (*p1*, *p2* and *p3*) are controlled

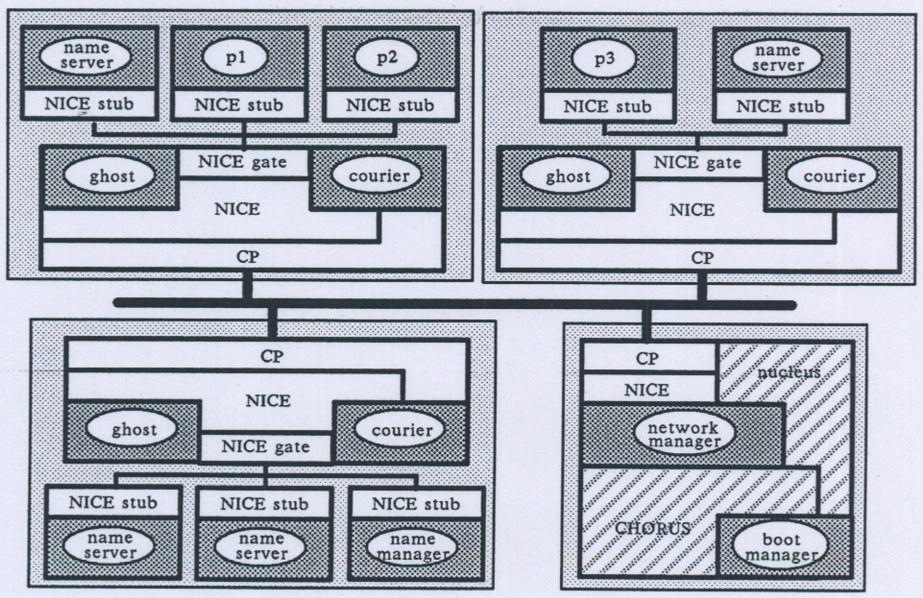

Figure 7: Multi-Processing Configuration

by a multi-processing NICE module. The second aspect is that for different user programs different name server are available. In the exemplified configuration, process *p1* and *p2* are associated with a name server and process *p3* with another one. Consequently, the name spaces for these processes are different, in this example they are node-related. In PEACE, the coupling of different name spaces is accomplished by another name server. In the example, the PEACE server node is used for this purpose. On this node, two name server processes are available. One name server to handle node-related services and another one to implement the common name space for the three user processes.

There is one more system process to be discussed, namely the *name manager*. Basically, this process realizes the coupling of different name spaces and, thus, makes it feasible to address global system services. In a similar fashion, this system process enables process *p3* to logically address process *p1* and process *p2*.

4.3. Fusion of Basic Configurations

A natural conclusion is to combine the different organizations discussed so far. The major problem merely is to configure PEACE in such a way that all the necessary system services, i.e. server processes, are present. As mentioned previously, once having implemented the services this is a question of the configuration specification being interpreted by the boot manager. In figure 8 the mixed organization is illustrated.

The major aspect of this system configuration is that user processes will only pay for services they really need. For example, process *p1* and process *p2* are executed in a multi-processing mode and are physically separated from the NICE module. In contrast to that, process *p3* and process *p4* are executed in a single-processing mode and have complete node

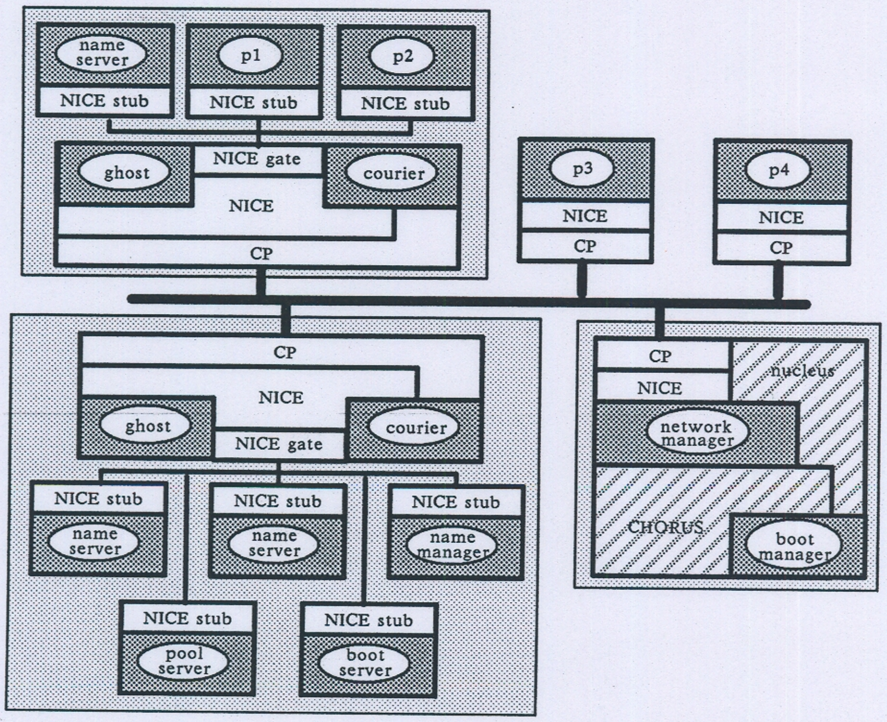

Figure 8: A Union of Basic Configurations

control. Merely process *p1* and process *p2* suffer from a larger portion of system overhead.

