Concepts of a Distributed Process Execution and Communication Environment

(PEACE)

W. Schröder

Gesellschaft für Mathematik und Datenverarbeitung mbH
GMD FIRST an der TU Berlin
Hardenbergplatz 2
1000 Berlin 12

ABSTRACT

The software architecture of the SUPRENUM operating system is explained refering to the design decisions, met. It is shown, what techniques are used to realize decentralized/distributed and fault-tolerant applications on top of SUPRENUM. As a special case, the operating system itself is regarded to be such an application. The fundamental techniques by which the system is constructed are consequent process structuring and very efficient mechanisms for inter-process communication using message-passing. The functional hierarchy of the operating system is given and the responsibility of each system component is explained, in short. It is exemplified how one can imagine to build up a decentralized/distributed PEACE operating system and to achieve network transparent inter-process communication as well as addressing.

* This work is supported by the Ministry of Research and Technology (BMFT) of the German Federal Government under Grant No. ITR 8502 A 2.

Table of Contents

Chapter 1

Introduction

The PEACE operating system is designed to use SUPRENUM [Behr et al. 1986] as a versatile virtual machine. It is this virtual machine that hides the fact of a complex multi-computer system from the application level. On the other hand, the network architecture of such a system is made visible to the application level to take advantage of the specific hardware architecture. Thus network transparency is achieved as well as dedicated and network oriented applications are supported.

In this paper the fundamental concepts and the functional hierarchy of PEACE are explained. Techniques are illustrated that allow for the construction of a high flexible operating system as well as application systems.

1.1. Fundamental Concepts

The design of PEACE has deeply been influenced by THOTH [Cheriton 1979]. Process structuring is the key to realize the PEACE operating system and message-passing is used as the fundamental inter-process communication mechanism. The concepts of V [Cheriton 1984] are the prototypes for the design of a decentralized/distributed and runtime efficient operating system. This is also valid for the design of applications intended to run on top of PEACE. The fundamental ideas about how one can hierarchically structure an operating system are taken from [Parnas 1975] and [Habermann et al. 1976].

With MOOSE [Schroeder 1986] it has been exemplified how to can construct a hierachically and process structured system that can be well suited to any application complex and which is based on message-passing. The basic mechanisms of MOOSE, especially the idea of consequent service structuring and service encapsulation by processes, are used to allow the dynamical reconfiguration of PEACE. These facilities are fundamental to the distribution and fault-tolerance of applications running on top of PEACE and of the PEACE operating system itself.

1.2. The Degree of Distribution

Due to the SUPRENUM hardware architecture at least a decentralized operating system must be realized. It suffices to provide mechanisms for remote service invocations in a fashion of remote procedure calls [Nelson 1982]. With respect to system services, there is no cogent necessity for a really distributed PEACE, e.g. by replication of primary data structures of the operating system.

System services, e.g. the creation or destruction of processes, are provided to the application level by dedicated system processes. These services are often composed from a set of internal and more basic services, which are itself provided by other system processes. The processes necessary to realize a system service, requested from the application level, need not reside on the same processor. It is the characteristic feature of PEACE that a system service is provided by a specific group of system processes and that these processes may be distributed over a given processor network. A system service may be provided in a distributed fashion if more than one system process is associated with the realization of the corresponding service request.

Because of its process structuring, decentralizing or even distributing PEACE is a natural consequence once the proper hardware architecture is given. With this respect, PEACE is a more decentralized/distributed system than e.g. LOCUS [Popek et al. 1981] is. In contrast to LOCUS, the typical operating system kernel is composed from a specific set of system processes. These processes, and thus the kernel, may be distributed over the processor network. There is no need in PEACE to condition each processor in a network with the same kernel functions, as it is e.g. the case with LOCUS.

The ability of a distributed realization of system service is given by the hierarchical process structuring of PEACE. Each system process provides some services to processes which are at a higher level in the system hierarchy. System processes at a higher level request services from lower level processes and may in turn provide specific services to higher level (system/user) processes. There is a strong uses relation [Parnas 1974] between processes and the interactions between processes are realized by remote procedure calls using the message-passing facility of the PEACE *nucleus*. The remote procedure call model of PEACE hides the multi-computer architecture of SUPRENUM. Each process complex that is responsible for some system service may be associated with a dedicated processor (node). Passing the node boundaries, because of a system or user initiated service request, is transparent to the processes.

1.3. The Degree of Fault-Tolerance

The main aspect to achieve fault-tolerance in PEACE is based on transaction oriented checkpointing and on the automatic generation of test sequences for the hardware as well as for the software components of the system [Fabry 1973]. The software related aspects of fault-tolerance will address both system and application processes.

The fault-tolerance of the system is defined by system processes which realize specific functionalities (e.g. checkpointing and recovery). The *nucleus* and very few low-level system processes of PEACE are the only system components which must reside on each SUPRENUM node. The functionality of these components is basic inter-process communication and the low-level aspects of naming, process creation/destruction and address space management. More complex functionalities [1] are realized by high-level system processes, which may be located only on a specific subset of all available nodes.

[1] e.g. the management, control and loading of jobs, deadlock detection and resolving, scheduling, propagating and handling of system specific exceptions, file handling, i/o management, etc.

The result of the consequent process structuring of PEACE is the very small size of system components. Additionally, each system component is given its own address space, which in turn is controlled by a specific set of processes. With that design decision a high secure system can be realized, because potential malfunctions of system components, e.g. the destruction of address spaces, are mostly of local nature. More specifically, the design of PEACE follows the idea of a security kernel, thus, once a formal specification is given, providing the chance of verification of the most crucial operating system components [Millen 1976].

The most significant aspect in PEACE to achieve a good basis for fault-tolerant system design is its consequent and actual structuring. As noted in [Randell et al. 1978], the important characteristic feature of an "actual" structure is the constrained inter-relationship between the system components. In PEACE the inter-relationship between system components is realized by synchronous message-passing and system components are encapsulated by processes. Replacement as well as reconfiguration strategies may be applicated to avoid, after its detection, the further use of a faulty system component. These strategies are supported due to the process oriented nature of PEACE, i.e. a faulty process (encapsulating a faulty system component) may be replaced by another ("stand-by") process or may be destroyed and re-created. These functionalities are at the system as well as the application processes' disposal.

One of the characteristic feature in the design of PEACE is to postpone design decisions as far as possible [Habermann et al. 1976]. With that means, design decisions for an actual model of fault-tolerance, e.g. statical/masking or dynamical redundancy of system components, are not included in the basic PEACE system structure. On the other hand, no design decisions had been met which would exclude any of the known models of fault-tolerance. It is the freedom of the system designer to realize specific and application oriented fault-tolerance models by one's own decision. That freedom is widely supported by the process structuring of PEACE as well as by the very basic services each system component (process) provides. One can say, on each level of system hierarchy fault-tolerance may be introduced, using specific system processes, without effecting the existing system processes.

1.4. Overview

In chapter 2 the basic components of PEACE are introduced. The functional hierarchy between these components is illustrated. It is shown in what way they are represented and what functionalities they provide.

In chapter 3 the more powerful and application oriented system components of PEACE are introduced. These components are termed "constructive components", because they are the building blocks of the operating system. Again, the functional hierarchy between them is illustrated and their relation to the basic componentes of PEACE is shown. The functionality of each constructive component is explained.

In chapter 4 a more enhanced model is introduced, which supports network transparency in PEACE. It is shown how one can imagine to decentralize or even distribute PEACE over SUPRENUM. The functionalities of specific system components for network communication are represented. Additionally, a short discussion about the functionality of certain distributed

constructive components takes place.

In chapter 5 some building blocks of PEACE are introduced in form of a short case study. It is shown how one can build up its own system services without reducing the functionality of the actual system. Each example gives a framework for representing in PEACE enhanced operating system services as accounting, monitoring, deadlock prevention, audit trailing and checkpointing.

Chapter 2

Basic Components

The basic system components of PEACE must reside on each SUPRENUM node. There are four system components, represented by three system processes and the *nucleus*. In this chapter these components and the functional hierarchy between them are illustrated, in short.

2.1. Functional Hierarchy

The basic system components of PEACE are the *nucleus*, the *name server* the *process server* and the *memory server*. The *nucleus* takes the position of a shared library, i.e. its services are invoked directly in a procedural fashion. There is no process switch performed if services of the *nucleus* are invoked and if results of the executed service are returned to the requesting process – although executing a *nucleus* service may involve a process switch, e.g. when applying certain inter-process communication primitives. The other named components each are represented by dedicated system processes[2]. Figure 2.1 gives the functional hierarchy of the basic system components.

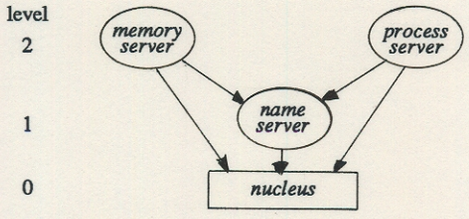

Figure 2.1: Basic system components

Common to all of these components is the address space switch upon requesting one of their services. The address space switch along with a service request to the *nucleus* normally is performed by hardware. A *nucleus* service is invoked by some trap instruction of the processor, thus switching from user to kernel mode for the requesting process. In each mode there is an own set of address space descriptors, as far as the processor supports this kind of address space management.

[2] In contrast to the *nucleus*, the service request to each of these processes, as well as returning results from them, means a process switch.

A system service is invoked by a remote procedure call to the corresponding system process. Moreover, the services of the *nucleus* are used to realize the remote procedure calls. These services are composed from the primitives for inter-process communication [Schroeder 1987]. The requested system service is coded in some kind of message which is transmitted by the *nucleus* on behalf of the requesting process, the client, to the providing process, the server. The address space switch in this situation is realized by the process switch along with the service request (the message transmission) and is performed by the *nucleus*.

The functional hierarchy shown is defined by the uses relation [Parnas 1974] between the single system components, encapsulated by system processes. This representational model is used throughout in this paper. Moreover, the uses relation between the system processes reflects the way the processes will block on each other upon requesting some lower level services. With this respect, the uses relation defines a blocking graph as applicated by [Cheriton 1979] for the design of THOTH.

2.2. The Nucleus

The *nucleus* takes the lowest level (level 0) in the overall system hierarchy. The main functionalities are inter-process communication, process dispatching, address space switching and trap/interrupt propagation. All these functionalities are invoked from the requesting processes by the trap sequence, supported by the used processor.

2.2.1. Inter-Process Communication

The mechanisms for inter-process communication are the basis for all system interactions in PEACE. Therefore, special requirements are worth to a qualified model and to a runtime efficient realization of the model. The fundamental concepts stem from THOTH.

Synchronous inter-process communication is supported by the primitives *send* and *receive*. Between the sending process, the client, and the receiving one, the server, a rendezvous is established, during which the transfer of data actually takes place. No structural requirements are stated about the message format, except its fixed size length. With the *reply* primitive the server normally terminates a rendezvous to a specific client. The *relay* primitive, applicated by the server, initiates a new rendezvous for the client. Addressing is performed by the process identification of the communicating processes.

The essential design decision taken with the inter-process communication model is that all communication activities are of local nature. There is always a local server, which means, that the *nucleus* needs only to handle local rendezvous. Remote rendezvous, i.e. rendezvous that crosses node boundaries, are handled by dedicated system processes. The multi-computer architecture is completely transparent to the *nucleus*, more specifically, it is transparent to all basic components of PEACE.

The fact that the basic inter-process communication mechanism deals only with local communication is the main requirement to realize very efficient inter-process communication. On the other hand, this model makes it feasible that higher level (global) communication

protocols are exchangeable without modification of any of the basic components of PEACE. Thus the dynamical reconfiguration of communication systems is given.

The independence from any communication protocol for inter-node inter-process communicationi, or more generally the ability of dynamical reconfiguration at runtime, is given by the routing facility of the PEACE *nucleus*. The techniques for routing were taken from MOOSE and have been improved for PEACE. In figure 2.2 the principles of routing on the very low level of PEACE are illustrated.

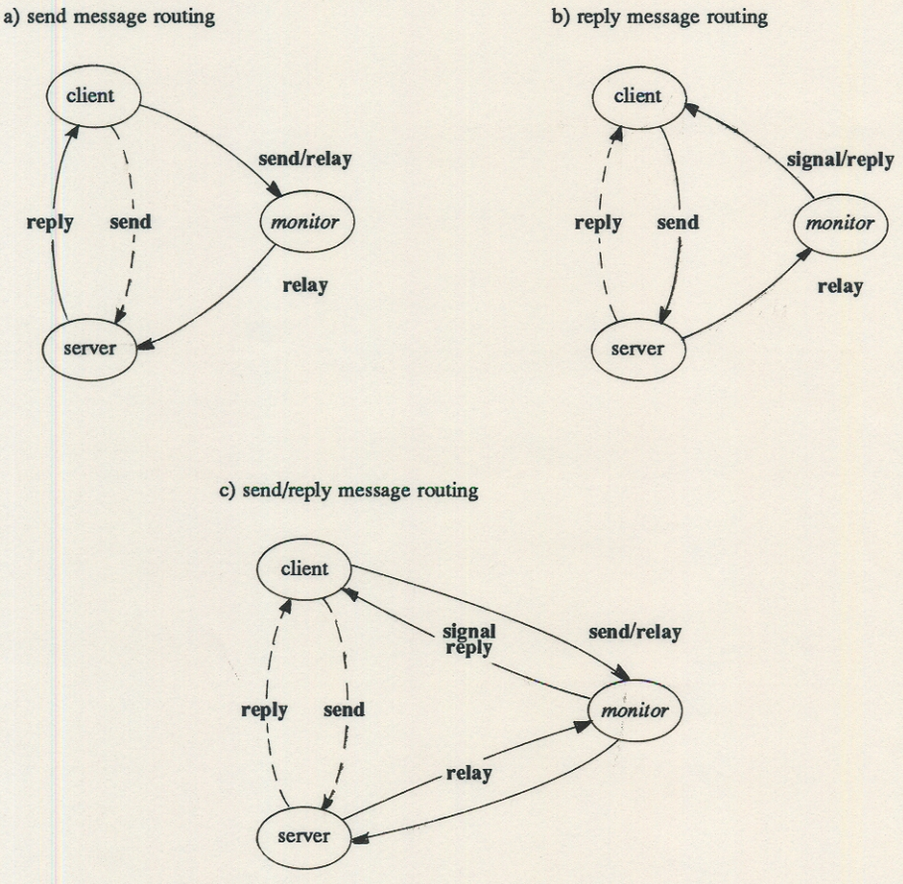

Figure 2.2: Monitoring send messages and reply messages

Routing inter-process communication is based on a technique that allows the adoption of one or both processes involved in a rendezvous. The send as well as the reply message can be routed to any other system process which in turn acts as a server, generally termed as the *monitor*. This process determines how to handle the received (routed) message. The routing process is able to terminate the rendezvous to the adopted client or relays the received message to the original server, by one's own decision.

