A Distributed Process Execution and Communication Environment for High-Performance Application Systems

W. Schröder

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

ABSTRACT

Simplicity is the slogan in order to design and implement high-performance communication systems. It is almost a natural consequence that simplicity in system design promotes a flexible and/or application-oriented operating system implementation, too. With PEACE, a process execution and communication environment is explained in this paper which consequently follows the maxim of keeping things as simple as possible.

1. Introduction

Up to now, the application of distributed operating systems is limited to a very small area, at least when compared with classic non-distributed operating systems. Basically, this is a consequence from two contrary situations as right highlighted in [Mullender 1986], namely the "*lack of truly distributed applications*" and that "*performance is wrong*". The drawback of many operating systems is the fact that they provide too many services and, most importantly, that they are designed to do so, i.e. not primarily following the maxim "*what ideas to exclude from the design*" [Liskov 1981] and to "*keep things as simple as possible*" [Lampson 1983]. The consequence is low system performance for dedicated application systems. With communication systems exactly the same problem exists and work is in progress in order to find appropriate solutions. On principle, in [Saltzer et al. 1984] rules are presented how to design a hierarchically structured communication system and in [Zwaenepoel 1985] problem-oriented protocol implementations are favorized. A specific structuring and implementation strategy is proposed in [Clark 1985] and one of the latest analysis, addressing the aspect of how to improve communication system performance, is presented in [Watson, Mamrak 1987]. The common tenor of these works is that operating system and communication system are required to consequently support each other in order to give an optimal basis for distributed application systems. In [Tanenbaum, van Renesse 1985] and [Balter et al. 1986] exemplary distributed operating systems are named.

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

As introduced a decade before now, the design principle of a *"family of operating systems"*, described in [Parnas 1975] and [Habermann et al. 1976], shows how to avoid many drawbacks of presently existing operating systems. Following this application-oriented design principle, a distributed *"process execution and communication environment"*, PEACE, is described in this paper, which serves as a basis for a family of distributed/non-distributed operating systems. From the hardware point of view, the architectural framework for PEACE is SUPRENUM, a super-computer for high-performance numerical applications based on a distributed hardware architecture. In [Behr et al. 1986] the rationale and concepts of SUPRENUM are described in detail. It is exactly this framework which mainly influenced the design of PEACE in order to fulfil the strong performance requirements given with SUPRENUM. From the software design point of view, the major foundation of PEACE stems from THOTH [Cheriton 1979] and MOOSE/AX [Schroeder 1986]. With respect to the distributed organization of PEACE, the main influences came from V [Cheriton, Zwaenepoel 1983] and AMOEBA [Mullender, Tanenbaum 1986].

The major goal in the design of PEACE was making SUPRENUM performance directly available to the application system. At the one end, *simplicity* and *performance* was the slogan for design and implementation of lower-level PEACE system components. At the other end, *flexibility* and *network transparency* was required, from the operating as well as application system. These contrary design aspects result in a consequent separation of a problem-oriented from an application-oriented runtime environment for SUPRENUM application systems. The problem-oriented runtime environment is represented by the PEACE kernel and merely provides message-passing and naming functionalities. This environment is 'PEACE' and it has been particularly tuned for network applications. The application-oriented environment is composed by several system processes running on top of the PEACE kernel and is intended for distribution over SUPRENUM.

In the following sections a short overview of the basic PEACE system structure is presented. Actually, the functionality of the PEACE kernel is described. In section 2 the fundamental design decisions for the PEACE kernel are discussed and the functionality of the message-passing kernel is explained. In section 3 it is illustrated how this kernel works on a network-oriented hardware architecture. Section 4 describes how PEACE services are named and identified in a distributed system. A case study for SUPRENUM is discussed in section 5, analyzing the performance of the first PEACE prototype implementation. The conclusion, section 6, is concerned with the illustration of experiences made during the design and implementation of PEACE.

2. Fundamental Design Aspects of PEACE

The following subsections are concerned with a brief illustration of fundamental PEACE concepts. For this purpose, the PEACE operating system structure is introduced and an excerpt of the functionality of the different operating system layers is given. The major concern, however, is to focus on basic functionalities provided by the PEACE message-passing kernel.

2.1. Application-Oriented Operating System Structure

The entire PEACE operating system is structured into ten functional layers. The fundamental functionalities of the various PEACE operating system layers are summarized in table 2.1.

layer	functionality	system processes
9	program loading	loader
8	file and/or directory management, essentially extended name services	file server
7	i/o management for block, character and network special devices	disk server, tty server, network server and appropriate device representatives (*deputies*), respectively
6	clock management and signaling of alarm clocks	clock server and device deputy
5	signaling of low-level system exceptions, such as address space errors, panic events and naming exceptions	MMU server, panic server, name replugger
4	signal propagation i.e. passing user/system exceptions	signal server, signal propagator
3	job management, application-oriented process abstraction	team server
2	management of process and address space objects	memory server, process server
1	naming	name server
0	message-passing and high-volume data transfer on a network-transparent basis, as well as process dispatching and trap/interrupt propagation	ghost and i/o server per major device

Table 2.1: PEACE Operating System Layering

With each layer of the PEACE operating system a set of server processes is associated. These server processes are responsible for the implementation of dedicated operating system services. The interactions between the different server processes are implemented using the remote procedure call paradigm of inter-process communication as described in [Birrell, Nelson 1984]. With each remote procedure call entry an own service function is associated. According to the server's service interface specification, represented by a MODULA-2 definition module, the user and server stubs are automatically created. For

that purpose, a stub generator utility is available in PEACE. A service invocation is topology and/or network transparent, i.e. the invoking process is unaware of the service providing process and of the localization of the process.