5. Dynamically Extending the Node Operating System

The purpose of the discussion in the previous sections was to illustrate basic PEACE configurations and to show by what means, step by step, a more powerful distributed node operating system can be build that satisfies different user requirements. As long as enough nodes are available, a distributed program that needs no multi-processing support is not handicapped by the system.

In the following subsection, further PEACE services are discussed, briefly. The common framework for these services is that they are provided by dedicated processes and, thus, can be configured in the same way as discussed so far. Moreover, these services can be dynamically created as well as destroyed at runtime, leading to a dynamically alterable node operating system.

5.1. Memory Management

Offering memory management services for a multi-processing system at least requires three basic PEACE services. The first service is concerned with programming the memory management unit. For this purpose, a lightweight process is added to the kernel team of the corresponding node. This process is a software representative, a *device deputy*, of the memory management unit. By this means, programming the memory management unit is done in an object-oriented way and is directly based on network-wide message-passing. The second service is concerned with handling traps signaled by the memory management unit. Generally, traps are translated into messages and transferred to dedicated system processes. This works network-wide, i.e., the *trap server* receiving the trap message and the process having caused the trap (and sending the trap message) need not to reside on the same node[4]. The third service is concerned with memory management and is provided by the *memory server*. Service functions are available to extend, reduce, lock, unlock, share, swap and shadow memory segments.

The trap server and the memory server may belong either to the same team or to different teams. In the latter case, both processes (i.e., teams) may be assigned to different nodes. Considering the entire memory management system, the three processes may be placed on different nodes. On each node, the device deputy is necessary if the memory management unit must be used.

5.2. Virtual Memory and Distributed Shared Memory

In order to implement virtual memory, the PEACE memory management system is to be extended by at least one basic service. This service is concerned with disk i/o, in order to transfer memory pages between primary and secondary storage. For this purpose the *disk server* is introduced and closely located to the actual mass storage device.

It is a function of the memory server to distinguish between a paged and a non-paged mode of operation. In the former case, a trap signaled by the memory management unit is considered as a page fault, whereas in the latter case it is an address space violation. The only difference is the way the trap is handled by the memory server. Being based on a paged memory management unit, the question of having a paging system is basically a question of having configured the memory server with the proper trap handler. That is to say, larger system overhead caused by paging will be determined exclusively by the memory management software and not by the memory management hardware.

The disk server encapsulates the knowledge about the organization of the mass storage device. However, it is by no means necessary to have a real disk device attached to the disk server. One can also request a node from the pool server, place a dedicated disk server onto this node and then let this disk server manage the entire physical memory of this node in terms of a page-oriented mass storage device. Thus, the disk server implements a *RAM disk*. Paging from/to the RAM disk will be realized by the fastest possible way, namely by message-passing, and implies no time consuming device handling.

If applicable, several disk server processes may be used in order to extend the page store. Nevertheless, this organization can only be a compromise, for the lack of a really large disk page store. Application programs, however, with a predicted working set size can be optimally supported. The disk server then takes the position of an application-related working set server. As an example, such a server then can be used to provide *virtual shared memory* services.

[4] Exactly in this way UNIX system call emulation will be supported by PEACE. The system call is sent as a trap message to a UNIX emulator being executed by CHORUS.

This organization is also useful for the implementation of *distributed shared memory* principles, which may be much more important than traditional virtual memory support. Instead of being solely an abstraction from a mass storage device, the disk server is used to implement a *page cache* for the non-distributed address space of a large distributed application. The cache will either manage the entire non-distributed address space or parts of it. By this means, the distributed-memory GENESIS machine can support MIMD programs that have been developed for non-distributed, shared-memory multiprocessor systems.

5.3. Team Management

Incremental loading of distributed application programs already implies dynamical team management, however only in terms of loading a single team onto the selected node. Excepted at bootstrapping time, creating more than one team on a single node was not supported by the configurations discussed so far. Rather, this additional service function is provided by the *team server*. A consequence of this service is the necessity of multi-processing on the node of the newly created team and/or process.

Usually, a single team server is responsible for team management on several nodes. The *ghost* of each node assists the team server in order to install or remove process objects as well as team objects. Creation of a process object results in the creation of a system-wide unique identifier which, as explained above, addresses the process gate. This identifier is created by the ghost upon a request to install a process object and it is used by NICE message-passing primitives to address the message recipient.