Common to all of these variants is that a *send* or *relay*, which addresses a server as the receiving process, and a *reply*, which addresses the client as the receiving process, each may be routed to the *monitor*. The *monitor* normally stays in a *receive*, thus, due to the routed

message, always is able to start a rendezvous with the client. Note that the server never is the rendezvous partner of the *monitor*, because the *reply* to the client actually is relayed and not sent. This means, that the server, without taken any notice of it, initiates for the client a new rendezvous, this time with the *monitor*.

The functionalities of the adopting/routing server, i.e. the *monitor*, can be manifold. Not only high level communication protocols are realized in this way. Process migration, monitoring, accounting and verification of communication activities between client and server are realized/supported in PEACE using the basic routing facility of the *nucleus*.

2.2.2. Process Dispatching

Process structuring an operating system leads to more runtime overhead when providing a requested system service. The mechanisms for inter-process communication are tuned to reduce the overhead involved with the message exchange. Because process dispatching takes place with each communication activity it must be very well tuned, too.

Another consequence of process structuring is the high degree of sequential execution of system services as usually not discovered with procedure oriented systems, as e.g. UNIX [3] [Thompson, Ritchie 1974]. This is due to the typical sequential execution of one process and that a process in PEACE is the focus of service realization.

The sequential execution of system services is loosen by the concept of light-weigthed processes [Schroeder 1987b], as introduced with THOTH and also found in other modern message-passing operating systems as e.g. [Haertig et al. 1986] and [Balter et al. 1986] [4]. A group of processes is given the same access right on system specific objects. Such a group is called a team in PEACE. The objects are typically memory segments, files, devices, but also processes itself.

The *nucleus* regards processes as objects with which certain access rights are associated during a rendezvous. Not only the single server, but the entire server team receives reply, relay and data transfer access on the blocked client. The data transfer access right is related to the entire team of the corresponding client. By this way, the realization of a system service can be easily distributed on all processes of the same team. Thus, the sequential execution of a system service in general does not hold for a PEACE team.

Process dispatching is initiated either each time a process blocks, because waiting on a rendezvous (i.e. executing either *send* or *receive*), or on an explicitly request without blocking. To speed up process switching time, there is no expensive schedule strategy associated with dispatching. The processes are dispatched in a simple two-stage *round robin* fashion. The first stage dispatches processes local to each team. At this level, the processes are contained in a per-team ready list. The second stage dispatches processes from other teams. At this level, teams, that encapsulate processes ready to run, are contained in a so called dispatch set.

[3] UNIX is a registered trademark of AT&T Bell Laboratories.

[4] See also [Tanenbaum, van Renesse 1985] for a short overview about one of the most important and state of the art message-passing based operating systems.

Additionally, each team has its own time slice, which defines the maximal duration a team can execute without blocking. Blocking a team means, that all encapsulated processes are blocked, i.e. the per-team ready list is empty.

Dispatching a team is initiated either because the time slice elapsed, or because blocking the actual executing team. Independent of the reason for team dispatching, a process from another team is selected for execution. This process always is at the head of its per-team ready list, taken from the successor team contained in the dispatch set. Upon resuming the selected process, a new team is activated, defining its own execution domain for a new group of processes.

More enhanced strategies are realized by specific system processes, running on top of the *nucleus*. As exemplified with MOOSE, priority based or job related scheduling can be dynamically integrated in PEACE. Moreover, the scheduling strategies are exchangeable at runtime without effecting the basic components of PEACE.

2.2.3. Address Space Switching

One of the main aspects of teams in PEACE is the implied sharing of code and data segments between the encapsulated processes. As already mentioned, memory segments are objects to which all processes of the same team gain the same access rights. This means that all processes of one team are able to manipulate a common data area in a direct way. Once received a message, e.g., all processes of the server team may access and decode the message – all processes encapsulated by a team may execute the same code sequences.

This implied sharing is the main aspect why distributed service realization by a team involves no expensive runtime overhead. The same address space is multiplexed between the processes on one's own decision. Multiplexing is realized by relinquishing control over the address space of the team either by blocking, because applying specific communication primitives, or by an explicit dispatch request, in which case the requesting process remains ready.

Each process has its own runtime context, defined by its runtime stack. It is completely defined by the services of the *memory server* if a stack is only accessible from the associated process or if it is accessible by all processes of a team.

The *nucleus* takes advantage of the concept of teams if process switching and address space switching must be performed. A complete address space switch takes only place if a process of another team than the actual executing one is resumed. As far as there are more processes in a team ready to run, no team switch and thus no complete address space switch will happen.

The common address space defined by teams means that all processes belonging to the same team are located on the same processor in a multi-computer system. Swapping is defined only for an entire team and migration of processes means moving the entire encapsulating team to another processor. Thus, a PEACE team is never distributed.

2.2.4. Trap/Interrupt Propagation

One of the critical aspects in operating systems design is the management of traps and interrupts. Traps must often be handled in a concrete application dependent way. Interrupts are of asynchronous nature and their processing must be proper synchronized. Moreover, they are always related to some kind of devices, whose management is an awkward task in systems programming. Because of these facts, trap/interrupt handling in PEACE is widely encapsulated by specific teams and thus (at least conceptually) removed from the *nucleus*. The main task of the *nucleus* is merely to propagate traps and interrupts as specific events to the serving system processes and to synchronize the communication activities occuring in conjunction with the interrupt propagation.

Common to both kinds of event propagation is a special *nucleus* object, the gate. A gate controlls the activities of the *nucleus* along with the occuring event. Upon request by system processes, a gate is placed between the trap/interrupt vector of the processor and the serving process, the so called event server. The event server determines if a specific processor exception vector is to be served as a trap or as an interrupt. In case of a trap, the event server is more specifically termed as trap server; in case of an interrupt, the more specifically term is interrupt server. Additionally, a second process, the so called interrupt client, is involved with the interrupt propagation. Both processes belong to the same team, that in turn is responsible for processing propagated interrupts. Figure 2.3 depicts the general model for trap/interrupt propagation as well as the inter-relationship between trap/interrupt client and server.

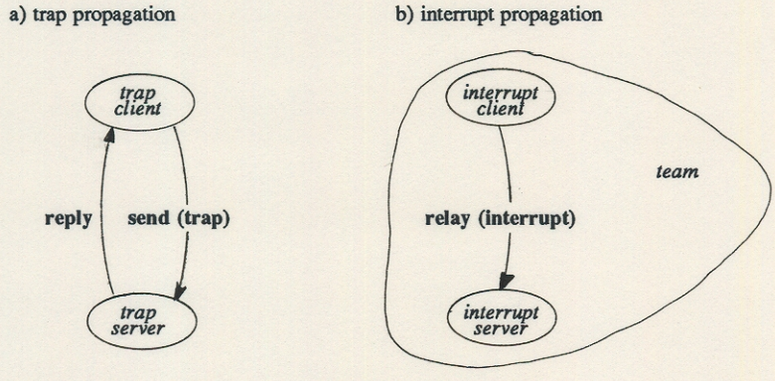

Figure 2.3: Trap/Interrupt propagation

The propagation of traps is a very simple task once the trap server is known by the *nucleus*. The trapped process, the trap client, is forced in a rendezvous with that trap server associated with the corresponding gate. The trap server team gains complete access on the trap client team – due to a trap, the trap client team uses the trap server team. By terminating the rendezvous with the trap client, execution of the trapped process is resumed. Because the data transfer access right onto the trap client team, the trap server team is able to manipulate the trap client's processor state. By this way, specific trap handling can be perceived by the trap server team. For example, process tracing can be controlled by dedicated system processes as well as serving of page faults, which may need the software enabled re-execution of the trap

initiating instruction by the trap client.

In contrast to trap propagation, the propagation of interrupts is a more complicated task. The main reason is given by the asynchronous nature of interrupts, whereas traps are always synchronous with the execution of a process. Interrupt propagation means relaying a message from the interrupt client to the corresponding interrupt server. This actually is a non-blocking communication activity. This communication activity is synchronized with respect to all other communication activities of the *nucleus*. The synchronization is controlled by the specific gate. Interrupt propagation is initiated after termination of the interrupt handler linked to the gate. The result of an interrupt handler specifies either to propagate or to ignore the interrupt.

Linking an interrupt handler to a gate may be performed in several ways [Schroeder 1986]. Basically, it can either be done in a dynamical way or it is statically established at *nucleus* generation time. At present, a PEACE interrupt handler is statical and procedural linked, thus executing completely in *nucleus* space. More enhanced strategies are planed to be realized, which allows the complete encapsulation of an interrupt handler by the interrupt client, thus giving the interrupt handler an own address space, the so called team space. This technique, as has been successfully realized with MOOSE, leads to the consequent decoupling of interrupt server space from *nucleus* space.

The dynamical reconfiguration of PEACE is praticable with a consequence that enables exchanging event server teams. From the security point of view, with each such a team there is an own address space associated. Thus, the system wide expansion of address space errors, because the potential malfunction of any of the system components responsible for trap/interrupt handling, can be reduced. With the proper hardware support, around each such system component a "fire wall" may be placed if teams are used consequently in one's system design.

2.3. The Name Server

The lowest level (level 1) of the hierarchy of system processes in PEACE is defined by the *name server*. The functionality of this process is to realize an abstraction from the service providing processes. Addressing a process for inter-process communication is based on the process identifications of the communicating processes. Once a process identification of a server is known by a client, the communication can take place applying the basic *nucleus* primitives. For example, this identification already may be made known to the client if a server creation request is stated. The result of a process creation service request usually gives the process identification of the new process. A more flexible solution is that of naming a server and addressing him by applying its name. A name is typically any character sequence defined by a server and applied by a client.

In PEACE the service a server provides is named and not only a single name is given (in an unstructured way) to the entire server. A server associates with each services he provides a certain service name and makes this name known to the *name server*. A client requests the identification of a server by asking the *name server* for a certain service name. The *name server* returns either an identification, that gives the way to the requested service, or indicates, that the service is not yet known. The identification is considered to be the service access

point of the requested service. If the service is provided local, the identification directly addresses the corresponding server. Is the service provided remote, the identification addresses some local network process, who realizes the remote service invocation. In either case the service requesting process is not aware about the different ways a service can be invoked. See [Schroeder 1987c] for more detail.

Basically, before the communication between client and server takes place, there is some kind of communication establishment to the server controlled by the *name server*. This phase is defined by asking the *name server* for the service name. The service connection may be released by the responsibility of the client or the server. The latter case is signaled as a PEACE specific system exception and may be propagated to all connected clients.

The essential facility of naming in PEACE is the ability of service replugging. The association between server and client can be manipulated in a way transparently to the connected clients. A system process can adopt a service and will in turn act as the new service provider. This mechanism is similar to that of adopting a process by applying the routing facility of the *nucleus*. The difference is that adopting a server means adopting all services the server provides. Adopting a service means that only a part of the servers communication activities is routed to the new server.

2.4. The Process Server

To keep the *nucleus* small in size and functionality no aspects of dynamic process creation/destruction are associated with that PEACE component. The *nucleus* regards processes or teams as statical known objects, although they are dispatched in a dynamical way. The dynamical view of processes and teams is realized by the *process server*, placed on level 2 of the system hierarchy.

Upon request from higher level system processes a new process may be created. Additionally, the new process is associated with a specific team. Thus its execution domain is manifested. The creation of a new team means the creation of a process which is the only one associated with the respective team. This process is called the *custodian* of the team. New processes may be added to an existing *custodian*. In this way a team actually is formed, because its characteristic feature is the group of processes belonging to the same execution domain. An added process is termed the *captive* of a team.

Destructing processes is analogous. Processes, more specifically *captives*, may be removed from an existing team. Removing the *custodian* means the complete destruction of the entire team controlled by this process.

Besides the creation/destruction of processes, the *process server* attributes processes as well as teams and creates system wide unique process identifications. The attributes control dispatching of processes and define certain access rights, global to all processes of the same team. Table 2.1 gives an overview about the team attributes recognized by the basic system components. With table 2.2 a summary about the meaning of process attributes in PEACE is given.

Team Attributes	
Attribute	Meaning
user	The team is allocated with normal user mode privileges.
system	The team is allocated with system mode privileges. The adoption/routing of processes is allowed.
privileged	The team is privileged. Connecting to trap/interrupt vectors is allowed, thus adapting traps/interrupts.