The PEACE communication kernel embraces layer 0 and layer 1, i.e. basic inter-process cooperation/communication and naming. The fundamental layer 0 system component is the *nucleus*, providing services for inter-process cooperation and communication. The nucleus is the only system component which is not definitely controlled by a dedicated system process. Its services are not made available by remote procedure calls but by appropriate local ones. In contrast to the nucleus, the PEACE message-passing kernel is associated with a dedicated process, the *ghost* (i.e. process 0). The ghost implements additional low-level services which are interdependent with specific nucleus and/or message-passing functionalities and which are accessible by remote procedure calls. Both, nucleus and ghost share the same address space, so called *nucleus space*, which corresponds to supervisor space of the underlying processor. As a consequence, both system components are required to reside on each node of a PEACE network environment. Against that, the entire PEACE operating system, excepted layer 0, is executed in user space and may be distributed over a network, arbitrarily.

The services provided by layer 0 are mandatory for higher-level user/system components in order to cooperate and/or communicate with each other. The services provided by the other PEACE operating system layers already are considered of being application-dependent. These services only are provided, i.e. the corresponding server processes only are present, if required by higher-level user/system components. For example, with respect to the first SUPRENUM prototype, distributed numerical application programs are available which require layer 0 and layer 1 services from PEACE, only. These applications, and all corresponding processes, initially are created by the PEACE bootstrap service, instead of applying appropriate layer 2 services. Thus, for a specific class of SUPRENUM applications merely a process execution and communication environment is required in order to run distributed/decentralized programs.

2.2. Process Execution Environment

As introduced with THOTH, processes in PEACE are associated with *teams*. A PEACE team specifies a common execution domain for a certain group of *light-weighted processes*. All processes of a team share the same access rights onto PEACE objects, whereby a PEACE object, for example, represents files, devices, address spaces, teams, processes and so on. Following the idea of abstract data types, these access rights are controlled by those system components responsible for the implementation of the respective object. There is no central access control mechanism in PEACE.

The main portion of PEACE objects are implemented by dedicated server processes and, therefore, access control onto these objects is directed to server processes, too. Merely the fundamental access control onto team and process objects is not performed by server processes, but by the PEACE nucleus. This essentially means the validation of interactions basing on the PEACE primitives for inter-process cooperation and communication. Having access right onto a process object, a team is allowed to

manipulate the execution state of the process represented by the respective object, i.e. setting this process ready to run. Having access right onto a team object, a team is allowed to read from and/or write into the address space of the team represented by the respective object.

Besides the team concept, the way scheduling and dispatching works is an important aspect in PEACE. Basically, with each process an own dispatch strategy is associated. This strategy is activated each time the execution state of the respective process changes. Such changes typically occur when a process blocks or is set ready. In a similar fashion, with each team an own schedule strategy and timeslice entry is associated. This strategy is activated each time the per-team's timeslice elapses. The per-team and per-process schedule/dispatch strategies may be combined in order to realize team specific scheduling at process block/ready intervals. In a similar fashion as a team defines a common execution domain for processes, a common scheduling domain for a group of teams may be constructed. For this purpose, the same schedule strategy is associated with all teams of interest. As a consequence of this mechanism, problem-oriented strategies are made feasible on a process/team basis without the need of a central scheduler/dispatcher.

2.3. Process Communication Environment

In PEACE, the communication environment for processes mainly is influenced by the team concept and, with its most elementary functionality, is implemented by the nucleus. According to a *synchronous request-response model*, mechanisms for inter-process communication by message-passing are available on a *send-receive-reply* basis. Between peer teams, 64 byte fixed-size messages are exchanged once a unique client/server inter-relationship on a rendezvous basis has been established. There are no multicast and/or broadcast mechanisms provided by the nucleus.

For a server team, which is qualified by the message receiving server process, a rendezvous actually enables access onto the client process. As a consequence, each process of the server team, and not only the server process which originally received a message, is allowed to terminate the rendezvous by replying a message to the client. In a similar fashion access onto the entire client team is enabled in order to read from and/or write into the client's address space. During a rendezvous, separate primitives for high-volume data transfer, *movefrom* and *moveto*, are applicable by any process of the server team.

The data transfer service provided by the PEACE nucleus is based on sending and/or receiving data items. A data item is the most elementary transfer unit which is processable by low-level hardware components, for example a specific network interface. In case of byte stream oriented interfaces, a data item is represented by a single byte. For SUPRENUM, however, a data item always is 64 bits wide, i.e. it consists of an eight byte fixed-size data block. In addition to this, the data segment is restricted to be data item aligned.

Besides the general message-passing and data transfer functionality, there are specific facilities in PEACE concerning the management of traps and interrupts. Basically, traps and interrupts are represented by messages and, if propagation is requested, passed to

dedicated system processes for further processing.

3. Message-Passing within a Network Environment

In the following subsections the functionality of fundamental PEACE communication and management protocols is described, especially the substantial design decisions are presented. A more complete description of these protocols is given in [Eichler et al. 1987].