Allocation and deallocation of process gates will be the function of the team server. Moreover, the team server signals system exceptions if a process gate is destroyed, which means that the associated process has been destroyed. Exceptions will also be raised in case of team migration, for the binding between process gates and process objects has changed.

5.4. Synchronization

One of the major problems of distributed systems is to globally synchronize concurrent activities. With respect to GENESIS, these activities are clock synchronization, writing of checkpoints, recovery from checkpoints and restructuring.

One basic PEACE synchronization is realized by the naming system. The existence of a specific name may be interpreted as the occurrence of the associated event and/or activity. In this sense, processes are able to synchronize on a specific event by querying the naming system. Having completed bootstrapping, exactly in this way the PEACE startup procedure works. Nevertheless, the problem exists to force processes into a specific synchronization phase, i.e., to freeze the state of a set processes.

In PEACE, global synchronization is controlled by the *ruler*. This system process uses the *signal server* to propagate synchronization events and, thus, forces processes to synchronize. The processes involved in synchronization are descheduled until the ruler cancels the freezing state, i.e., terminates the synchronization phase.

5.5. Fault Tolerance

A machine consisting of thousands of tightly coupled nodes will only be workable in case of a fault tolerant system design. The GENESIS hardware architecture is fault tolerant, at least with respect to the large number of autonomous nodes coupled by a message-passing network. This capability must be preserved by the software architecture, especially by the node operating system.

The major aspects of fault tolerance in PEACE are classified by the keywords *checkpointing*, *backward error recovery* and *restructuring*. The mechanism to restructure a complex of related teams is based on process migration techniques. Therefore, *team migration* will be supported, however not primarily for load balancing purposes. Restructuring is necessary in case of a node crash, i.e., teams are migrated from a crashed node onto other nodes and are recovered from the previously written checkpoint, accordingly.

The fundamental basis for implementing fault tolerance is the ruler, as explained in the previous section. During the synchronization phase of a group of processes/teams, checkpoints are written, previously written checkpoints are recovered and restructuring is performed. For these activities, dedicated system processes are provided by PEACE.

In order to improve the performance of checkpointing and recovery, a *log-structured file system* organization is used. On this basis, a single checkpoint will be represented by a single *logfile*, whose datablocks are stored on the disk in sequential order. Thus, reading/writing a checkpoint will be possible with a minimum of disk seek overhead. Moreover, several versions of logfiles belonging to the same checkpointed application will be maintained, implementing a checkpoint history for the application.

5.6. Diagnosis

A precondition for the implementation of software fault-tolerance is to detect hardware as well as software failures. The approach is to configure PEACE with a set of dedicated *sensor processes* which will implement various supervisor functions. In case of a system failure, the sensor processes will request recovery of the corresponding system component.

It is not only software, i.e., system server, which will be supervised by sensor processes. Besides global checking the state of clusters or nodes, it will be also important to perform fine grain hardware checks related to memory, the vector floating point unit, the CP and other devices. There must be the opportunity to perform these checks at system bootstrapping time as well as system runtime.

5.7. Debugging, Monitoring and Accounting

For the development of distributed GENESIS programs, enhanced debugging mechanisms must be provided on the nodes. It is necessary to have the opportunity to observe and manipulate the processor state of a process from remote, i.e., from some server station. Because distributed GENESIS programs are based on message-passing, the same holds for the observation and manipulation of messages. Both functionalities, i.e., debugging and monitoring, are supported by PEACE.

Tracing and handling of breakpoints is directly based on message-passing, because in PEACE traps are translated into messages and sent to some trap server. By default, the trap message contains the processor state of the debugged process. Changing the processor state is done either by replying a trap message to the process or by applying the NICE primitives for high-volume data transfer.

Monitoring is based on the NICE routing facility. Transparently for the sending/receiving processes, messages can be routed to a *monitor*, system-wide. In case of different application processes, different monitor processes may be introduced; whereby each monitor process is programmable to filter out application-oriented message data.

A specific type of monitoring will be associated with *job accounting*. Accounting data collected for a user job at least will consist of the number of nodes and teams used to execute the job, the actual CPU time (user, system, kernel), the weight of the job in terms of memory utilization, message-passing and file store usage. The information is gathered by dedicated PEACE system processes and made available to system programs being executed by the host operating system, i.e., CHORUS.

5.8. UNIX System Call Compatibility

As mentioned above, UNIX system call emulation will be supported by the propagation of *trap messages* to CHORUS. Upon trap detection, by a low-level PEACE trap handler being part of the message-passing kernel, a *trap server* will be requested to further process the trap. This trap server represents the *UNIX emulator* which will normally be part of the host operating system, i.e., CHORUS, and maps UNIX system calls encoded in trap messages accordingly.

The binding to trap server processes is team relative and dynamic. Different teams may be controlled by different trap server. In addition to that, a single team may be bound to several trap server, each of which serving different trap types such as UNIX system calls, CP exceptions, page faults, address space violations, etc. Dependent on the trap type, a trap server may be directly part of the team which causes the trap, e.g. in case of *copy-on-write* traps to support asynchronous communication by dedicated memory management functions. Any way, trap propagation always is implemented on a message-passing basis and, therefore, works system-wide.