Table 2.1: The meaning of team attributes

Process Attributes	
Attribute	Meaning
custodian	The process is the initial one created in its team.
captive	The process is added to the team.
indivisible	The process is not dispatched if the time slice of its team elapses.
server	The process acts as a trap/interrupt server and thus can't be destroyed.
client	The process acts as a trap/interrupt client and thus, in case of an interrupt client, can't be destroyed.
interrupt	The process encapsulates an interrupt handler and is activated upon each signalled hardware interrupt at its corresponding interrupt vector. Additionally, the process must be attributed to be an interrupt client.

Table 2.2: The meaning of process attributes

A process identification represents a system wide unique object, designating a process. The uniqueness is given by the structure of the object – the concepts of V are recovered with a PEACE process identification. See table 2.3 for a summary about the meaning of the single structure members within a process identification object.

Basically, the *index* member of a process identification object is used to achieve efficient localization of control structures which are subject to process management[5]. Its application by

[5] A typical example is the *process control block* discovered in an operating system. PEACE specific, the *index* gives the *route map* [Schroeder 1987] entry, which contains the address of the process control

Process Identification	
Member	Meaning
index	Identifies the proper entry into several process tables.
version	Makes a process identification locally unique.
node	Gives the processor identification with which the process initially has been associated and makes a process identification globally unique. The cluster sub-field of this member allows for the identification of the cluster associated with the process.
class	Qualifies the process as remote or as system.

Table 2.3: Structure of a process identification object

the *nucleus* gives the basis for very fast inter-process communication primitives. The *version* member serves for the unique generation of a local *index* member associated with a new created process. Each time a process identification is fixed to a new process the *version* member is incremented by one. The *node* member enables the system wide uniqueness of the process identification. It identifies the processor initially associated with the new created process. Migrating the process onto another processor necessarily does not mean manipulating the *node* member of the corresponding process identification. The *class* member, more precisely the *remote* attribute, enables the *nucleus* to route inter-process communication with processes which reside on different nodes. Additionally to the *remote* attribute, there exists the *system* attribute as a further *class* member. This attribute says that the corresponding process is on a certain privilege level and is allowed to perform system specific operations. For example, in conjunction with the inter-process communication primitives, which all use process identifications for addressing server and client, a privileged client is recognized by inspecting the process identification returned from *receive* to the server.

It is important to note that the applicating processes of a process identification need not be aware of the concrete object structure. The process identification can be used as a plain, unstructured number for addressing a certain process [6]. This usually will be the case with simple application processes. However, system processes in PEACE will take advantage of the given object structure. By this way a runtime efficient realization of system services is supplied.

block, associated with the given process identification.

[6] In PEACE, a process identification object is 32-bit wide. This design decision leads to efficient application of process identification objects.

All tasks of the *process server* will effect certain data structures maintained by the *nucleus* to control team/process dispatching and inter-process communication. These data structures, the *process* and *team control blocks*, are collected together in proper tables located in *nucleus* space. The *process server* directly shares these data structures and is allowed to manipulate them by one's own decision.

2.5. The Memory Server

Just as no dynamical process creation/destruction is performed by the *nucleus* there is no dynamical address space management integrated in this system component. The *nucleus* merely performs address space switching each time a request for process dispatching is stated. From the *nucleus* point of view the address spaces are of statical nature. The dynamical view is given by the *memory server*, more precisely, it is located on level 2 of the PEACE system hierarchy.

Address space management means the creation/destruction as well as the expansion/reduction of address spaces. Creation/destruction of address spaces goes along with creation/destruction of processes. Against that, expansion/reduction of address spaces takes place during the processes lifetime. Memory segments are given the processes upon request and may be returned to the system, again.

The framework for an address space in PEACE is given by a team, or vice versa, a team owns a couple of code, data and stack segments, all together constituting its address space. Expanding the address space of a team is allowed by each process, encapsulated by the respective team.

The purpose of the *memory server* is to realize an abstraction from the given hardware architecture for address space management. Basically, the memory structuring mechanism is given by segments. For access control, with each segment certain attributes are associated. A segment descriptor maintains the physical location and size, the attributes and the ownership of the corresponding memory range.

A memory segment can explicitely be shared by a couple of teams – the segments constituting a team are implicitely shared by all processes of the team. Therefore, each sharing entity has its own access rights on the segment. Basically, the segment descriptor may be considered as the capability [Dennis, van Horn 1966] of a team for memory access. The capabilities on memory segments (objects) can be passed to other teams. This is typical in PEACE to support high-volume data transfer during a rendezvous [Schoen 1987].

Proceeding from the logical description of memory objects, based on segment descriptors, the *memory server* realizes a mapping onto the hardware. The mapping results in the definition of so called software prototypes which manifest the address space of a team/process. These software prototypes are known by the *nucleus*, because they are neccessary to fix the address space upon each process/team switch. More precisely, the *memory server* shares the software prototypes with the *nucleus*.

For teams, the *nucleus* maintains so called *image descriptors* which contain the software prototypes for the code and data segments accessible by the team. For processes, the *nucleus*

maintains so called *context descriptors* which contain the software prototypes for the stack segment of the process[7].

Address space switching, as performed by the *nucleus*, merely means copying the software prototypes, as defined by the *memory server*, onto the hardware prototypes for address space management. Usually, some kind of memory management unit (MMU) must be programmed according to the informations contained in the software prototypes. The *image* and *context descriptors* are structured in such a way that a simple and fast one-to-one mapping onto the hardware prototypes is sufficient to realize an address space switch – hence the term "software prototypes".

[7] Additionally, the stackpointer is remembered in the *context descriptor* of a process.

Chapter 3

Constructive Components

The constructive system components of PEACE all using the services provided by the basic components, introduced in the previous chapter. In contrast to the basic components, excepted some specific system processes, there is no cogent necessity to associate with each SUPRENUM node the same constructive components. The decentralization/distribution of PEACE is given by the actual placement of the constructive components on the nodes. In this chapter, the constructive components, the functional hierarchy between them and their relationship to the basic components are illustrated, in short.

3.1. Functional Hierarchy

Conceptually, the constructing components of PEACE completely are represented by dedicated system processes/teams which define level 3 upto level 9 of the system hierarchy. The distributeable system processes are the *team server*, the *signal server*, the *clock server*, the *tty server*, the *disk server*, the *file server* and the *loader* [8]. In contrast to these system processes, the *name replugger*, the *MMU server* and the *panic server* each serve for per-node and application oriented handling of specific system events. The functional hierarchy between these processes is given by figure 3.1.

The interactions between the constructive components are completely realized by the message passing facility of the *nucleus*. Moreover, each component exports services remembered by the *name server*. Therefore the uses relation to these both basic components is not completely shown – all constructive components of PEACE, more generally all system components placed on level 2 and above, will use *nucleus* and *name server* services.

In PEACE, the most significant aspect of the uses relation between the constructive components is its transparency to some network architecture. Remote service invocations are supported to realize the interactions between a decentralized set of constructive components. To achieve network transparency in PEACE and decentralization of PEACE additional system processes/teams are neccessary. These system components are associated with specific levels in the system hierarchy. However their functionality and their position in the system hierarchy is explained in the following chapter to keep the description of the minimal set of constructive

[8] The *clock server*, the *tty server* and the *disk server* each are representatives of PEACE interrupt servers. For clearness, the corresponding *clock client*, *tty client* and *disk client* has been removed from the figure. Actually, these system processes are mandatory for PEACE due to the *nucleus* interrupt propagation model. Without the presence of these processes, the corresponding servers never would receive propagated interrupt signals from the respective interrupt handler.

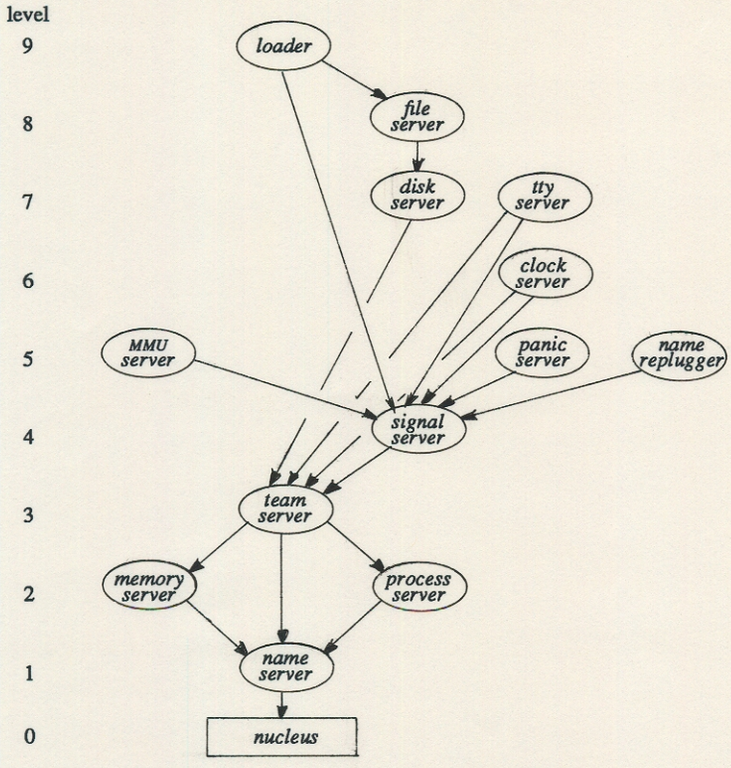

Figure 3.1: Constructive system components

components elementary.

Dependent on the functionality of the given hardware architecture, especially the used processor, parts of the components responsible for interrupt management must be migrated into the *nucleus*. These parts are defined by the interrupt handling modules, procedurally and statically linked to their controlling gate in *nucleus* space. Nevertheless, the respective interrupt server team and therefore the main functionality of the corresponding system component is associated with some higher level system hierarchy.

The given structure of the constructive components shows three specific system teams which may be regarded as a "bracket" around other system processes. The necessity of these teams and of both system processes in each team is given by the uses relation between the various system components. It is a general principle in PEACE to realize inter-relationships, whereby two system components uses each other, by representing one system component as two sub-components and associating with each sub-component an own process. In [Parnas 1976] this technique is called sandwitching of system components, because they in turn can take advantage of the services provided by each other. This principle is mandatory in PEACE, because services are invoked using blocking inter-process communication mechanisms. Thus if sandwitching would not take place, there will be a potential deadlock situation defined by the system uses relation. The "sandwitch" technique generally ensures the construction of a top-down and circle-free blocking graph [Cheriton 1979].

The term administrator in PEACE defines a system complex composed out of a specific set of processes. There may be only one process, a team encapsulating a group of processes or a certain group of teams, each constituting an administrator. In case of structuring an administrator as a group of teams, conceptually its possible distribution is given. An administrator is responsible for a certain set of system services and represents a complete autonomous system complex in PEACE.

Actually, there are eigth administrators associated with the constructive components of PEACE. The name, team and address space administrator each takes advantage of the team concept because "sandwitching" of their system components. On the other hand, the clock, disk and tty administrator each uses the team concept due to the *nucleus* interrupt propagation model. These administrators are generally termed device administrator. The file administrator, by now only represented by a single process, will provide functionalities for remote file access. To realize the network transparency with respect to file management, this administrator, in a more enhanced version, will consist of a couple of teams. And finally, the signal administrator controlls the propagation of system specific events.

3.2. The Team Administrator

Teams are constituted using process and memory objects. Each process represents a certain team activity and the memory objects are represented by code, data and stack segments. The team administrator associates processes with segments and processes with teams. More specifically, the team administrator defines the process model as e.g. provided to the application level.

Process and team objects are requested from the *process server* and memory objects are requested from the *memory server*. Requesting a process object results in the delivery of a process identification, uniquely designating that object. The same is true when requesting a team object, in which case the process identification of the corresponding *custodian* is returned. The result of a request for some kind of memory object (representing a code, data or stack segment) is the corresponding segment identification of that object. The team administrator remembers for each team what process and memory objects have been allocated for team construction.

The creation of a new team, as controlled by the team administrator, is possible in several ways. Existing memory objects, not yet associated with any team, may be related to the new team object. Another variant is relating already associated memory objects to the new team. In this variant some memory objects are duplicated (in case of data and stack segments) and some other memory objects are shared (in case of code segments). Basically, the corresponding segments already have been allocated by the team administrator. The alternative to that is loading segments from the file system. Because of these two strategies, the team administrator is "sandwitched" and realized by two system processes.

The functionality of the *loader* is controlling team creation, which needs reading of segments from the file system. A UNIX *exec* or LOCUS *run* service is the typical example for that. Additionally, the *loader* controlls process termination and raises this event, as well as the process creation event, as a system specific exception. These events are propagated by the

signal server in sequenced multi-casting[9].

The *team server* controlls creation of teams and processes for which already loaded segments are available for setting up the corresponding runtime image. An example for that is the UNIX *fork/vfork* service. In addition to that, the expansion/reduction of address spaces is controlled by the *team server*. In this case new memory segments are associated with the given team and memory segments of a team are released.

3.3. The Signal Administrator

The solely task of the *signal server* is the propagation of specific exceptions. The exceptions are raised by certain processes and their handling is properly propagated to some higher level components. By this way typical mechanisms for application oriented exception handling [Goodenough 1975] are supported. The main aspect with exception handling in PEACE, however, is the application oriented propagation of system/user defineable events across address space as well as node boundaries.

3.3.1. Signal Propagation Model

Exceptions in PEACE are represented by expedited messages, so called signals, whose transmission and reception is guaranteed by the system. Because of that message oriented nature of signal propagation in PEACE, there is a specifc model necessary. This model stems from MOOSE and is shown in figure 3.2.