3.1. Problem-Oriented Communication System

In PEACE, network communication is based on three main protocol layers. At the top, a *remote procedure call protocol* controls the interface to operating system server processes. Application systems, however, are free to use this protocol for own purposes. The *remote procedure call protocol* is implemented on a library package basis and is supported by dedicated server processes. Basically, this protocol implements duplicate suppression of request and response messages, authentication of client processes and teams, as well as topology transparency.

The next two lower-level protocol layers are implemented by the PEACE message-passing kernel, more specifically, by the nucleus. A *dispatching protocol* regulates remote rendezvous and controls access onto remote team and process objects. The *data transfer protocol* actually handles the transportation of messages on the basis of a datagram service.

3.2. Renunciation on Buffer Management

The PEACE message-passing mechanism for network environments is mainly influenced by the design decision not to use any buffering of messages at the server site. In PEACE, there is no concept of *alien* process descriptors as in V, that is to say for remote processes no virtual representation, on the basis of appropriate state information, is maintained by the nucleus. Rather, processing of incoming rendezvous request messages is controlled by the means of a *dispatching protocol* which is based on CSMA/CD techniques.

If a server is unable to process the incoming message, which may imply buffering, then this situation is considered as a *service collision*. The receive request actually is rejected with a proper indication and the nucleus at the client site is informed about this event. The client process, on behalf of its nucleus, retries the message transmission later on. In order to reduce starvation situations, the receive reject indicates the number of service collisions with the same server process. This collision counter is used by the nucleus at the client site in order to determine the relative retry delay for a new receive request issued by the same client. The retry delay is not global to the entire nucleus but rather specific to the client process which produced the service collision, respectively. The collision counter is decremented each time the server blocks because applying *receive* and it is incremented each time the server is unable to receive the incoming message.

Obviously, this strategy does not completely solve starvation but merely makes it more improbable. More specifically, if client state information is not maintained at the remote server site, starvation is not solvable, at all. The advantage of this strategy is its simplicity. On principle, this strategy may be considered of being the appropriate one even if buffering at the server site is done. For example, if the remote buffer pool exhausted and a client state is not remembered then the receive request is usually to be rejected, too. In PEACE, the buffer pool at the server site exactly consists of one buffer for each process.

3.3. Recommendation of Server Pools

In order to avoid starvation within the *dispatching protocol*, service collisions must be avoided, obviously. Consider the situation in which for each client a light-weighted server process is present. Because a client can only request one rendezvous at a time, no service collision is given any more. Exactly this idea is accounted with PEACE, following the pattern of AMOEBA. A *server pool* of problem-oriented size is maintained by the server team.

Server pool management is a functionality of the PEACE *remote procedure call protocol*. During the binding phase, a client initially makes itself known to the server team and a server process is allocated for this client. By this way, a *service connection* is established between a PEACE client and server team. As a consequence of this service connection, service collisions within the *dispatching protocol* are avoided.

3.4. Renunciation on Perfect Communication

For the design of PEACE communication protocols, three main aspects had been considered. First, avoidance of redundant and/or not required protocol functionalities on different layers. For example, there is no need to provide duplicate message suppression by the lower-level communication system if still accomplished by higher-level application systems [Saltzer et al. 1984]. Second, todays transmission media, especially for local area networks, are of high quality and lost of data packets is not only a problem because of transmission errors. There is also a significant packet lost because of operating system buffer management problems and network interface capabilities [Lantz et al. 1985]. Third, the communication network of SUPRENUM is of very high quality. Transmission errors within the SUPRENUM network are expected with a similar probability as parity errors do occur during memory-to-memory copies [Behr et al. 1986].

Considering the protocols of the PEACE nucleus, the *dispatching protocol* is concerned with management activities, only. These activities mainly are checking access rights onto remote residing team and process objects. Actually, this means the validation of certain rendezvous inter-relationships between the client process and the server team. This validation procedure merely introduces some means of security instead of reliability.

The *data transfer protocol* follows the principle of a *blast protocol* [Zwaenepoel 1985]. The main purpose of this PEACE protocol is to make high-volume data transfer feasible if different network interfaces are used and if network boundaries are to be crossed. For example, segmenting and blocking is not done in order to improve reliability but rather

for being able to use frame-oriented network interfaces, such as ETHERNET [Metcalfe, Boggs 1976], to reduce buffer management problems on network gateways and to avoid excessive blocking delays because of physical transmission activities. Reliability is achieved by an application-dependent end-to-end protocol which is handled either by specific library packages and/or appropriate server complexes. The PEACE *remote procedure call protocol* is a typical example for that.

4. Naming and Identification of Services

The previous sections were concerned with the description of fundamental PEACE ideas. The purpose of this section, now, is to show by what means processes are addressed and on what level topology transparency is achieved in PEACE.

Processes are addressed by unique process identifiers. A PEACE process identifier is represented as a low-level pathname and consists of the triple {*host, team, task*}. Given a process identifier, the team and host membership of a process can be determined as well as the per-team task which actually represents the process. As a consequence of this organization, the PEACE process identifier is a handle how to locate a specific process object within network environments, absolutely and efficiently. Abstraction from team and host membership of a process is achieved by the PEACE naming facility, as described below.

4.1. Service Access Points

Services in PEACE are considered of being any functionality provided by a process. This process is termed the "server" and processes invoking a specific service are termed "client". Services are explicitly made known by server processes applying the PEACE naming facility. Basically, this facility associates *plain character strings*, which represent service names, with process identifiers. Asking for a service name results in the delivery of the associated process identifier. This identifier denotes a *service access point* (SAP) and not the service encapsulating process – although in most cases it directly represents the server process, by default. Figure 4.1 illustrates this functionality.