6. Architecture of the Host Operating System

The GENESIS host operating system will be a distributed implementation of a UNIX operating system. Conformity with the X/OPEN and POSIX standard is one of the major requirements. The host operating system provides the *programming environment* for distributed program development as well as for the development of distributed programs for GENESIS. Access to and *maintenance* of the PNs and IONs will be controlled, or control will be supported, by proper system programs.

In order to easily extend UNIX services into a distributed environment, the GENESIS host operating system will be represented by CHORUS, which then is responsible for the control of IONs, server stations and other host machines being attached to the GENESIS core system. Concerning the fundamental building blocks to construct a distributed operating system, there are many similarities between PEACE and CHORUS, making it easy to provide a homogeneous user interface.

6.1. Overall Organization

As PEACE, the host operating system, i.e., CHORUS, is composed of a number of *system servers*, thus being *process-structured*. System server cooperate in the context of *subsystems* providing a coherent set of services. Concerning GENESIS, an important subsystem will be UNIX. Interaction between system server as well as between user processes and system server is by the means of *message-passing*. For this purpose, a small-sized *nucleus* provides proper message-passing services. Figure 9 shows the overall architecture.

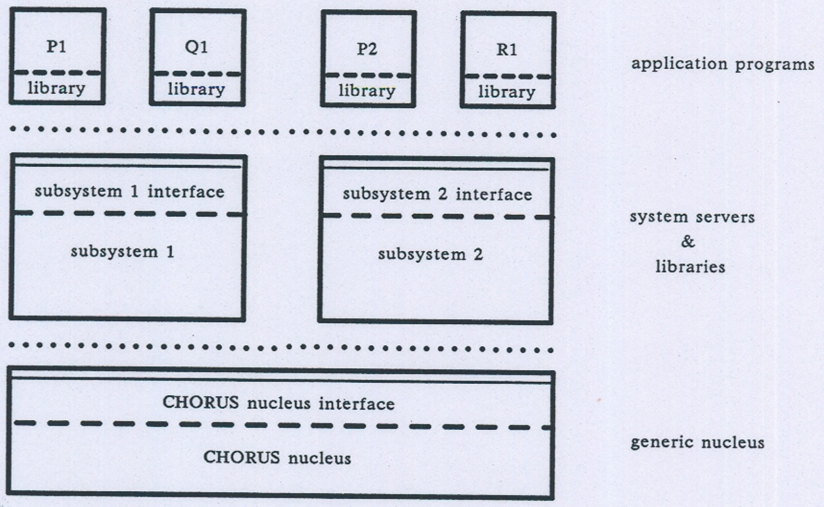

Figure 9: The CHORUS Architecture

System servers implement high-level system services, and cooperate to provide a coherent operating system interface. They communicate via the message-passing facility provided by the CHORUS nucleus. Several levels of interface can be exhibited. The *nucleus interface* is composed of a set of procedures providing access to the services of the nucleus. If the nucleus cannot render the service directly, it communicates with a distant nucleus via message-passing. The *subsystem interface* is composed of a set of procedures accessing the nucleus interface, and some subsystem specific protected data. If a service cannot be rendered directly from this information, these procedures invoke the services provided by system servers on a remote procedure call basis.

Both interfaces are enriched by libraries. Such libraries permit the definition of programming-language specific access to system functionalities. These libraries (e.g., the UNIX C library) are made up of functions linked into and executed in the context of user programs.

The nucleus plays a double role, namely offering local services as well as global services. With respect to local services, it manages, at the lowest level, the local physical computing resources of a server station, called a *site*, by means of three clearly identified components:

- allocation of local processor(s) is controlled by a *real-time multi-tasking executive*. This executive provides fine grain synchronization and priority-based preemptive scheduling,
- local memory is managed by the *virtual memory manager* controlling memory space allocation and structuring virtual memory address spaces,
- external events – interrupts, traps, exceptions – are dispatched by the *supervisor*.

With respect to global services, system-wide inter-process communication (IPC) is provided by the *IPC manager*, delivering messages regardless of the location of their destination within a collective of CHORUS systems. The IPC manager supports *remote procedure call* facilities and *asynchronous* message exchange, and implements multicast as well as functional addressing. It may rely on external system servers (i.e., network managers) to operate all kinds of network protocols. See figure 10 for the general structure of the CHORUS nucleus.