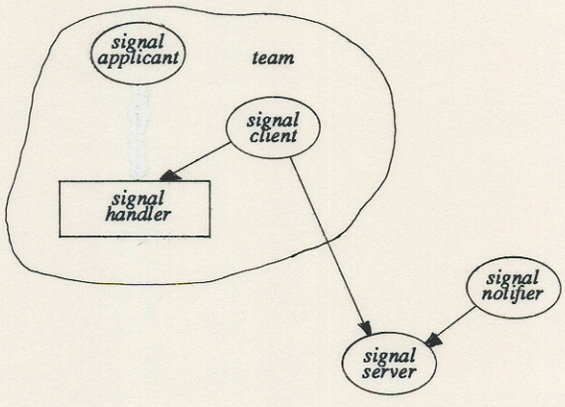

Figure 3.2: Propagation of signals

In general, there are four processes involved with signal propagation. On the raising side, the *signal notifier* raises or notifies the occurance of an exception. On the receiving side, the *signal client* serves and awaits propagated signal, originally dedicated to the *signal applicant*. In general, *signal applicant* and *signal client* both are encapsulated by the same team. On the

[9] Sequenced multi-casting is explained along with signal management in PEACE.

propagating side, the *signal server* directs and distributes raised signals to all clients which requested to be informed about the occurance of a specifc exception. The *signal server* and the *signal client* both constitute the signal administrator.

The *signal client* is blocked as long as there are no signals pending. This blocking is simply represented by a non-terminated rendezvous with the *signal server*. Thus, propagating a signal by the *signal server* means terminating the rendezvous to the corresponding *signal clients*.

The *signal notifier* is only blocked as long as the *signal server* has not yet processed (remembered) the request for signal propagation. From the *signal server* point of view the *signal notifier* executes asynchronous to the *signal applicant* and the *signal client*.

The *signal applicant* is the actual destination of a raised signal. The *signal notifier* usually only knows the *signal applicant*. The *signal client* is hidden to both processes, because its existence is required only by the specific signal propagation model of PEACE. Consequently, the *signal client* usually will be part of the system library, responsible for signal handling.

The existence of the *signal client* is motivated due to the fact, that signals must be propagateable to a specific application process (in this model represented by the *signal applicant*) independently of its actual processing state. Specifically this is ensured in PEACE by the non-terminated rendezvous with the *signal server*. Because the fact that signals are represented by messages and the *signal clients* will represent the receiving processes of a propagated signal, the distributed/decentralized realization of signal propagation is possible in PEACE.

3.3.2. Signal Distribution

The actual propagation of signals is realized by their distribution on the known *signal clients*, each one waiting on the signal. With each client, a set of signal numbers is associated, whereby each signal number is an application dependent identification of the propagated signal. Additionally, each client is given a team group, which is inherited from the *custodian* (the *signal applicant*) of the *signal client*. The group identification and the signal number are used to distribute raised signals, thus actually propagating the signals.

It is important to note that a single *signal client* may serve signals for a couple of *signal applicants*. More specifically, a hierarchy of *signal servers* may be constructed if a *signal client* is projected to act itself as a *signal server*. This is due to the team group relationship between these processes. Thus encapsulating the pair *signal applicant/client* by the same team is no cogent necessity, although it will be the usual case if propagating system exceptions to the application level.

The *signal server* remembers which *signal applicant* will accept and which *signal client* will handle the occurance of a specific signal. If a *signal applicant* does not accept the signal, its termination is initiated by requesting the proper *team server* service. If a *signal applicant* accepts the signal, however there is no *signal client* for handling it, the signal is ignored. If there is a *signal client* known to the *signal server*, the signal will be propagated properly. In

case that the *signal client* is already processing a propagated signal, the new signal is made pending. As soon as the *signal client* again is registered by the *signal server*, all pending signals are propagated to him. In this case the *signal server* immediatly terminates the rendezvous to the *signal client*.

If there are no signals pending, the rendezvous to the *signal client* itself is made pending, i.e. the *signal server* waits on *signal notifier* requests for signal propagation to the waiting client. The *signal client* implicitely is selected by the *signal server* due to the information given by the *signal notifier*. Either the process identification or the team group of the *signal applicant* is used for selection. Given a team group identification[10], all *signal applicants* with that group identification are considered and the corresponding *signal clients* are selected. In either case, the raised signal is propagated to each selected client. However, only those *signal clients* are selected for propagation whose *signal applicant* will accept and does not ignore the corresponding signal.

Additionally to the raised exception, the way propagating the corresponding signal can be defined by the *signal notifier*. In general, a signal is propagated to all waiting clients in parallel. There is no ordering and no dependence defined between the clients. The other way of signal propagation is based on some kind of sequencing the activation of the selected *signal clients*. In this case the signal is propagated on a client-by-client basis. At each point in time there is only one *signal client* processing a specific signal. A typical case of sequenced propagation is given with the process termination/creation signal.

3.3.3. Multi-Casting

Basically, signal propagation in PEACE means multi-casting an expedited message. Realized is multi-casting due to the ability for the *signal notifier* to supply a team group identification along with raising a signal. All *signal clients* belonging to the supplied team group may receive the same signal, respectively the same message.

Within the complex of system processes, multi-casting is a fundamental technique in PEACE to distribute system specific events. Such events, e.g., are termination/creation of processes and detaching services from server processes, as done in conjunction with process migration and service replugging just as with replacement and reconfiguration strategies for achieving fault-tolerance.

One of the significant applications of multi-casting in PEACE is checkpointing, consistence checking of specific data structures, test pattern generation and recovery. A fault-tolerant application in PEACE uses the *signal client* model to encapsulate application oriented management activities, triggered at the responsibility of dedicated system processes, e.g. a *checkpoint server*. For example, this system process controlls writing of application dependent checkpoints and directs the *signal client* to actually write out critical data objects of its team. The team itself is part of the application to be checkpointed. Is the application composed out

[10] A team group identification actually is represented by a process identification object whose *index* member is 0.

of a couple of teams, these teams each may given the same team group identification. Thus, initiating checkpointing would lead to reactivation of all the *signal clients* acting as *checkpoint clients* of the corresponding teams.

3.4. The Panic Server

Trap handling in PEACE exclusively is controlled by system processes. These processes, so called trap server, are connected to distinct processor trap vectors. The connection is managed by the *nucleus* and a trap is represented as a specific system message, transmitted to the connected trap server.

There may exist a couple of trap servers in PEACE – extremely, with each trap vector an own trap server can be associated. On the other hand, only a single trap server may be present, receiving all kinds of trap messages. In this case the trap server is connected to each trap vector of the processor.

The functionalities of trap servers can be manifold. For example, they may emulate certain processor instructions, serve page faults, handle address space errors or they may trace/debug specific processes. Basically, they accept traps, propagated by the *nucleus*, and they may itself direct the traps to be propagated as specific exceptions. In this way, a signal client may be notified about the accurance of a trap, e.g. initiated by its *custodian* (i.e. the signal applicant).

The trap message, as propagated by the *nucleus*, contains the vector identification of the corresponding trap and the processor state of the trapped process, the trap client. Because of the rendezvous between a trap client and a trap server, the processes of the trap server team are able to manipulate the processor state of the trap client. Handling of specific traps makes this necessary.

The position of a trap server in the PEACE system hierarchy is denpendent on its functionality. Presently, there is one system process in PEACE, whose responsibility is to signal all non-specific [11] traps as some system specific exception. This process is the *panic server* and its position is just above the *signal server*. The *MMU server*, e.g., is a trap server that handles MMU specific traps. Advanced handling of these traps may result in page loading from disk if a paging system is to be supported. In this case, the position of the corresponding trap server (*pager*) would be just above the system process responsible for disk management. Another example will be a trap server whose functionality is loading code segments upon the first reference on it. If properly arranged, this reference results in a trap and may be propagated to a corresponding trap server. In MULTICS [Organick 1972] such a functionality is used to realize dynamical binding of application programs. In this case the corresponding trap server will be placed in the system hierarchy above the *loader* to read in the referenced code segment. The concept of shared libraries under PEACE will use such a trap server semantic.

[11] With "non-specific" a trap is termed, with which no specific exception, e.g. divide-by-zero or illegal instruction, is associated by the operating system.

3.5. The Address Space Administrator

Address spaces in PEACE are controlled by a paging MMU. MMU traps may be generated and signalled by the processor if a process performs an exceptional access on MMU protected objects. However, this does not mean to abort the execution of the trapped process, rather the trap may be projected and thus is expected for the process. A typical example is the page fault in virtual memory systems. In contrast to that, violating the virtual or logical address space given to a process/team results in a non-expected MMU trap and may lead to the termination of the trapped process/team.

To distinguish an expected MMU trap from a non-expected one, the address space definition of the trapped process must be investigated. This definition is manifested in the software prototypes of a process or team. Thus, properly handling the address space trap means sharing knowledge about certain data structures with the *memory server*.

If the trap is not expected, an address space violation exception is to be signalled. To realize that, the *memory server* must use *signal server* services. From the hierarchical point of view the *memory server* is unable to use these services. More specifically it is not allowed for the *memory server* to do so, because violating the inter-relationship defined by the blocking graph may result in a system deadlock. Therefore, the original *memory server* is "sandwitched". The *MMU server* is the responsible process for accepting, validating and signalling address space traps. This process as well as the *memory server* both constitute the address space administrator.

The *MMU server* acts in a way as a general trap server does. A connection to the MMU trap vector is requested from the *nucleus*. This enables the *MMU server* to receive propagated traps. One can say the *MMU server* is a problem oriented trap server, handling address space traps. Problem oriented aspects are, e.g., realizing copy-on-write or paging functionalities, each triggered by the propagated trap. In PEACE, copy-on-write semantic of a page fault trap is intended for use in conjunction with address space creation [12] as well as high-volume data transfer, where actually pages are exchanged (i.e. re-mapped), only[13].

3.6. The Name Administrator

A significant functionality of PEACE is the propagation of a service re-association with some server. Generally known as service replugging, an existing service is associated with another service access point. The necessity for that functionality is given to support fault-tolerant applications as well as team/process migration, whereby the migrated team encapsulates at least one server process. The name administrator is responsible for supporting the service replug facility in PEACE.

Basically, the name administrator is realized by the *name server*, which in turn has already been introduced as one of the basic PEACE components. Adding the functionality of service replugging, however, needs the existence of some higher level system process. This process,

[12] in a similar way to the *vfork* system call of 4.2BSD [Joy et al. 1983].
[13] Paging, thus really supporting virtual memory, is for further study.

the *name replugger*, shares the same knowledge with the *name server* to be directly informed about the actual service-to-server relationship.

The *name replugger* raises the re-assignment of a service name to another server as a name replug exception. This signalling is the motivation for "sandwitching" the original *name server*, thus to allow the usage of *signal server* services by the name administrator.

In its more enhanced functionality, service replugging may ensure the re-creation of a terminated server. By this way a certain service never vanishes from the set of all system services. The process realizing that functionality, however, will serve the name replug exception, as propagated by the *signal server*, instead of burden the *name replugger* with the server re-creation semantics. This case is a typical example in PEACE of a postponed design decision.

3.7. The Clock Administrator

Low level clock management already is performed by the *nucleus*. However, the functionality of that *nucleus* facility is restricted to time slice control and to realtime clock counting. More enhanced strategies are realized by the clock administrator.

The clock administrator actually is represented by a team that encapsulates two processes. This organization is required because the interrupt propagation strategy, as defined by the *nucleus*. Generally, the clock administrator belongs to the class of device administrators in PEACE. Its services, however, may be used by other device administrators to realize e.g. timeout controlled device management. Additionally, the clock administrator enables time controlled process execution as well as time controlled signal propagation, e.g. the distribution of an alarm clock signal. Therefore, its position in the system hierarchy is different from that of other PEACE device administrators.

Process execution may be delayed upon request to the *clock server*. Delaying simply is realized by a non-terminated rendezvous with the requesting client. Resuming execution of a delayed process is done either if the specified delay time has been elapsed or if it is triggered upon request from some other process. Additionally to that functionality the actual time-of-day clock is maintained as well as alarm clock signals may be raised by the *clock server*.

The *clock client* actually waits on the propagation of clock interrupt events. More specifically, these events already have been scalled with a software implemented interval parameter, done by the *nucleus* clock interrupt handler. A clock event is directed to the *clock server* as a clock message whose producer is the *clock client*.

3.8. The Device Administrator

Level 7 of the PEACE system hierarchy is concerned with i/o management. On this level several interrupt server teams are located, each performing device specific functionalities. Therefore, the number of system processes associated with this level is highly configuration dependent.

At least two system teams are necessary to support typical device handling functionalities as discovered with operarting systems. The teams are the disk administrator, to serve block special devices, and the tty administrator, to serve character special devices[14]. With respect to the network architecture of PEACE an additional device administrator exists for serving frame special devices. Its functionality is explained in conjunction with the network transparency of PEACE.

As with the trap server, a device administrator is connected to a specific interrupt vector of the processor. More specifically and as required by the interrupt propagation model, there will exist certain interrupt server and interrupt client processes in the corresponding device handling team. If interrupt handling is performed in *nucleus* space, the respective team shares a data segment with the *nucleus* device handling modul. If interrupt handling is performed in team space, the corresponding device handling team completely represents an autonomous system component. In either case the interrupt propagation model is the same, i.e. the interrupt client will send a message to the interrupt server each time requested by the interrupt handler. Actually the interrupt client is relayed, thus propagating an interrupt is achieved using non-blocking inter-process communication.

The associated system hierarchy with device administrators in PEACE enables them to use certain system services. Most importantly, the interrupt server will create the interrupt client within its team before the linkage to a specific interrupt vector is established. Analogous, the interrupt client is destroyed if the connection is released. By this way dynamical reconfiguration of distinct device administrators is supported in PEACE.

The relation between the tty administrator and the *signal server* enables operator initiated propagation of signals. Specific control characters are associated with signals and stroking the proper key at the keyboard will initiate the propagation of the actual signal. The destination for such a signal propagation is the team group of the tty owner. This functionality allows, e.g., the termination of processes, running out of control by other processes.