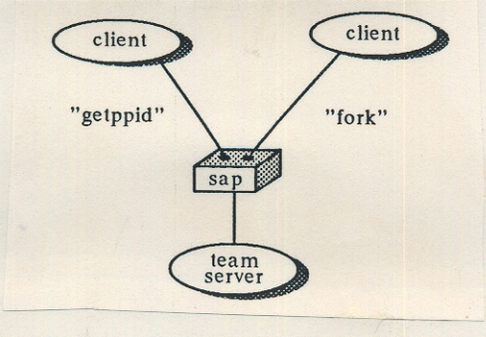

Figure 4.1: Application of Service Access Points

The application of the PEACE naming facility stays in correlation with *remote procedure call protocol* activities and is controlled on a runtime library package basis

supported by appropriate server processes. If a service connection is going to be established, because the client initially issues a service request, a service access point according to the service name is requested from the naming facility. Applying the fundamental nucleus primitive *send*, to this service access point the actual service request is sent and, actually, a remote procedure call is invoked.

4.2. Name Planes

Names exported by processes constitute the PEACE *name space*. Actually, this name space is structured into one or more *name planes*. Within a name plane all names are definite. In contrast to that, within a name space the same name may be multiple defined.

With each name plane an own *name server* is associated, i.e. there is always a one-to-one relationship defined. As a consequence, the PEACE name space may be controlled by several name server processes, dependent on the actual number of name planes constituting the name space. A specific name plane is addressed by the service access point of the corresponding name server. This actually means that name planes itself are identified by name, that is to say according to a service exported by some process.

Applying this naming mechanism, the PEACE name space may be hierarchically structured, thus building a *name tree*. Figure 4.2 depicts this, briefly.

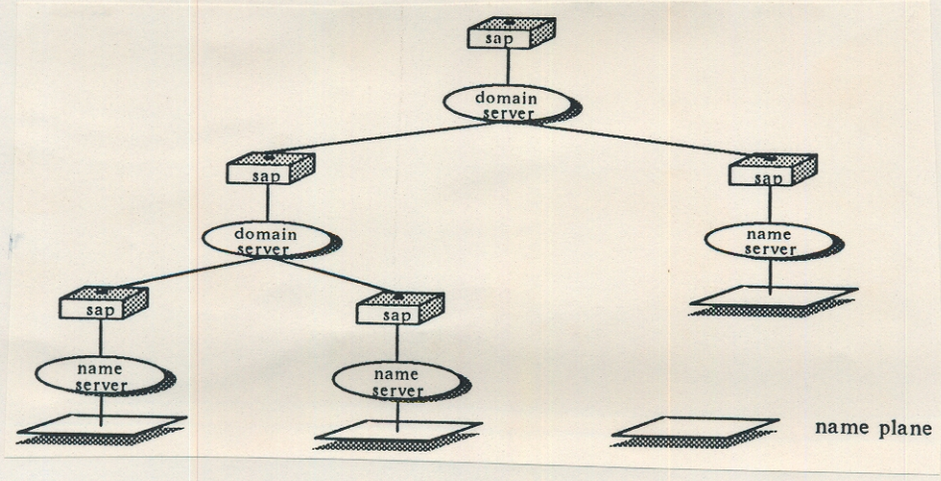

Figure 4.2: A PEACE Name Tree

The leaves of this tree are represented by name servers, more specifically name planes. The coupling between different subtrees is accomplished by dedicated system processes.

4.3. Name Domains

In PEACE, with each team an own *name domain* is associated. A name domain basically is an excerpt from the entire PEACE name space and contains a directory of services which are directly accessible by the specific team. This directory is represented

by a specific set of name planes. The name domain itself is controlled by a dedicated server process, the *domain server*. This process may be identical with a name server if the management of a single name plane is required, only.

As with the per-team schedule strategies, there is no strong one-to-one relationship between a team and its domain server. A group of teams, maybe constituting a distributed application, can be associated with the same domain server. In this situation these teams share the same name space excerpt. Generally, teams associated with different applications are bound to different domain servers. With this principle, an application is made self-contained with respect to identification and addressing of processes and/or teams.

Without a domain server linkage, a process/team is unable to request an operating system service unless the service access point is already known, statically. In PEACE, there is only one such service access point which always is represented by the ghost. The most important services the ghost provides are requesting the domain of a team (*domain*) and directing a team by a specific domain server (*direct*). The latter mentioned ghost service, *direct*, is used for the establishment of a domain server linkage for a team. For each successful request, the ghost returns the process identifier of the previously linked domain server. By this way, a hierarchy of domain servers, and thus of name domains, is created in PEACE. More specifically, in order to establish a name tree the domain server accomplishes the coupling between different subtrees. Figure 4.3 illustrates the inter-relationship between teams and domain server, i.e. name server.