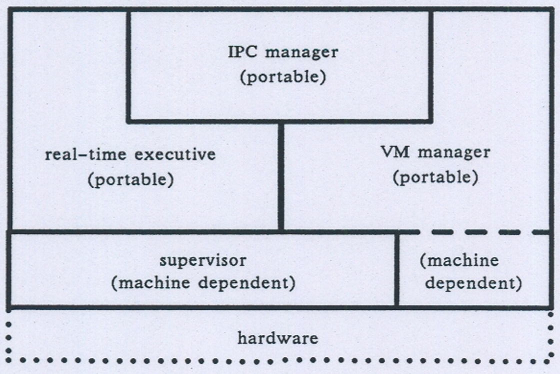

Figure 10: The CHORUS Nucleus

6.2. Basic Abstractions

There are five basic abstractions which will have counterparts in PEACE. The *actor* (team) is the unit of resource allocation and memory address space. A *thread* (lightweight process) is the unit of sequential execution and a *port* (gate) is the unit of addressing and (re)configuration basis. In both cases, the unit of communication is represented by a *message* and a global name is represented by a *system-wide unique identifier*. These types of abstractions are implemented and managed by the CHORUS nucleus as well as by the PEACE kernel. However, the actual representation will be different in both systems, for having to meet different application requirements[5]. There are different concerns, which is expressed by using different names for these abstractions in both the node operating system (PEACE) and the host operating system (CHORUS).

[5] For the execution of distributed MIMD programs on the GENESIS core system scaled down and very-high performance system services are necessary; whereas for the development of these programs enhanced system services providing a comfortable programming environment as well as traditional timesharing functionalities are required.

There are further basic abstractions which are not extended into the node operating system: *port groups* and *regions*. A port group combines several ports and provides multicast or functional addressing capabilities. A region is the unit of structuring an actor address space.

All these abstractions correspond to object classes which are private to the CHORUS nucleus: both the object representation and the operations on the objects are managed by the nucleus. Those basic abstractions are object classes to which the services invoked at the nucleus interface are related.

Three other abstractions are also managed both by the CHORUS nucleus and subsystem actors. The *segment*, as the unit of data encapsulation, the *capability*, as the unit of data access control and the *protection identifier* as the unit of authentication. For a more detailed description, see [Rozier et al. 1988].

6.3. UNIX as a CHORUS Subsystem

The major subsystem being provided for GENESIS is concerned with the implementation of basic UNIX abstractions. Some of the abstractions are already implemented by the CHORUS nucleus and, thus, are provided by corresponding nucleus calls. Others are implemented in terms of CHORUS actors.

In the following subsections, the main design decisions are given and the general architecture is presented. Some implementation choices are explained in more detail, the emphasis being on solving problems which arise when introducing distributed processing into a UNIX system.

6.3.1. Objectives

Applying the operating system family concept to reorganize UNIX covers a number of well recognized limitations of current ("traditional") UNIX implementations. It has been applied with the following general objectives.

Modularity

To implement UNIX services as a collection of servers, so that some may be present only on some sites (such as file managers, device managers, etc.), and when possible build them in such a way that they can be dynamically (without stopping the system) plugged into/out the system when needed. This true modularity will allow simpler modifications and maintenance because the system is built of small pieces with well-known interactions.

Openness and Expandability

To permit application developers to implement their own servers (e.g., time manager, window manager, fault-tolerant file manager) and to integrate them dynamically into the UNIX subsystem.

Extending UNIX Functionalities

To extend UNIX with the real-time facilities provided by the low-level nucleus real-time executive and to operate UNIX in a distributed environment with no limitations on the types of resources shared. Furthermore, to extend UNIX services with services provided by the nucleus, e.g., multi-threaded UNIX processes.

Orthogonality

To keep UNIX specific concepts out of the nucleus. But in turn, to use such concepts as actors, threads, ports, etc., to implement UNIX ones outside of the nucleus. This allows other subsystems (e.g., *object oriented systems*) to be implemented also on top of the nucleus without interfering with the particular UNIX philosophy.

Compatibility

On a given machine, to be compatible at the executable code level with a given standard UNIX system (e.g., System V Release 3.2), to ensure complete user software portability. To be able to adapt a UNIX driver into a UNIX server on CHORUS with a minimum effort and to provide the same services about as fast as a given UNIX system on the same machine architecture (i.e., the one chosen for binary compatibility).

6.3.2. Extensions to UNIX Services

The file system is fully distributed and file access is location independent. File trees can be *automatically interconnected* to provide a name space where all files, whether remote or local, are designated with homogeneous and uniform symbolic names, and in fact with no syntactic change from current UNIX. Operations on processes (at the *fork/exec* level as well as at the *shell* level) can be executed regardless of the execution site of these processes; on the other hand, the creation of a child process can be forced to occur on any given compatible site.

Network transparent CHORUS inter-process communication is accessible at the UNIX interface level, thus allowing the easy development of distributed applications within the UNIX environment. More importantly, the interconnection between CHORUS and PEACE at the low-level of inter-process communication enables user processes, being executed on server stations, to directly communicate with numerical processes, being executed on the GENESIS core system.

Distribution extensions to standard UNIX services are provided in a natural way (in the UNIX sense) so that existing applications may benefit from those extensions even when directly ported onto CHORUS, without modification or recompilation. This applies not only to file management but also to process and signal management.