3.9. The File Administrator

File handling in PEACE is related to UNIX file handling. The file structure implemented, using disk administrator services, basically is that of UNIX SYSTEM V. Responsible for file management in PEACE is the file administrator.

Because the file administrator is seperated from the disk administrator, it can be placed on each node in a network oriented system. Moreover, replication of files, as e.g. done with LOCUS, and thus supporting a distributed file system may be achieved.

A special functionality of the PEACE file administrator is locating certain device administrators upon request from higher level processes. As with UNIX, the concept of special device files is supported. Opening of such a file will direct the file administrator to ask the

[14] With respect to the SUPRENUM architecture, the disk administrator usually will reside on the *disk node* and the tty administrator will realize the character special interactions with some *host*, running UNIX on top of it.

name server for the device administrator of the corresponding device. The service name exported by the device administrator may be constructed simply by representing the UNIX major/minor device number as a character sequence.

The file opening request to the file administrator terminates with the process identification of the server for the just opened file. In case of a plain file the process identification designates a *file server*, belonging to the file administrator. Particularly, this *file server* may reside on any node in a network oriented system.

In case of opening a special device file, the delivered process identification may designate any device administrator. The opening request usually is repeated, this time, however directed to the new identified server.

It is a principle in PEACE that i/o requests by higher level processes are supplied with the process identification returned from the (evtl. repeated) file opening request. This identification actually represents the access point of the requested i/o service – PEACE and thus i/o services always are requested by inter-process communication via the corresponding service access point, implemented by some server process. Because of that abstraction, remote file/device access is supported in a very simple and runtime efficient fashion. The requesting process is not aware about the actual localization of a service provider. All remote files/devices are served by a problem-oriented *file/device deputy* [Schroeder 1987c], just close to the requesting client and, more specifically, residing on the client's node.

Chapter 4

Network Transparency

PEACE is a decentralized/distributed system in which its constructive components may be distributed over a given network architecture. To achieve a high degree of distribution within the operating system one important fact is to hide the network architecture from the constructive components. Only with this aspect the different components can be arbitrarily associated with any node in the network.

The required network transparency in PEACE is realized by a specific set of system components. These components widely follow the design principle in PEACE of a process structured operating system. Distinct system processes are used to realize the inter-node communication. Especially these processes support mechanisms for remote service invocation and actually realize the network transparency.

In this chapter the system processes necessary for achieving network transparency in PEACE are introduced. A basic model is illustrated which supports high-performance network communication and which provides a high degree of flexibility in system design. The actual position of the necessary communication processes in the system hierarchy is explained along with their inter-relationship to other PEACE system components. It is exemplified how a typical configuration of a decentralized/distributed operating system for SUPRENUM looks like.

4.1. The Network Architecture

SUPRENUM is a multi-computer system which is based on two different inter-connection systems. The one inter-connection system is given by the so called cluster bus which connects upto 20 nodes, thus forming a cluster. The cluster bus is a high-speed parallel bus and supports transmission speeds of upto 256 Mbytes/sec. The other inter-connection system is given by the SUPRENUM bus which connects a couple of clusters. The SUPRENUM bus is a slotted-ring bit-serial bus with a transmission rate of upto 25 Mbytes/sec. With this inter-connection system as the basis so called hyper-clusters are realized by connecting a couple of clusters. Thus, a cluster represents the SUPRENUM hardware building block to construct a specific multi-computer system with a performance to any extend desired by a customer. See [Behr et al. 1986] for more detail.

To allow the flexible inter-connection of clusters, in each cluster a qualified node is identified for that task. This node, the communication node, acts as a gateway, i.e. it bridges the two different SUPRENUM inter-connection systems. Additionally, specific communication nodes are connected to a host, running UNIX on top of it. There are three of such hosts identified by the overall SUPRENUM architecture, each one associated with some specific controlling task.

This specific network architecture requires the very efficient realization of a communication model with respect to maximal network performance as specified by the hardware. The first step in that direction already has been realized by the high-performance local inter-process communication mechanisms of PEACE. The comparison to other message-passing systems, particularly V [Cheriton, Zwaenepoel 1983], AMOEBA [Tanenbaum, van Renesse 1985] and AX [Schroeder 1986], shows that PEACE takes the leading position with respect to its message-passing performance. The remote communication model benefits from the runtime efficiency associated with the local inter-process communication mechanisms, from the team concept and from the efficient mechanisms for process dispatching.

4.2. Remote Service Invocation

All interactions between user and system processes in PEACE are controlled by a remote procedure call like fashion. Because services are the focus of system activities, the principles of a remote procedure call are applicated to realize a model that allows for a remote service invocation.

It is important to note that all service invocations in PEACE potentially are of remote nature. This is also true if local service activities between processes are considered, i.e. between processes residing on the same node. In this case, service invocations are remote with respect to its initiating and its providing team, its address space. The same remote service invocation protocol data unit is applicated, independently of the node or cluster membership of the corresponding processes.

The generation of the protocol data unit as well as "marshalling", i.e. the generation of code for the definition and interpretation of a protocol data unit, is done automatically. Given an interface specification of the requested service, a PEACE utility arranges the necessary steps for generating a framework to cope with the definition and interpretation of a protocol data unit for remote service invocation. In terms of [Nelson 1982] and applicated to the service model in PEACE, this framework is represented by the *service stub* on the client's side and by the *call stub* on the server's side[15]. Basically, the protocol data unit used in conjunction with remote service invocation is structured as illustrated by table 4.1.

One great advantage of the mechanism just mentioned is the resulting runtime efficiency in conjunction with the service invocation and acception sequence. A typed message directly is coded without any procedural or system specific overhead. One can say, the actual programming language, in which client and server are implemented, has been extended by a new data type. However, this data type is not visible to the programmer.

The principle of abstract data types strongly is followed with the service interface design in PEACE. Not only high-performance is achieved when invoking a service. More importantly is the fact of abstraction from the way a service is encoded by a message, the way the

[15] The interface specification is represented by a MODULA-2 definition module. With UNIX LEX and YACC a parser for MODULA-2 and a code generator for marshalling procedure (service) arguments has been built. Given any definition module, the library for service invocation (*service stub*) as well as for service acceptance (*call stub*) is generated automatically.

Remote Service Invocation	
Protocol Data Unit	
Member	Meaning
client	The process identification associated with the service requesting client.
team	The team identification associated with the service requesting client.
server	The process identification associated with the service providing server.
status	Indication about the successful/non-successful service execution.
service	The internal identification of the requested service.
data	The argument list associated with the service function.

Table 4.1: Structure of a remote service invocation protocol data unit

corresponding server is identified by asking the *name server*, the way the service is decoded from the received message and, finally, the way the client/server interaction is realized by a specific communication protocol and system. It is this mechanism of remote service invocation which enables the network transparency in PEACE. For example, service replug exceptions are served on this level, thus hiding the corresponding signal client from the application level. Handling this kind of system specific exceptions within the application context, always can guarantee the direct addressing of service providing processes.

4.3. Inter-Node Inter-Process Communication

In PEACE, inter-node communication is based on a model with minimal process overhead. Considering two nodes, remote service invocation and thus remote communication is modelled as given by Figure 4.1.

It is important to note that the actual underlying network hardware architecture does not influence the functionality associated with that model. Actually, the *node server* hides the structure of the respective network interface[16]. More specifically, the *node server* encapsulates the device handling components necessary for driving the physical network interface. A virtual network device realizes the abstraction from the specific physical characteristic of the networking hardware.

[16] The network interfaces in PEACE are given by the cluster bus and the SUPRENUM bus.

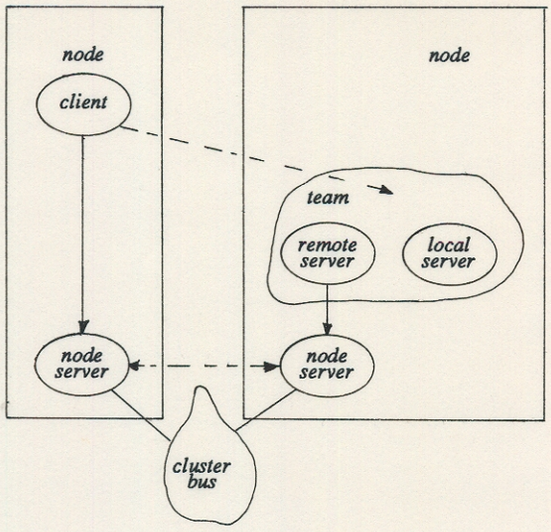

Figure 4.1: Communication between different nodes

4.3.1. Remote vs. Local Server

It is a principle in PEACE that a server, which provides remote accessible services, at least is constituted by two processes encapsulated by the same team. The one process, the *local server*, is responsible for receiving (by *receive*) and processing local service requests, whereas the other process, the *remote server*, is responsible for requesting (by *send*) from the *node server* remote service requests and, once delivered by the *node server*, controlls processing of these requests.

With that very simple model the *remote server* acts in the same way as the *local server* does – it blocks as long as there are no pending service requests. More importantly, the (client-to-server) inter-relationship between *remote server* and *node server* allows the continous (pre) processing of incoming service requests. Either the rendezvous to the blocked *remote server* is terminated by the non-blocking *reply* issued by the *node server*, or the incoming message is made pending (queued) and delivered upon one of the next rendezvous.

The *node server* only blocks on the actual network interface, thus waiting on more incoming messages. Because handling the network device, the *node server* belongs to the class of PEACE device administrators, in this case the node administrator[17].

Terminating the rendezvous to the *remote server* has the same effect as if sending a message to the *local server*, with except that the reply message contains the message-encoded remote service request and not the send message. In either case, the requested service (coded in the message) immediatly may be processed, because both processes reside in the same

[17] Due to the *nucleus* interrupt propagation model, a device administrator consists of an interrupt client and interrupt server, both encapsulated by the same team. As illustrated in one of the following sections, the *node server* actually combines the functionality of interrupt client and interrupt server. From the conceptual point of view, however, there is a seperate *node client* representing the actual network hardware interface.

server team. Moreover, there is no lost in efficiency while processing a service, because services are still received in a client-by-client basis without the necessity of any specific client/server synchronization. The *local server* only executes if the *remote server* does not, and vice versa. Each process will block if there are no more requests pending, thus, due to the dispatch strategy locally to one team and enforced by the *nucleus*, enabling the execution of the other, non-blocked process.

4.3.2. Invoking a Remote Service

To complete the scenario of inter-node inter-process communication in PEACE, the client side is considered in the following. In general, a client is unaware about the localization of the server. In case of a remote residing server, the service access point, at which the client may state certain service requests, is represented by the *node server* residing on the client's node. Sending a message-encoded service request to such a service access point, will restart the *node server*, which in turn will initiate the interaction with the corresponding *node server* residing on the service providers's node.

Distinguishing a remote service from a local one is done once the service provider has been asked for by supplying the *name server* with a service name. The process identification delivered by the *name server* designates the actual server for the requested service, independently of the server's node membership. In case of a remote residing server, its process identification is attributed as *remote* and the *node* member of this identification is different from that of the client's process identification. Because the *remote* class attribute or the *node* member associated with a PEACE process identification, one can efficiently distinguish a remote from a local residing server. The *remote server* process identification is returned to any client requesting a service from remote. In contrast to that, the *local server* process identification is returned to any client locally requesting a service. However, it is not required that a service completely is provided on the *local server's* node. The local residing server may be the *deputy* of a remote residing server, thus allowing for a service specific, i.e. problem-oriented, inter-node protocol.

In case of a remote residing server, the message-encoded service request is transmitted using *node server* services. Actually the remote service invocation protocol data unit is sent to the *node server*, which in turn starts the necessary inter-node communication with some other *node server*. The node address of the remote residing *node server* is taken from the *node* member of the *remote server* process identification. This process identification is taken from the protocol data unit, more specifically from the *server* member, used along with a remote service invocation.

4.4. Inter-Cluster Inter-Process Communication

A principle problem arises with remote inter-process communication if network boundaries must be passed. In this case, typically, a *network gateway* must be installed, whose responsibility it is to shift all communication activities between different network architectures. This means an additional overhead, because converting protocol data units as well as

programming other network devices will be necessary. Moreover, it may be required to drive another protocol and to perform some other time consuming management activities as, for example, buffering of data items. Actually with that situation the overall communication system for SUPRENUM is confronted. Figure 4.2 shows in general the inter-networking model in PEACE.

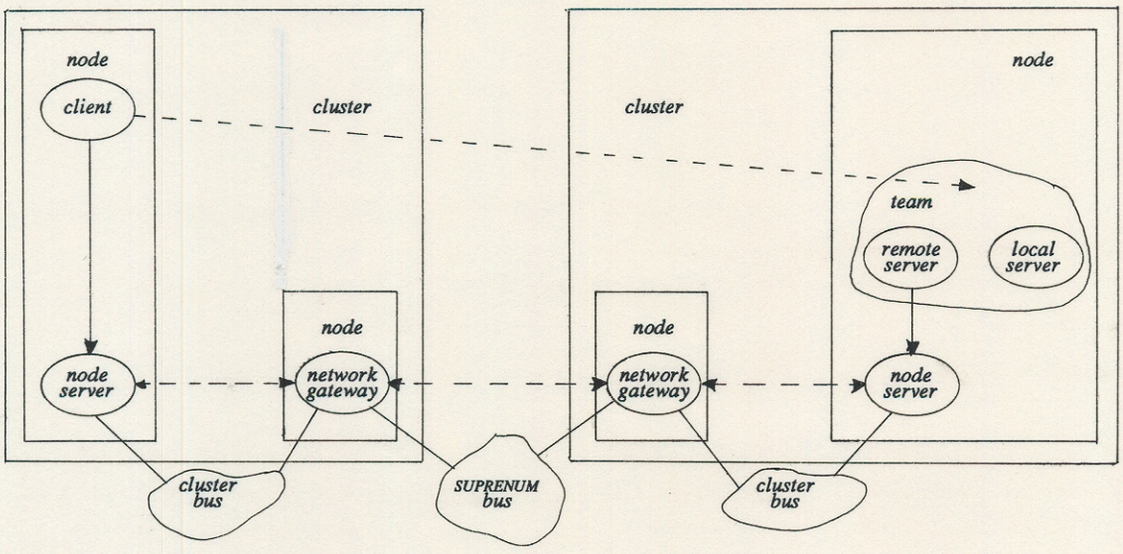

Figure 4.2: Communication between different clusters

The *network gateway* resides on the communication node. Actually, this process combines the functionality of the *node server*, as well as the *cluster server*. The latter mentioned system process drives the SUPRENUM bus, however it realizes the same functionality as the *node server* does. More specifically, the main difference between *node server* and *cluster server* is given by the various device handling modules necessary for driving the two SUPRENUM network interfaces[18].