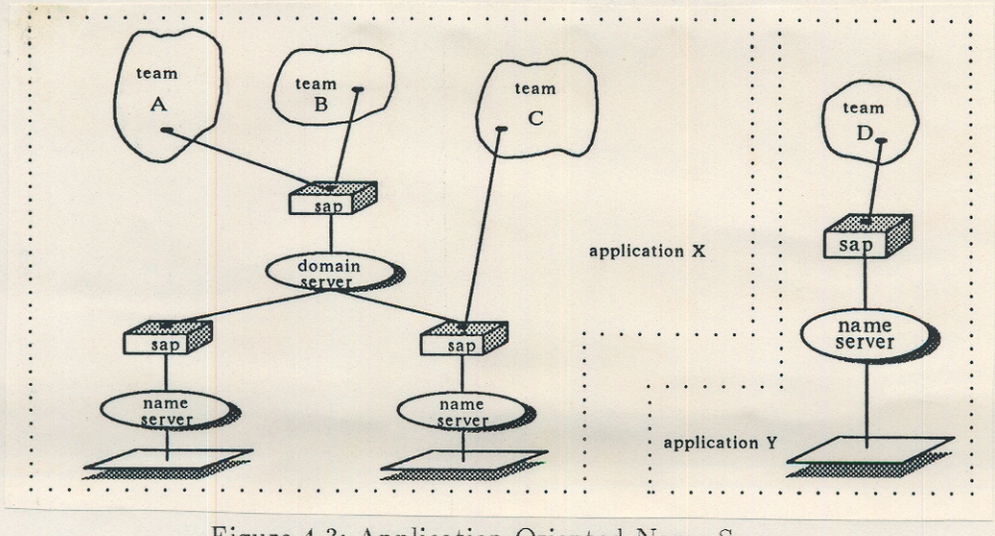

Figure 4.3: Application-Oriented Name Space

There is at least one name domain which is constituted by the fundamental PEACE server processes. From this *PEACE domain* the currently known operating system services can be ascertained. If a system team is created, it is associated with the PEACE domain. The same holds for all teams initially created by the PEACE bootstrap. If a user team is created, it is associated with a default *user domain*. As a consequence, a user application is unable to intrude upon the system in creating service names which are identical with PEACE service names. Solely the application itself is affected.

4.4. Name Scopes

The scope of a service name depends on the functionality of the domain server. The name space observation starts with a search through the per-team name domain. Simple sequential strategies may be considered as well as strategies basing on some kind of multicasting as described in [Cheriton, Mann 1986]. In the same fashion, the domain server decides how to proceed in case of a service name mismatch. By default, the client team terminates if a service name mismatch is indicated. Alternatively, the domain server might try to enforce the existence of the requested service function. This actually means the creation of an appropriate server team.

Generally, the PEACE naming facility provides for a domain relative name resolution, starting from the per-team name domain server. If a service name is made known, i.e. created, this domain server is used, too. It is the responsibility of the domain server to ensure that the service name is unique within the corresponding name domain – a name server merely ensures the uniqueness within a single name plane.

The per-team domain server linkage is of importance if teams are migrated. Independently onto what host a team is migrated its domain server does not change. Thus, for the migrated team the scope of its name domain remains unchanged. The same holds for all other teams belonging to the same application, i.e. sharing a common domain server. Independently of the distribution of this application, the name scope always remains identical.

5. SUPRENUM Case Study

The purpose of the foregoing sections was to illustrate fundamental PEACE design decisions and concepts, briefly. In this section a SUPRENUM case study is discussed by the means of a PEACE prototype implementation and its performance. For that purpose, the SUPRENUM hardware architecture is explained and the PEACE message-passing kernel performance is analyzed.

The GMD research center FIRST at the Technical University of Berlin is associated with design and implementation of a high-performance multi-computer system for numerical applications. This super-computer is called SUPRENUM and its fundamental concepts are illustrated in [Behr et al. 1986], in more detail.

The SUPRENUM development at GMD FIRST addresses both hardware and software for the so called *high-performance processor kernel*. The main portion of software development is concerned with PEACE, the distributed operating system for the SUPRENUM processor kernel. Besides these fundamental project activities, several other areas are covered. A backup file system is developed just as compiler for MIMD FORTRAN and MODULA-2, diagnostic utilities, performance analysis mechanisms, distributed programming environments, process mapping tools, UNIX interfaces, dedicated application programs, and so on. Additionally, the adaptation of UNIX is considered, too.

5.1. Hardware Configuration

SUPRENUM is a multi-computer system, based on a distributed hardware architecture. The building block of SUPRENUM is the *cluster*. According to the functionality and capacity as required by the user, a couple of clusters are inter-connected, thus building a SUPRENUM processor kernel. The inter-connection is accomplished by a high-speed bit-serial slotted-ring bus, the *SUPRENUM bus*, on a row/column basis. The physical bandwidth of this transmission media is approximately 20 Mbytes/sec. Each row and/or column of clusters form a so called *hyper-cluster*. Figure 5.1 depicts the principal SUPRENUM inter-connection structure.