Multiprocessing within a UNIX process is possible with the concept of *U_thread.* A U_thread can be considered as a lightweight process within a standard UNIX process. It shares all the process' resources and in particular its virtual address space and open files. Each U_thread represents a different locus of control. Thus when a process is created by a *fork*, it starts running with a unique U_thread; the same situation occurs after an *exec*; when a process terminates by *exit*, all U_threads of that process terminate with it.

With each U_thread is associated a list of signal handlers. Depending on their nature, signals are delivered to one of the process U_threads (alarm, exceptions, etc.) or broadcast to all process U_threads (DEL, user signals, etc.). Inter-process communication and U_thread synchronization rely on the CHORUS message-passing functionalities provided by the nucleus. The same holds for real-time facilities with priority based scheduling and interrupt handling.

6.4. The UNIX Subsystem Architecture

UNIX functionalities may logically be partitioned into several classes of services according to the different types of resources managed: processes, files, devices, pipes. The design of the structure of the UNIX subsystem gives emphasis on a clean definition of the interactions between these different classes of services in order to get a true modular structure.

The UNIX Subsystem will be realized as a set of system servers, running on top of the nucleus. Each system resource (process, file, etc.) is isolated and managed by a dedicated system server. Interactions between these servers are based on the inter-process communication which enforces clean interface definitions.

As illustrated in figure 11, several types of servers may be distinguished within a typical UNIX subsystem: *process managers*, *file managers*, *pipe managers*, *device managers* and *user defined servers*.

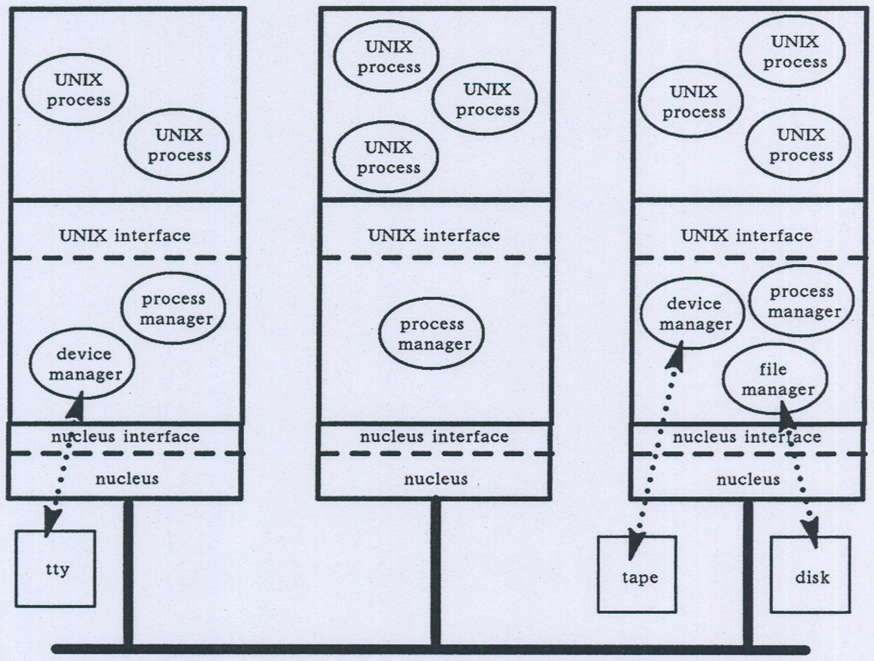

Figure 11: A Distributed UNIX

The following sections describe the general structure of UNIX servers. The role of each server and its relationships with other servers are summarized.

6.4.1. Structure of a UNIX Server

UNIX servers are implemented as actors. They have their own context (virtual memory regions, ports, etc.) and thus may be debugged as "standard" actors. Several servers – file manager, device manager, process manager – will be multi-threaded, allowing the parallel execution of system services. Their threads may be executed either in user mode or in – protected – system mode. Each request to a server is processed by one thread of this server which manages the context of the request until the response is complete.

Simpler servers may be mono-threaded (e.g., the pipe manager). When a request cannot be served immediately, it is queued by the server, in a private queue. Thus the server is not blocked waiting for an event in order to complete a request. When the server detects that an expected event has occurred, it processes the requests in the queue.

Each server creates one or more ports which clients send requests to. Some of these ports may be inserted into port groups with well-known names. Such port groups can be used to access a service independently of the server which will actually provide it.

In order to provide compatibility with existing UNIX system services, servers may also attach some of their own routines to system traps. A service (e.g., *open*) is realized partly by a stub which executes, in system mode – behind a trap –, in the context of the calling client process. This routine manages the context of the process and when needed, invokes the appropriate subsystem server on a remote procedure call basis.

In order to facilitate porting drivers from a UNIX kernel into a CHORUS server, a UNIX kernel emulation library, to be linked with UNIX driver code, will be provided with such functions as *sleep*, *wakeup*, *splx*, *copyin*, *copyout*, etc.