If a *node server* receives a message-encoded service request from a client, whereby the service request is directed to a different cluster than that of the *node server*, it relays this message to the *network gateway*. The *network gateway* address (process identification) is the same as that of the *node/cluster server* residing on the communication node of the client's cluster. Once the client cluster *network gateway* receives an inter-cluster message, it determines the actual cluster where the *remote server* of that message resides. To distinguish an intra-cluster message from an inter-cluster one, the *node* member, more specifically its *cluster* sub-field, of the *remote server* process identification is used. If that member is different from the corresponding one of the *network gateway* process identification then inter-cluster communication is established.

[18] As with the node administrator, conceptually a seperate *cluster client* assists the *cluster server* in processing inter-cluster communication requests. Both processes constitute the cluster administrator. The same holds for the *network gateway*, which more generally is termed gateway administrator in PEACE.

To realize the *network gateway* route facility, the process identification of the *remote server* itself is taken from the message to be routed. More precisely, this identification is extracted out of the protocol data unit, i.e. the *server* member, associated with the message-encoded remote service invocation. Once determined the cluster[19] on which the *remote server* resides, the client's *network gateway* (*cluster server*) interacts in a direct way with the corresponding *network gateway* (*cluster server*) responsible for inter-cluster message delivery to the *remote server*. The latter mentioned *network gateway* itself directs all received messages to the corresponding *node server* responsible for intra-cluster message delivery to the *remote server*. Again, the address of this *node server* is taken from the *node* member field of the *remote server* process identification contained in the remote service invocation protocol data unit.

4.5. The Network Communication Protocol

The main requirement stated at inter-node and inter-cluster communication is its performance, i.e. maximal utilization of the physical communication bandwith as specified for the SUPRENUM inter-connection architecture. To fullfil that requirement to a great extend, from the communication software point of view very optimistic assumptions about the underlying network hardware must be met. Especially, this means that time consuming recovery procedures, for example given by complex retransmissions strategies, must be excluded from the communication protocol design, as far as possible.

In [Zwaenepoel 1985] measurements about several communication protocols, well suited for high-performance local area networks, are given. Although these measurements are based on 10 Mbit Ethernet interfaces [Metcalfe, Boggs 1976], it can be taken as a basis for deciding what kind of communication protocol will be the best for SUPRENUM. Moreover, it has been shown that most lost of performance was given because transfering data by the processor from/to the network interface (the Ethernet controller). Such a bottleneck, however, is not given with the hardware design of the SUPRENUM networking interface.

Maximal utilization of the SUPRENUM network hardware is achieved by using a simple *blast protocol* [Zwaenepoel 1985]. The principle behind this protocol is to transmit all data packets belonging to the same message in sequence, with only a single acknowledgement for the entire packet sequence. Thus, network traffic with respect to transmission of acknowledgement packets is minimized. Moreover, the retransmission strategy applied either leads to the complete retransmission of the entire packet sequence or allows for selective retransmission of erroneous packets. What strategy actually would be the one with the best performance is to be investigated once both protocol variants have been realized for SUPRENUM. At first sight, it seems that the complete retransmission of an entire packet sequence will be the best solution. Due to the high-performance cluster bus and SUPRENUM bus, it would take more time controlling selective retransmission by a specific protocol functionality, than it would take to retransmit an entire packet sequence upon the negative acknowledgement issued by the receiving *node/cluster server*. Additionally, to reduce protocol

[19] The cluster address is taken from the *remote server* process identification, too.

overhead while passing network boundaries, thus switching from cluster bus to SUPRENUM bus and vice versa, the same *blast protocol* is used for controlling the communication over these two SUPRENUM inter-connection systems.

With a *blast protocol* the overhead necessary for coping with transmission errors is minimized and, in the very optimistic case, would not be present at all. It is assumed that such errors are very rare, because the quality of the SUPRENUM inter-connection hardware. Without such an assumption, a high-performance communication system, with respect to the requirements stated at SUPRENUM, could never be realized. More precisely, the quality of the SUPRENUM network hardware architecture determines what software overhead must be considered when realizing a high quality communication system.

4.6. Functional Hierarchy

Basically, there are two strategies for realizing an actual structure of PEACE, within which the system components encapsulating the inter-node/inter-cluster communication are included. The one strategy would be to place these system components in team space between level 2 and 3, the other strategy considers the placement on level 0, i.e. in *nucleus* space. The conceptual structure, however, places these components always between level 2 and 3, i.e. just above the basic system components. This is motivated by the fact, that the basic PEACE components must reside on each node and that decentralizing/distributing of PEACE will be done by a proper arrangement of the constructive components.

The system components for inter-node/inter-cluster communication together constitute the communication administrator. This administrator consists of the *node server*, the *cluster server* and the *network gateway*. However, due to the hardware design of a SUPRENUM node, the communication administrator must reside in *nucleus* space and thus actually is placed on level 0. Figure 4.3 depicts that design and the resulting functional hierarchy of the actual structure of PEACE.

Because of that actual structure, the communication administrator is solely realized by one system process [20]. This process is termed the *ghost* of the *nucleus* and represents "process zero" of PEACE. Independently of its functionality, the *ghost* always is present in PEACE because it is the first process started, once the node bootstrap has been performed.

Although placing the communication administrator into *nucleus* space represents not that conceptual clean realization of PEACE, from the technical point of view it is the solution with the best performance achieved. Activating the *ghost* means no additional address space switch and no return from *nucleus* to team space. The same holds for blocking the *ghost* [21]. On the other hand, the communication administrator is integrated in such a way into *nucleus* space that local communication does not suffer from lost of performance. Thus, both forms of

[20] This specific realization only is possible because the communication admimistrator resides in *nucleus* space.

[21] First benchmarks have shown that a runtime overhead of 27% is involved with a *nucleus* call when performing one rendezvous. A rendezvous is realized by the sequence of three *nucleus* calls: *send*, *receive* and *reply*.

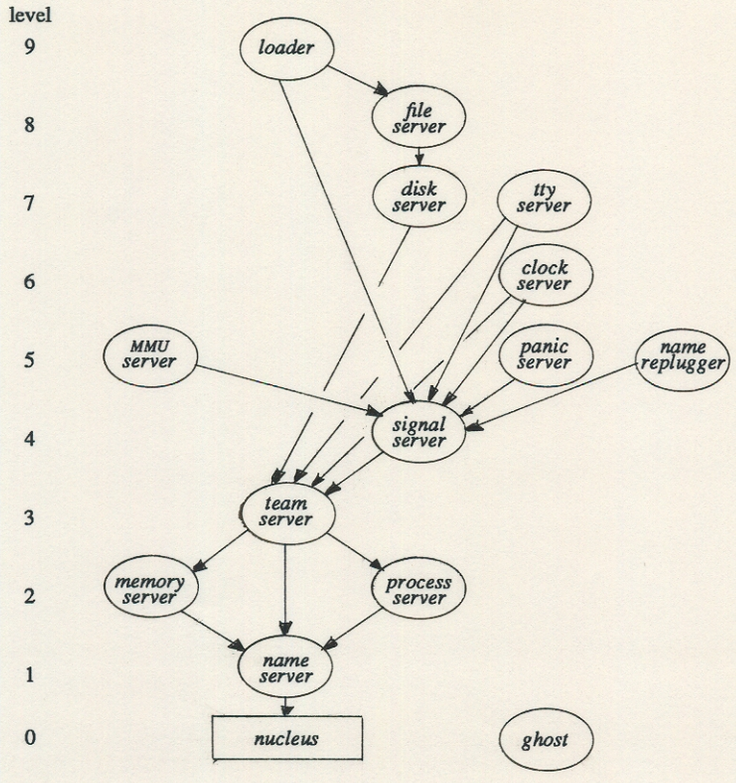

Figure 4.3: The actual structure of PEACE

communication, the local as well as the remote one, optimally are supported by PEACE.

So far the actual hierarchical system structure of PEACE in terms of the inter-relationship between the basic and constructive system processes (components). The following four sub-sections consider this system structure in more detail. It is exemplified in which way PEACE can be distributed over SUPRENUM. In this example, a cluster and some of its qualified nodes are taken as the basis for distributed PEACE. However, this exemplification does not mean the only way of distributing PEACE. There are many other variants practicable. Therefore, the following short breakdowns are only of exemplary nature.

4.6.1. Application Node Breakdown

The subsequent model discussion considers the application node as the part of a cluster that is responsible for running user application programs. In SUPRENUM context, the application programs solve some specific kind of numerical problems and thus can take advantage of certain hardware support, e.g. the floating-point unit of an application node. Figure 4.4 represents the minimal configuration of an application node – it is the minimal configuration of PEACE components residing on one node at all.

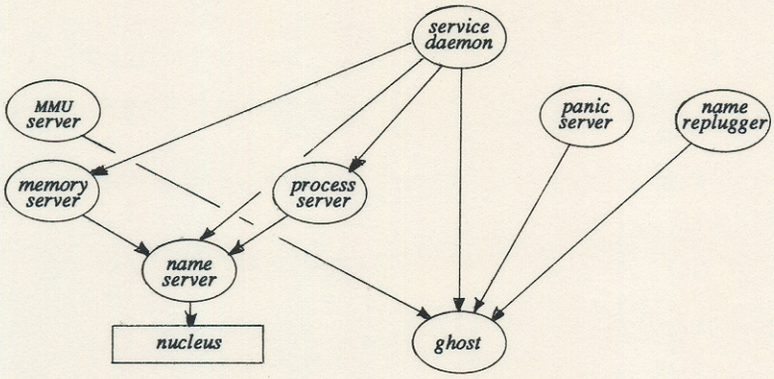

Figure 4.4: An actual application node structure

The only services completely, i.e. stand-alone provided on this node are the one of the name administrator and the address space administrator. Additionally, the *process server* and the *nucleus*, as part of the basic components, completely realizes services on its own.

As shown by that figure, a new and somewhat special system component is identified. This system component is termed the *service daemon*. The only functionality of the *service daemon* is to act as a *remote server* for all remote stated service requests. However, there is no *local server* associated with it, because all services, with except of the name service, are provided remote. The *memory server* and *process server* services only are used by the team administrator, which resides on some remote node. Because of this organization, all stated service requests are mapped by the *service stub* onto remote service invocation. On some other node, may be in a centralized fashion, the constructive services are provided by the corresponding server. Thus, the functionality of the *server daemon* is to bridge the gap between remote residing constructive system components and the basic components, residing on the application node.

Actually, the *service daemon* requests a local basic service for some other remote residing constructive server. To realize its task, the *service daemon* uses services of the *ghost*, which in turn represents the *node/cluster server* or the *network gateway*, respectively. One can say, the *service daemon* is the *deputy* for all non-basic system processes in PEACE. In terms of the remote service invocation model, the *service daemon* encapsulates the corresponding *call stub* for the remote residing *service stubs*.

This general *deputy* functionality of the *service daemon* especially realizes the remote creation and remote termination of processes and/or address spaces, for example. Another important aspect is that of making *remote server* known to the local *name server*. In this way, on any node a *remote* attributed process identification is associated with a service name.

The *MMU server*, the *panic server* and the *name replugger* functionality remains unchanged by this model. The only difference is that signalling some kind of system specific exceptions is realized by remote service invocation. Therefore the inter-relationship between these three system processes and the *ghost*.

4.6.2. Communication Node Breakdown

The next model considered is that for the communication node. The main functionality of this node is to route inter-cluster messages and to control other intra-cluster communication activities. The corresponding model is given with figure 4.5.

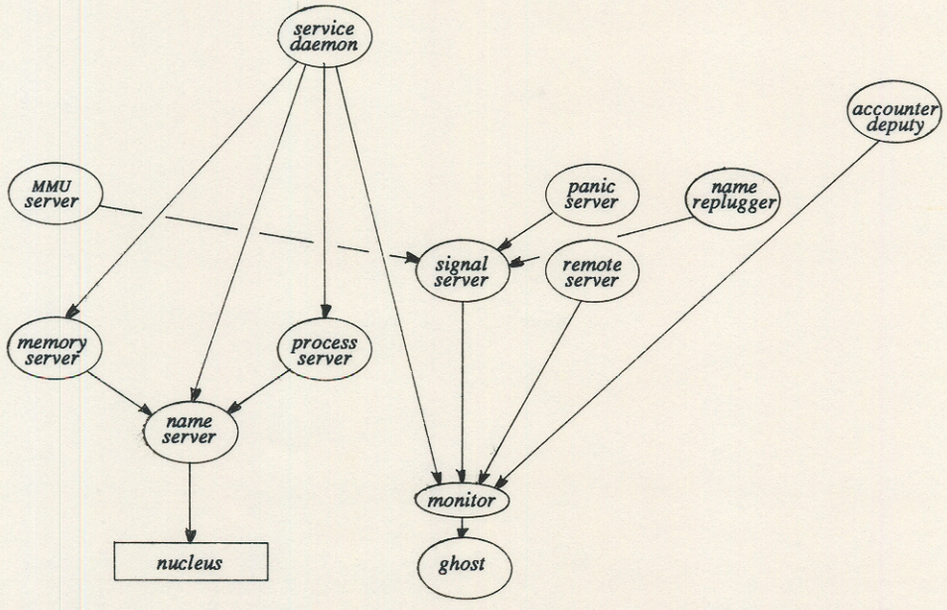

Figure 4.5: An actual communication node structure

The main difference to the application node is the presence of the signal team which actually is responsible for global signal propagation. The other difference is given by the building block for remote controlled accounting and/or monitoring. The functionality of this building block, more specifically that of the *monitor* and the *accounter deputy*, is explained along with the diagnostic node breakdown.