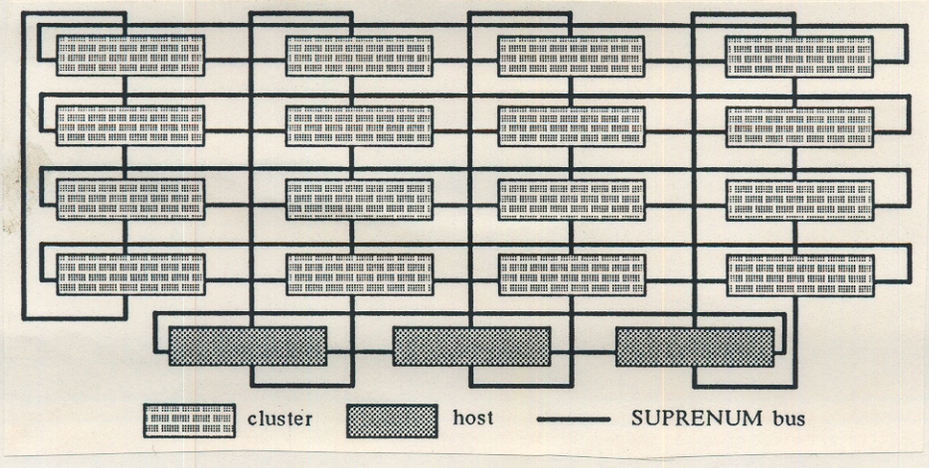

Figure 5.1: SUPRENUM Inter-Connection Structure

This example shows the SUPRENUM processor kernel consisting of 16 clusters and connected to 3 host computers. Each host computer runs a multi-processor version of UNIX SYSTEM V. At the one end, the main functionalities of these hosts are downloading of SUPRENUM applications, diagnostic and maintenance of the SUPRENUM processor kernel. At the other end, SUPRENUM programming environments are supported and the inter-connection with public data networks is made feasible.

The basic processing unit of SUPRENUM is the *node*. Upto 20 nodes constitute a cluster and are inter-connected by a very high-speed parallel bus, the *cluster bus*. The physical bandwidth of this bus is approximately 128 Mbytes/sec – each 50 ns clock tick a 64 bit cluster bus word can be transmitted. The cluster bus is doubled, thus a total bandwidth of 256 Mbytes/sec is specified, physically.

The nodes of a cluster are partitioned into five functional units. From a total of 20 nodes, for the execution of application programs 16 *application nodes* are available. One *stand-by node* serves for fault-tolerant purposes. In addition to these application-oriented nodes, the *disk node* provides for disk i/o services and the *diagnostic node* provides for maintenance services. And finally, the inter-connection of different clusters, as well as the inter-connection to host machines, is made feasible by the *communication node*, which actually serves as a gateway between cluster bus and SUPRENUM bus. Figure 5.2 shows the SUPRENUM cluster structure. Each cluster node is equipped with a

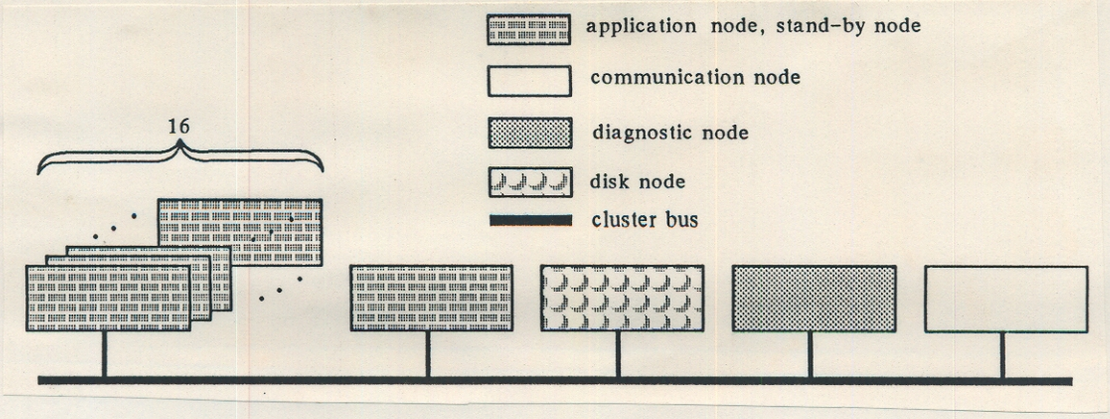

Figure 5.2: SUPRENUM Cluster Structure

20 MHz Motorola mc68020, mc68851 (PMMU), 8 Mbytes of main storage (2 wait states) and a communication coprocessor implementing the cluster bus interface. Each application node is equipped with a floating-point coprocessor, whereas the disk node, the diagnostic node and the communication node, in each case, is equipped with dedicated hardware units for its original purpose.

The first release of SUPRENUM is qualified with a 4×4 cluster matrix. This version consists of 320 nodes, whereby 256 application nodes are made available to the user. The net performance of each application node is specified with 4 Mflops. As a consequence, a net performance of 1 Gflops is calculated for this SUPRENUM release.

5.2. Software Configuration

In order to early provide a process execution and communication environment for a distributed SUPRENUM application, a minimal software configuration was required. This configuration consists of the distributed PEACE kernel. All processes required for this PEACE configuration as well as for the application systems are initially created by the PEACE bootstrap procedure.

5.2.1. Message-Passing Kernel

The actual implementation of the PEACE message-passing kernel addresses all topics discussed previously. This especially means providing mechanisms for network-wide basic inter-process cooperation, by *send* and *reply*, as well as basic inter-team communication, by *movefrom* and *moveto*. Additionally, *send* and *reply* operations can be routed, locally as well as remotely, which is used in PEACE for the integration of relay processes in order to forward messages for migrated processes. Processor utilization is measured on a per-process and/or per-team basis. For this purpose appropriate dispatch and schedule strategies are associated with processes and teams, respectively.