6.4.2. Process Manager

The *process manager* maps UNIX process abstractions onto CHORUS concepts (actor, thread, regions, etc.). It implements all of the UNIX process semantics including process creation, context consistency and inheritance, and process signaling.

On each UNIX site, a process manager implements the entry points used by user processes to access UNIX services (other UNIX servers are not accessed directly by user processes). To maintain binary compatibility, these services are accessed through traps. The process manager uses the nucleus facility to attach routines to these traps.

Those routines connected to traps either call other process manager routines to satisfy requests related to process and signal management (*fork*, *exec*, *kill*, etc.) or invoke via remote procedure calls file managers, pipe managers or device managers to handle other requests (e.g., *read/write* on files). Thus, process managers interact with their environment through clearly defined interfaces.

Process managers cooperate to implement remote execution and remote signaling. Each process manager dedicates a port (the *request* port) to receive remote requests. Those requests are processed by process manager threads. The request port of each process manager (of a given CHORUS domain) is inserted into one port group. Any process manager of any given site may thus be reached through one unique functional address: the port group name.

6.4.3. File Manager

There is one file manager on each site supporting a disk. Diskless stations don't need a local File Manager. File Managers have two main functions:

- to provide UNIX file system management (compatible at disk level),

- to act as a segment server (called *mapper*) to the nucleus, managing segments (i.e., files) mapped into actors contexts (executable binary files, swap files, etc.).

As UNIX file servers, they process UNIX requests transmitted by the means of remote procedure calls (*open*, *stat*, etc.). In addition, file managers provide some new services needed for process management:

- sharing (on *fork*) and releasing (on *exit*) directories and files between processes,
- associating capabilities to text and data segments of executable files, used to create regions in processes contexts.

These two services represent the only interactions between process and file management.

As external mappers, file managers implement services required by the CHORUS virtual memory management: swapping pages in/out, creating swap files, etc. They use the standard mappers interface services provided by the nucleus to control and to keep consistent the data transferred between CHORUS sites when local virtual memory caches are involved: flush, invalidate pages, etc.

To avoid maintaining unnecessary copies of pages of a given file in physical memory, and to optimize physical memory allocation, CHORUS virtual memory mechanisms are used to implement file system caches and some of the UNIX calls such as *read*, *write*, etc.

The naming facilities provided by the UNIX file system will be extended, to permit the designation of services accessed via ports. *Symbolic port names* (represented by a new UNIX file type) can be created in the UNIX file tree. They associate a file name to a port unique identifier. This is very similar to the PEACE naming function and to UNIX device designation. When such a name is found during the analysis of a pathname, the corresponding request is forwarded to the port to which the unique identifier is associated – marked with the current status of the analysis. Servers can be designated by such symbolic port names. In particular, this functionality is used to interconnect file systems and provide a global name space.

Transparent access to remote files is provided through *symbolic links*. When such a symbolic name is found during the analysis of a pathname, the pathname is transformed into a new one and the analysis is restarted. The file system generates the new pathname by prefixing the non yet analyzed part of the pathname with the value of the symbolic name – this value is also a pathname.

6.4.4. Pipe Manager

A pipe manager implements UNIX pipe management and synchronization. It cooperates with the file managers to manage named pipes. Pipe management is no longer done by file managers as in "usual" UNIX kernels. Pipe managers may be active on every site, thus reducing network traffic when pipes are invoked on diskless stations.

6.4.5. Device Manager

Devices such as ttys and pseudo-ttys, bitmaps, tapes, network interfaces are managed by device managers. Needed (respectively not used) drivers can thus be loaded (respectively unloaded) dynamically (i.e., while the system is running). Software configurations can be adjusted to accommodate the use of the system or the local hardware configuration. For example, a bitmap driver might be present on a diskless station; a file manager might not be.

A CHORUS message-based facility is used by these drivers to cooperate with the UNIX servers instead of the original UNIX *cdevsw* mechanism. When starting up, these drivers tell the file manager which type of devices (i.e., which *major* number) they manage and the name of the port on which they want to receive *open* requests.

6.4.6. User Defined Servers

The homogeneity of server interfaces provided by inter-process communication allows system users to develop new servers and to integrate them into the system as user actors. One of the main benefits of this architecture is to provide a powerful and convenient platform for the experimentation of system servers: new file management strategies, fault-tolerant servers, etc., can be developed and tested just like user level utilities, without disturbing a running system.

7. Concluding Remarks

The paper illustrated the basic architecture of the GENESIS distributed operating system. This operating system will be a symbiosis of PEACE and CHORUS, following the design principle of a family of operating systems.

The purpose of this section is to point out some basic requirements which will support both the operating system and the application system. These requirements address software as well as hardware functionalities.

7.1. Configuration and Interface Language

Allocation and deallocation of nodes is not only an important function to distribute user teams after the system has been bootstrapped, i.e., after PEACE has been loaded. PEACE itself consists of several system teams being distributed over the GENESIS machine, and the same holds for CHORUS. Thus, there is a requirement to perform fundamental resource management functions already at bootstrapping time.