The signal team consists of the *signal server* assisted by its *remote server*. As shown with the application node model, some system processes (the *MMU server*, *panic server* and *name replugger*) will signal system specific events using remote service invocation. Therefore, the *remote server* is placed side by side with the *signal server* in the same team, which in turn is able to process remote requests for signal propagation.

Besides processing of remote signal propagation, remote residing *signal clients* are activated by the *remote server*, too. Because the message-oriented nature of signal propagation in PEACE and because awaiting propagated signals is nothing more than a service provided by the signal team, re-activating a *signal client* is straight forward – there is no difference from terminating other remote rendezvous. The *remote server* just sends a *reply* request to the *ghost*, which in turn interacts with the corresponding remote *ghost*, with the result of replying the *signal client* still blocked on this remote *ghost*.

4.6.3. Disk Node Breakdown

To proceed with the exemplary discussion of the distributed actual structure of PEACE, the team administrator, the *file server* and the disk administrator must be associated with some node. The disk node is used as the host for these system components, as shown by figure 4.6.

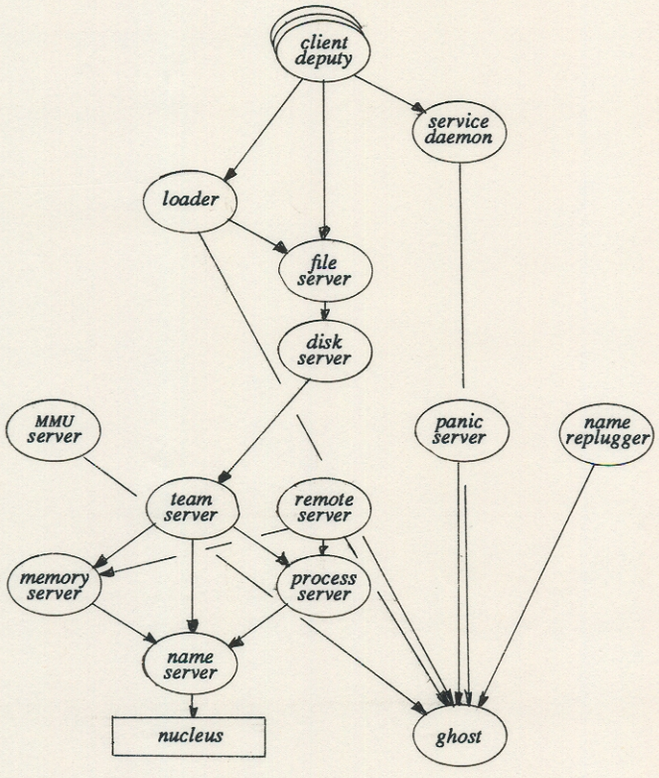

Figure 4.6: An actual disk node structure

The most important aspect given with the disk node model is the necessity of a *client deputy* pool in the *service daemon's* team. Each member of this pool represents a remote residing client which states a remote i/o request, in this case disk i/o.

Some of the *file server* services as well as some of the *loader* services will work blocking on the requesting client. For example, a client receives the *reply* only if the requested file block has been read from disk (and if the block is not hold in the cache, already). The same holds for loading a new program. If the *service daemon* would state these requests on its own, it would block and thus unable to accept in the meantime more incoming remote service requests. However, while a system service is still processed, it should be possible to accept and process other remote stated service requests. Otherwise a dangerous bottleneck, which additionally may produce global deadlock situations, will be designed with the system.

Each time a remote service request, which is a blocking one, is received by the *service daemon* an inactive *client deputy* is taken from the process pool and instructed to process the service request just received. By this way, the *client deputy* blocks for the *service daemon*, which in turn is able to proceed accepting more remote stated requests.

4.6.4. Diagnostic Node Breakdown

The last considerd example is a configuration suited for the diagnostic node. On this node, specifc control and monitor functionalities, for example checkpointing and accounting, are realized. In the model introduced for this node, tty i/o and central clock management is considered, too. Thus, there are two device administrators associated with the diagnostic node, the tty administrator and the clock administrator. Moreover, as an example for the monitor[22] functionality of the diagnostic node, an *accounter* is integrated in the system structure. Figure 4.7 depicts the corresponding actual system structure for the diagnostic node.

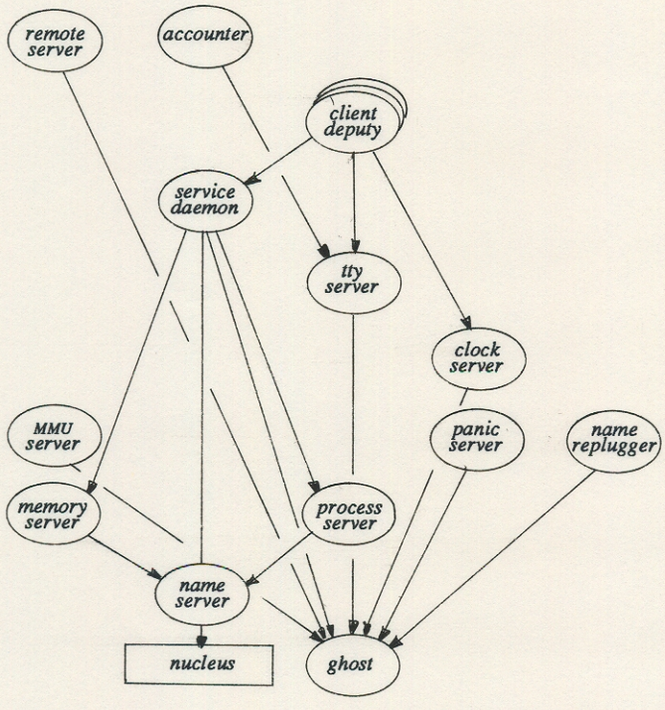

Figure 4.7: An actual diagnostic node structure

The purpose of the *client deputies* assisting the *service daemon* is the same as that already explained in conjunction with the disk node structure. Because there are *clock server* services which, upon request, will delay clients for a specified duration, the *client deputy* principle must be applied, too.

The *accounter* collects informations from several remote residing *monitors*. Such a *monitor* has been placed on the communication node, as an example for this system discussion, and monitors all communication activities in direction to the *ghost* of that node. Thus, all outgoing remote, i.e. inter-cluster and/or intra-cluster, communication activities are remembered. The *accounter deputy* on the communciation node is responsible for the delivery of the remembered account information to the *accounter*. This actually is realized by remote

[22] The model for general monitoring is discussed in more detail in the next chapter. This functionality is realized in PEACE by a so called "building block" for enrichment of the minimal system functionalities.

service invocation on behalf of the *accounter deputy*.

To process remote incoming account informations, the *accounter* is assisted by a *remote server*, both encapsulated by the same team. Once the account information is present, the *remote server* and/or the *accounter* will initiate proper (pre) processing. This may involve printing of status informations onto the connected console, using *tty server* services, or to remember the account informations onto stable storage, using remote file access services relayed by the *ghost*.

Chapter 5

Building Blocks

The basic and constructive components of PEACE together constitute a decentralized/distributed operating system with minimal functionality. Taken these components as the basis, several application specific services may be provided, once the corresponding server/administrator is installed. More specifically, this application oriented configuration of PEACE is supported in a dynamical fashion. With that functionality, PEACE follows the concept of a family of operating systems as, for example, realized with MOOSE.

The system components representing dynamical operating system services are termed the building blocks in PEACE. The administrators of such building blocks are placed at level 10, and above, in the PEACE system hierarchy. However, depending on the functionality of the administrators, some of its processes may be associated with lower levels. In this chapter some of these adminstrators are introduced. By this way the suitability of PEACE to support a broad area of (system/user) applications is exemplified.

5.1. Adoptation of Processes

The basic mechanism for adopting a process in PEACE is given by the routing facility of the *nucleus*. Adopting a process means relaying all messages directed to an adopted process to some specific server. These messages are transmitted by *send*, *reply* and *relay*. More specifically, it is specified by the routing process, what kind of messages, the one supplied by *send* and *relay* or the one supplied by *reply*, is wanted to be routed.

The main system application using the principle of adopting a process are bulk message accounting, monitoring and audit trailing of messages. Additionally specifc forms of deadlock prevention are realized by this principle in PEACE, too. Basic to all these functionalities is a so called *monitor*. This process is responsible for routing a send message, issued either by *send* or *relay*, or a reply message, issued by *reply* and for receiving the re-directed messages.

5.1.1. Accounting

The advantage of the *nucleus* routing facility is illustrated in the following by giving the framework for realizing an *accounter*. The *accounter* constitutes with the *monitor* an accounter team. Figure 5.1 shows that model.

The *monitor's* task is to receive the re-directed messages and to remember these message in some accounter work area. If the *accounter* is not yet running, the *monitor* will *reply* it, thus initiating the specific accounting functionality. It is a good strategy, that the *monitor*

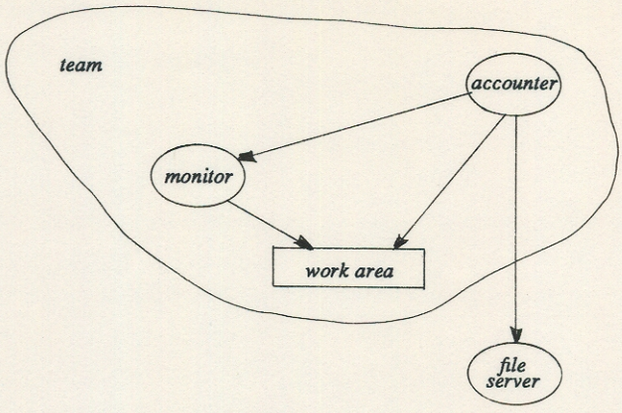

Figure 5.1: Accounting of messages

merely queues the re-directed messages and does not process them. Applying this principle means realizing the minimal delay associated with message accounting.

This model illustrates another important aspect given by the team concept. Actually the *monitor* performs a no-wait send to the *accounter*. This is realized first by queueing the message in the team data space and then replying – which already is a non-blocking communication activity – the *accounter* if it is not yet running. Using the team concept, any non-blocking communication semantic may be realized [Schroeder 1986].

5.1.2. Exceptional Termination of Rendezvous

One aspect for the runtime efficiency of the local inter-process communication primitives in PEACE is that these primitives don't consider exceptional process states which temporarely are associated with any process. This means, for example, that a rendezvous with a terminated process from the *nucleus* point of view is legal. Thus the client will block indefinatively long once it sends a message to the already terminated server. The *monitor* and the *nucleus* routing facility gives an elegant solution of this communication problem. Figure 5.2 illustrates the necessary model as applicated in PEACE.

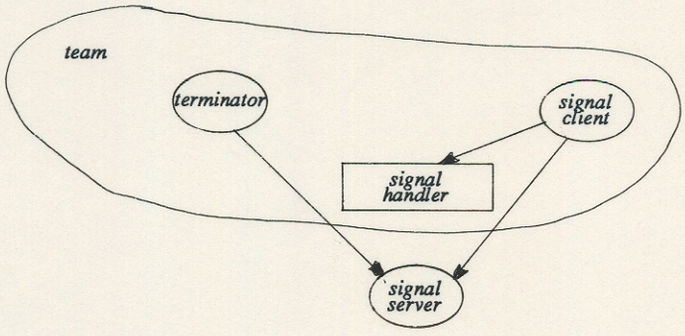

Figure 5.2: Signalling an exceptional rendezvous

In this model the *monitor* is represented by the so called *terminator*, whose task it is to signal a rendezvous exception using *signal server* services. This exception is raised, each time the *terminator* receives a message from a client which sent a message to an already terminated process. Simply, the *terminator* merely accepts re-directed messages. In this case, the terminated server has been adopted by the *terminator*.

The *terminator* functionality is supported by the *signal client* of the terminator team. Each time a process terminates, which is signalled as a process termination exception by the *loader* and propagated by the *signal server*, the *terminator* adopts that terminated process. In contrast to that, each time a process is created, which is signalled and propagated as the process creation exception, the *terminator* relinquishes the adoptation of the respective process. The *signal handler* of the terminator team calls the actual *nucleus* primitive to enable and disable routing for terminated processes, respectively. In the meantime of one's process incarnation, the phase between dead and alive, the *terminator* considers all communication activities with that process as illegal and enables application oriented handling of this system specific exception.

5.2. Adoptation of Services

As with adopting a process, adopting a service means that the original server will not receive a service request on a direct way. Some other server takes the position between client and original server and catches all stated service requests by the client.

The difference to process adoptation is that routing a message-encoded service request is realized by the *name server* and not by the *nucleus*. More precisely, the *name server* does not really route a message, it merely provides services to change the association between a service name and the process identification of a server, i.e. service access point. This means that routing a service request only is possible if a client asks for the server address of the given service name, i.e. at client/server connection establishment time[23)].