Given a process identifier, the PEACE message-passing kernel distinguishes between local and remote residing process objects. The host member of the process identifier is used for that purpose. Following the pattern of THOTH, access onto local process objects

is directly achieved on the basis of mapping tables. For remote access, first the *dispatching protocol* is used and then the mapping tables of the remote nucleus are applied. Monitoring of communication activities is achieved by the manipulation of mapping table entries, which are forced to address a monitor process.

Bulk data transfer by the means of *movefrom/moveto* distinguishes between inter-cluster and intra-cluster communication. With intra-cluster communication, direct end-to-end data transfer is performed without message segmentation. With inter-cluster communication, a store and forward principle is followed on basis of light-weighted processes acting as representatives for the original client/server processes. As a consequence, high-performance intra-cluster communication is applied in order to store message segments on the communication node and the problem-oriented *data transfer protocol* regulates flow-control, respectively.

The most essential aspect of the present message-passing kernel implementation is a software package which emulates the cluster bus communication coprocessor interface on the basis of a *word transfer protocol*. Currently, the low-level word transfer interface of the cluster bus is made directly accessible by the nucleus network driver. As a consequence, the entire message transfer is controlled by the central processing unit on a 64 bit cluster bus word basis without hardware support for direct memory access.

5.2.2. Name Server

The PEACE name space is structured according to the SUPRENUM architecture. With respect to the operating system, this actually means the presence of at least four different name planes. The *node name plane* contains all names defined relative to a specific node. The *cluster name plane* makes names globally known to all nodes of a cluster and defines cluster-relative names. In a similar fashion, the *hyper-cluster name plane* contains names relative to a SUPRENUM row/column of clusters. And finally, the *system name plane* contains all names unique within SUPRENUM, including the UNIX host machines.

As illustrated in figure 5.3, the PEACE name space is hierarchically structured into four layers.

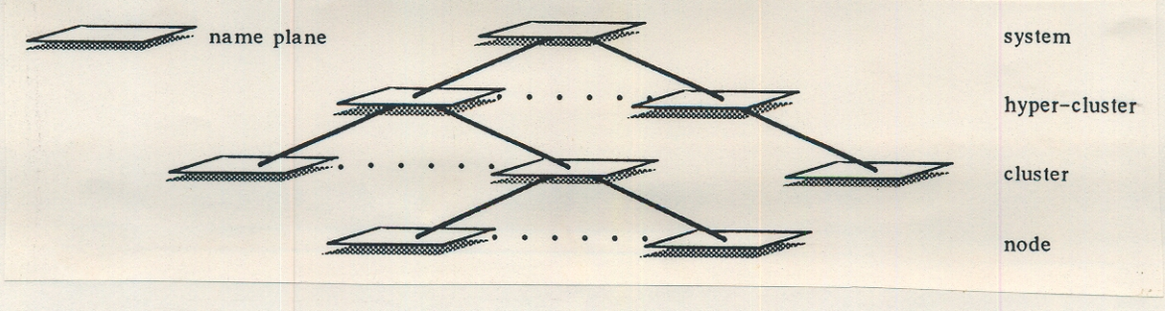

Figure 5.3: The PEACE Name Space for SUPRENUM

In order to locate PEACE services, the search strategy is from bottom up. The PEACE domain server is aware of this hierarchy and, properly, issues name lookup requests to

the various name servers. A sequential request strategy is followed.

5.2.3. Prototype System Structure

Actually, a SUPRENUM cluster consisting of 5 nodes represents the hardware environment for the PEACE kernel. Basing on this hardware facility, the message-passing kernel, more specifically nucleus and ghost, is replicated on each of these nodes.

Because of a limited application environment, the installation of a single name server was required, only. This name server implements the cluster name plane and is directly associated with the domain server linkage of all teams of this cluster. On the basis of remote procedure calls, the name services are made directly available inside the cluster. The name server stub does not implement a server pool.

5.3. Performance Measurements

The SUPRENUM system configuration, which serves as the basis for the performance measurements, was introduced in the previous section. The components of this PEACE configuration were implemented in C. The actual hardware configuration consisted of 5 nodes, each one equipped with an 16 MHz Motorola mc68020, only, and the main storage was accessed with 1.5 wait states. In addition to that, the cluster bus communication coprocessor interface was software emulated.

The following measurements result from running dedicated benchmark sequences. Each sequence was executed 100.000 times. For each run, the elapsed time interval was determined by reading the actual start and stop clock tick value. A clock tick was represented by a 50 ms timeslice and the clock tick interval was expressed in terms of microseconds.

A benchmark sequence consists of one or more nucleus operations, dependent on what measurement was requested. Basically, three different sequences had been considered. For each of these sequences the cluster was exclusively allocated to the benchmark suite. The following subsections present the results of these benchmark sequences.

5.3.1. Fundamental Parameters

The fundamental performance parameters determine the general overhead associated with nucleus and/or message-passing kernel calls, interrupt handling and context switching. Table 5.1 summarizes the benchmark results.

In order to determine the nucleus call overhead, i.e. the delay for switching from user to nucleus space and vice versa, *getpid* was applied. This call simply returns the process identifier of the calling process without any management overhead within the nucleus. In a similar fashion the overhead for remote procedure calls to the message-passing kernel, i.e. the ghost, was measured. In this case, *relinquish* was applied, a ghost call which executes the per-process block and ready strategies. This simply is achieved because of requesting and terminating a rendezvous between user process and ghost.