The best way to express the initial utilization of a GENESIS machine is by the means of a dedicated *configuration language*. The bootstrapping as well as loading activity can be controlled by proper high-level language constructs. Most importantly, resource management is *application-oriented*. Programs that are going to be distributed over the machine need not be changed, because the configuration language is not intermixed with the original programming language. Consequently, already existing packages of parallel and distributed programs can be handled without any changes.

For these reasons, a configuration language should be provided for the description of the actual structure of the operating system. This language should be based on *object-oriented* principles and shall give the system programmer the opportunity to express program distribution in an application-oriented way. As outlined above, PEACE will be constituted by a multitude of system teams. These teams represent the objects referred to by the configuration specification. Because distributed application programs are implemented by a multitude of teams, mapping of applications onto the GENESIS machine can be supported, too.

In GENESIS, system services are invoked on a message-passing basis. Consequently, the system call itself including arguments and results must be encoded in a message. In addition to that, system service invocations are synchronous, corresponding to procedure calls. In a distributed computing environment as introduced with GENESIS, *remote procedure calls* are necessary for the invocation of system services. In order to hide marshalling of arguments and results, a *stub generation* utility is useful, enabling the automatic creation of code sequences needed to encode and decode system messages. Using an *interface language*, the service functions provided by a system team can be specified and the stubs can be automatically generated.

In order to load application-oriented PEACE configurations, interpretation of a configuration specification as well as interface specification of PEACE will be handled by utilities being executed by CHORUS. A basic PEACE system then executes dedicated system teams that manage the requested configuration and reconfiguration of the GENESIS core system.

7.2. Hardware Functionality

Several user requirements can only be realized in case of dedicated hardware support. It is important to point out, that the following requirements are by no means necessary to build up a GENESIS operating system. The requirements are:

- a CP which is capable to perform dedicated software functions such as routing, monitoring, message accounting, setting up DMAs, etc. Consequently, the CP must be a processor which is controlled by dedicated software packages.
- a flexible and easy to use CP interface which is physically represented by a small set of device registers. The CP, i.e., the software package being executed by the CP, should be capable of interrupting the node, indicating message transmission, message acceptance or some error condition. These CP interrupts should be maskable by the kernel.
- selective node reset should be supported at the lowest possible level of the communication system
- a paged memory management unit should be available on each node, in order to implement team/actor address space isolation, virtual shared memory, distributed shared memory, distributed file caching and paging
- the node should support context switching in case of interrupts, traps and CPU multiplexing

It is one of the major requirements to implement the GENESIS distributed operating system without being dependent on the availability of the listed hardware functionalities. Nevertheless, most of the operating system tasks will be performed faster if these functionalities are available. This will have direct impact on the overall application program performance.

References

[Behr et al. 1988]

P. M. Behr, F. Schön, W. Schröder: **The PEACE Message-Passing Kernel Family**, Technical Note, GMD FIRST, 1988

[Cheriton 1979]

D. R. Cheriton: **Multi-Process Structuring and the Thoth Operating System**, Dissertation, University of Waterloo, UBC Technical Report 79-5, 1979

[Giloi 1988]

W. K. Giloi: **The SUPRENUM Architecture**, CONPAR 88, Manchester, UK., 12th-16th September, 1988

[Giloi 1989]

W. K. Giloi: **GENESIS: The Architecture and its Rationale**, ESPRIT Project No. 2447, Draft Report, GMD FIRST, Berlin, FRG, 1989

[Giloi, Schroeder 1989]

W. K. Giloi, W. Schröder-Preikschat: **Very High-Speed Communication in Large MIMD Supercomputers**, ICS '89, Crete, Greece, June 5–9, 1989

[Mierendorff 1989]

H. Mierendorff: **Bounds on the Startup Time for the GENESIS Node**, ESPRIT Project No. 2447, Internal Paper, GMD-F2.G1, 1989

[Parnas, Siewiorek 1972]

D. L. Parnas, D. P. Siewiorek: **Use of the Concept of Transparency in the Design of Hierarchically Structured Systems**, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA. 15213, 1972

[Rozier et al. 1988]

M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Leonard, W. Neuhauser: **CHORUS Distributed Operating Systems**, Computing Systems Journal, Vol. 1, No. 4, University of California Press & Usenix Association, also as Technical Report CS/TR-88-7.9, Chorus systemes, Paris, 1988

[Schroeder 1988]

W. Schröder: **PEACE: A Distributed Operating System for an MIMD Message-Passing Architecture**, Third International Conference on Supercomputing, Boston, MA, May 15-20, 1988

[Thole 1989]

C. Thole: **Functional Requirements for Process Management and Communication from a user's point of view**, ESPRIT Project No. 2447, Draft Paper, SUPRENUM GmbH, Bonn, FRG, 1989