An important aspect of service adoptation in PEACE is service-accounting (monitoring), in contrast to the more general message-accounting as realized with process adoptation, and service migration. Once a service has been adopted, another service access point, designating the path to the adopting server, is returned from the *name server* each time the adopted service is asked for by a client. Additionally, adopting a service means signalling the name replug execption by the *name replugger*. In this way, already existing service connections from a client to a server can be updated with respect to the client's context.

[23)] Adaptation of services can be combined with adoptation of processes. In this case, a specific service request may be monitored (service adoptation) as well as the corresponding service termination, indicated by a *reply* (process adoptation, more specifically reply message routing).

5.2.1. Service Migration

As an example for service adoptation the migration of a service onto another node is considered. Service migration in PEACE does not necessarily mean the migration of the entire server or, more specifically, of the entire server team. A server may be splitted in such a way that its main functionality still is provided on its original node and that only a single service of it is migrated onto another node. However, it is worth to note that the model described, here, merely enables continuous service invocation, i.e. identification and addressing of the actual service providing process. Other, service specific management activities, as e.g. updating of data structures local to the server, still must be realized by the service providing process on its own responsibility. The considered model is given by figure 5.3.

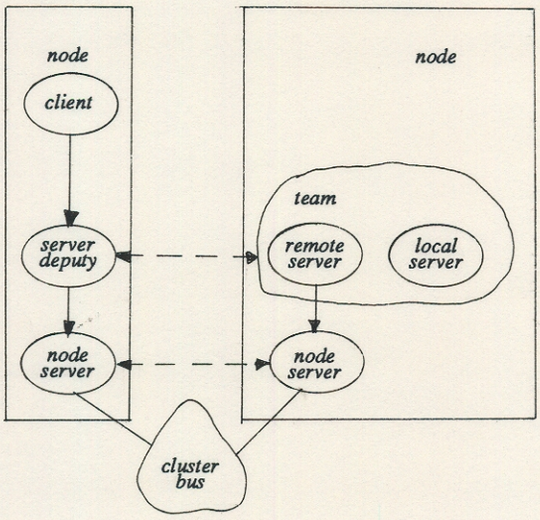

Figure 5.3: Typical configuration for a migrated server team

In this model, the original client/server inter-relationship on one node has been represented by a relationship between the original client and a so called *server deputy*. The *server deputy* realizes a specific interaction with the *remote server*, which in turn represents the migrated service providing process. For example, this configuration is typical for remote file access in PEACE. A *file deputy* resides on the client's node remote from the actual *file server* and drives a file transfer protocol with the corresponding *remote file server*. Usually, the *remote file server* resides on the disk node associated with the SUPRENUM architecture.

5.3. Propagation of Signals

Signal propagation in PEACE is used to distribute system specific events. The applicants of these events are system processes as well as user processes. The global propagation of signals (messages) basically supports the realization of a distributed operating system. For example, distributed resource or process management means the decentralization of process specific control structures. In PEACE there is no single per-process *user structure* as it is for example the case in UNIX. Moreover, in PEACE this structure is decentralized, i.e. each system

administrator maintains an own per-process *user structure* for its management activities. Thus, it is important for these administrators to receive, for example, a notification about the termination of one of its potential client's. Once received that notification, the per-process *user structure* is invalidated by all system administrators.

5.3.1. Control of Checkpointing

One of the main application areas of signal management in PEACE is supporting a checkpoint mechanism for achieving a certain level of fault-tolerance. Signal propagation is used in PEACE to initiate and control the communication activities along with checkpointing, recovery and consistence checking of data structures. Figure 5.4 gives an overview how the common basic model in PEACE looks like to support the mentioned functionalities.

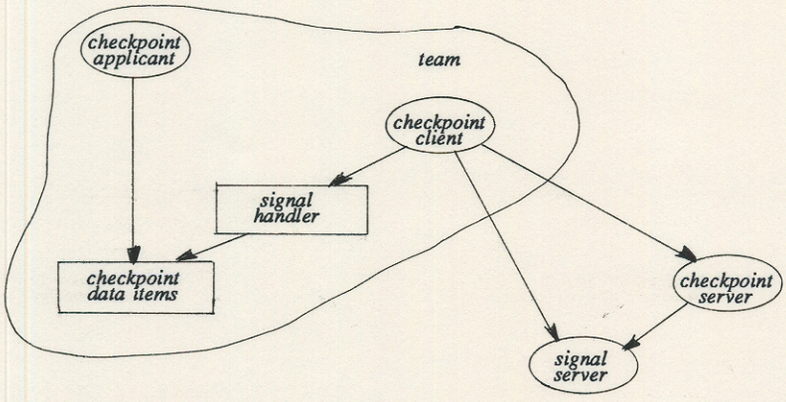

Figure 5.4: Control of Checkpointing

The application, or part of it, to be checkpointed consists of the *checkpoint applicant* and the *checkpoint client*. From the *signal server's* point of view, both processes are the *signal applicant* and the *signal client*, respectively. The checkpoint controlling process is termed the *checkpoint server*.

The *signal server* services are used by the *checkpoint server* to enable the asynchronous activation of *checkpoint clients*, more specifically to call the corresponding *signal handlers* from remote (with respect to the address space and/or to the node) and thus interrupting the execution of the *checkpoint applicant*. Each time the *checkpoint client* receives a checkpoint signal, as raised by the *checkpoint server*, it requests from the *checkpoint server* further instructions. One of such instructions might be to write out critical data items using *file server* services, thus actually fixing a checkpoint for the *checkpoint applicant*. Another, much easier example might be to initiate some check sequence over the critical data items produced by the *checkpoint applicant*. Thus to trigger dynamic data structure verification.

As has already been illustrated, signal propagation is applicable from remote, too. The *checkpoint server* typically resides only once per SUPRENUM cluster, or even once per hyper-cluster. From remote, the *checkpoint server* controlls the activities of the *signal clients*

distributed over SUPRENUM. The *checkpoint clients* may be considered as the "Sleeping Beauty", awoken by the *checkpoint server* in time and realizing team specific checkpointing tasks.

However, it is noted, again, that there is much more work to be done by a checkpoint administrator to achieve fault-tolerance. At least the problems resulting from passing address space as well as node boundaries are removed from such a checkpoint administrator. Thus the only concern of the checkpoint administrator is the checkpointing itself and its co-ordination with other system activities.

Chapter 6

Conclusion

The actual structure of PEACE has been realized by a large number of system processes, following the pattern of a family of operating systems as exemplified by [Parnas 1975], [Habermann et al. 1976] and [Schroeder 1986]. The motivation behind this consequent process structuring is to achieve a high degree of decentralization/distribution of an operating system. With this respect, PEACE is superior to many other operating systems [Balter et al. 1986].

Besides the high degree of decentralization/distribution achieved with PEACE, the flexibility in terms of a dynamical reconfiguration of the system at runtime is another characteristic feature. Reconfiguration may be necessary due to several reasons [Isle et al. 1977]. The main reason for making PEACE dynamical reconfigurateable is to support system fault-tolerance extensively. Because all typical operating system functionalities are encapsulated by dedicated processes, exchanging a faulty system complex, for example, may be realized. Of course, there is a certain limit given because the functionalities of particular system components. In either case, "fire walls" are placed arround the system components to realize a high degree of protection.

Very consequent process structuring must not mean to realize an operating system with little performance. Two aspects are important. At first, no PEACE service request, for example, is provided by more than two processes, if the local case is considered. Thus process switching time is low if compared to the total time a service takes to be executed, e.g. creation of processes. And secondly, inter-process communication directly is supported by the PEACE *nucleus*, independently of user or system processes. Most importantly, routing send and/or reply messages is not realized by the depit of the basic mechanisms of inter-process communication [Schroeder 1987]. In addition to that, remote inter-process communication in PEACE either is realized by routing of messages or by using corresponding services provided by the *ghost* (representing the *node/cluster server* and/or *network gateway*).

As a result of this procedure, service invocation overhead is minimized. For example, a rendezvous is slowed down due to ca. 27 % general overhead because the seperation of team (user) and *nucleus* (supervisor) state. This relative large quota in overhead due to the *nucleus* call/return sequence and argument/result passing is an indication for the high-performance realization of inter-process communication in PEACE. More specifically, it shows that actual *nucleus* management in conjunction with inter-process communication, e.g. process dispatching, process synchronization and message-passing, by way of comparison is really inexpensive with respect to runtime.

Bibliography

[Balter et al. 1986]
R. Balter, A. Donelly, E. Finn, C. Horn, G. Vandome: **Systems Distributes sur Reseau Local – Analyse et Classification**, Esprit project COMANDOS, No 834, 1986

[Behr et al. 1986]
P. M. Behr, W. K. Giloi, H. Mühlenbein: **Rationale and Concepts for the SUPRENUM Supercomputer Architecture**, Gesellschaft für Mathematik und Datenverarbeitung (GMD), 1986

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

[Cheriton 1984]
D. R. Cheriton: **The V Kernel: A Software Base for Distributed Systems**, IEEE Software 1, 2, 19-43, 1984

[Cheriton, Zwaenepoel 1983]
D. R. Cheriton, W. Zwaenepoel: **The Distributed V Kernel and its Performance for Diskless Workstations**, ACM Operating Systems Review, 17, 5, Proceedings of the Ninth ACM Symposium on Operating Systems Principles, Bretton Woods, New Hampshire, 1983

[Dennis, van Horn 1966]
J. B. Dennis, E. C. van Horn: **Programming Semantics for Multiprogrammed Computations**, Comm. ACM, 11, 3, 143-155, 1966

[Fabry 1973]
R. S. Fabry: **Dynamic Verification of Operating System Decisions**, Comm. ACM, 11, 6, 659-668, 1973

[Goodenough 1975]
J. B. Goodenough: **Exception Handling: Issues and a Proposed Notation**, Comm. ACM, 18, 12, 683-696, 1975

[Habermann et al. 1976]
A. N. Habermann, P. Feiler, L. Flon, L. Guarino, L. Cooprider, B. Schwanke: **Modularization and Hierarchy in a Family of Operating Systems**, Carnegie-Mellon University, 1976

[Haertig et al. 1986]
H. Härtig, W. Kühnhauser, W. Lux, H. Streich, G. Goos: **Structure of the BirliX Operating System**, GMD, Insitut für Systemtechnik, 1986

[Isle et al. 1977]
R. Isle, H. Goullon, K.-P. Löhr: **Dynamic Restructuring in an Experimental Operating System**, Technical Report 77-27, TU Berlin, Fachbereich 20 (Informatik), 1977

[Joy et al. 1983]
W. Joy, E. Cooper, R. Fabry, S. Leffler, K. McKusick, D. Mosher: **4.2 BSD System Manual**, University of California at Berkeley, CA 94720, 1983

[Metcalfe, Boggs 1976]
R. M. Metcalfe, D. R Boggs: **Ethernet: Distributed Packet Switching for Local Computer Networks**, Comm. ACM, 19, 7, 395-404, 1976

[Millen 1976]
J. K. Millen: **Security Kernel Validation in Practice**, Comm. ACM, 19, 5, 243-250, 1976

[Nelson 1982]
B. J. Nelson: **Remote Procedure Call**, Carnegie-Mellon University, Report CMU-CS-81-119, 1982

[Organick 1972]
E. Organick: **The Multics System: An Examination of its Structure**, MIT Press, 1972

[Parnas 1974]
D. L. Parnas: **On a 'Buzzword': Hierarchical Structure**, Information Processing 74, North-Holland Publishing Company, 1974

[Parnas 1975]
D. L. Parnas: **On the Design and Development of Program Families**, Forschungsbericht BS I 75/2, TH Darmstadt, 1975

[Parnas 1976]
D. L. Parnas: **Some Hypotheses about the 'uses' Hierarchy for Operating Systems**, Report, TH Darmstadt, 1976

[Popek et al. 1981]
G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, G. Thiel: **LOCUS: A Network Transparent, High Reliability Distributed System**, ACM Operating Systems Review, 15, 5, Proceedings of the Eigth Symposium on Operating Systems Principles, Asilomar Conference Grounds, Pacific Grove, California, 1981

[Randell et al. 1978]
B. Randell, P. A. Lee, P. C. Treleaven: **Reliability Issues in Computing System Design**, ACM Computing Surveys, Vol. 10, No. 2 (June), 1978

[Schoen 1987]
F. Schön: **Hochvolumen Datentransfer in PEACE**, Technical Report, GMD FIRST an der TU Berlin, 1987

[Schroeder 1986]
W. Schröder: **Eine Familie von UNIX-ähnlichen Betriebssystemen - Anwendung von Prozessen und des Nachrichtenübermittlungskonzeptes beim strukturierten Betriebssystementwurf**, Dissertation, TU Berlin, Fachbereich 20 (Informatik), 1986

[Schroeder 1987]
W. Schröder: **Basic Inter-Process Communication in a Decentralized Operating System**, Technical Report, GMD FIRST an der TU Berlin, to be provided, 1987

[Schroeder 1987b]
W. Schröder: **Ligth-Weigthed Processes and the Concept of Teams**, Technical Report, GMD FIRST an der TU Berlin, to be provided, 1987

[Schroeder 1987c]
W. Schröder: **Naming and Identification of Services in a Distributed Operating System**, Technical Report, GMD FIRST an der TU Berlin, to be provided, 1987

[Tanenbaum, van Renesse 1985]
A. S. Tanenbaum, R. van Renesse: **Distributed Operating Systems**, ACM Computing Surveys, Vol. 17, No. 4 (December), 1985

[Thompson, Ritchie 1974]
K. Thompson, D. M. Ritchie: **The UNIX Timesharing System**, Comm. ACM, 17, 7, 365-375, 1974

[Zwaenepoel 1985]
W. Zwaenepoel: **Protocols for Large Data Transfers over Local Networks**, Proceedings Ninth Data Communication Symposium, IEEE, September, 1985