In order to determine interrupt handling overhead, a dedicated nucleus version was generated. In PEACE, interrupt management is partitioned into three phases. The

action		time (μ sec.)
user/nucleus switch		25
user/kernel switch		375
context switch	*task*	28
	team	28
interrupt phase	*prologue*	36
	synchronization	35
	epilogue	29

Table 5.1: Fundamental Performance Parameters

prologue phase directly is started with each interrupt request and is executed asynchronous, i.e. non-synchronized, to all other operating system activities. In contrast to that, the *epilogue phase* is executed synchronous, i.e. synchronized, and enables interrupt propagation on a message-passing basis to higher-level system processes. The *synchronization phase* serves as the coupling between interrupt prologue and epilogue. This phase is only entered if requested by an interrupt handler.

The context switch overhead was determined on basis of a special purpose nucleus version, too. A specific nucleus call was introduced, which simply forces the calling process to perform a context switch to itself.

5.3.2. Message-Passing

The message-passing performance of the PEACE nucleus was measured for local as well as for remote operation, in each case with a different process pool size. Additionally, the local operation distinguishes between inter-team and intra-team communication. For each case, a *send - receive - reply* sequence was applied. The corresponding results are given in table 5.2.

Dependent on a local or remote operation, the process pool serves for two different purposes. In the local case, the process pool is maintained within the client team and consists of client processes sending to the same server process. In this situation the influence of the server's sender queue size on the overall rendezvous timing is determined. In the remote case, the process pool is maintained within the client and server team, each one residing on different nodes. For each client exactly one server is available. As a consequence, the measured rendezvous timing reflects the message-passing performance in case of no service collisions at the server site.

pool	time (μ sec.)		
	local (n-to-1)		*remote (n-to-n)*
	intra-team	*inter-team*	
1	345	385	2030
2	630	675	3205
4	1195	1245	6375
8	2325	2375	12775
16	4590	4645	25560

Table 5.2: Message-Passing Performance

5.3.3. High-Volume Data Transfer

As done with the message-passing primitives, the performance of high-volume data transfer was measured for local as well as remote operation. For each case, different transfer sizes had been considered. In table 5.3 the results are represented.

size (bytes)	time (μ sec.)		
	local	*remote*	
	movefrom/to	*movefrom*	*moveto*
0	80	1570	2380
8	90	1615	2420
64	105	1635	2435
512	200	1735	2540
1024	305	1860	2660
4096	950	2570	3185

Table 5.3: High-Volume Data Transfer Performance

A transfer size of 0 was used in order to determine the raw management overhead required for the verification of the rendezvous inter-relationship between server and client. The transfer size of 8 bytes determines the minimal overhead for the delivery of a single cluster bus word, whereas the transfer size of 64 bytes does so for the delivery of a single *send/reply* message. The transfer sizes of 512, 1024 and 4096 bytes had been used in order to determine the expected file i/o performance at the client/server interface. Especially, the timing for a transfer size of 4096 bytes indicates the minimal overhead in

case of network-wide paging.

5.4. Performance Analysis

According to the measurements presented in the previous section, a discussion of the results and a general performance assessment is given in the following.

5.4.1. Fundamental Parameters

The fundamental timing parameters of the PEACE nucleus stress the high-performance implementation. These parameters generally influence the performance of each nucleus primitive. Context switching takes place at least once for each rendezvous, in addition to 2 and/or 3 nucleus calls. In case of remote operations, interrupt handling is required. Considering a *send*, *reply* and *movefrom*, the worst case is 2 interrupts per remote operation because of a request and response packet produced and processed by the *dispatching protocol*, not counted interrupt handling because of segmented data transfer. In case of *moveto*, at least one more interrupt is accounted.

Especially for the assessment of remote operations, it is important to notice that, presently, a software emulated *word transfer protocol* interfaces the PEACE nucleus to the SUPRENUM cluster bus. This protocol is migrated into firmware, i.e. microprogram, once the appropriate hardware features are implemented on a SUPRENUM node. By now, this emulation package produces an overhead of at least 500 μsec if a single *dispatching protocol* packet is sent to peer protocol entities, not counted interrupt latency. Presently, this overhead determines the per-node *interface penalty* of a single message setup for a transfer of each one of these packets. Avoiding the interface penalty by a microprogram implementation of the *word transfer protocol*, it is expected to achieve a timing for remote operations which is approximately 36 % of that of the currently accounted one.

5.4.2. Message-Passing

The *dispatching protocol* is responsible for the control of remote message-passing operations. Each packet corresponds to a nucleus message-passing primitive, for example *send*, *reply* and *relay*. Basing on the *word transfer protocol* emulation package, a single remote rendezvous, i.e. a *send - receive - reply* sequence, takes 2.03 ms, including the additional interface penalty because of the *send* and *reply* packets issued by the *dispatching protocol*. First analysis show, that a remote rendezvous timing of less than 730 μsec is possible on basis of the actual PEACE nucleus implementation.

The local rendezvous timing stresses the quality of the PEACE nucleus with respect to high-performance message-passing. An additional performance gain of 30 μsec is achieved, which actually means an improvement of 11 % for intra-team and 8 % for inter-team communication, by combining *receive* and *reply* to a single nucleus call. In PEACE, this call, *replace*, is applied by a server in order to process remote procedure calls, i.e. replacing a rendezvous by another one.

Considering the measurements for different process pool sizes, as illustrated by figure 5.4, a deviation (dark line) from the theoretically expected linear increase (light line) in overall rendezvous timing is obvious.

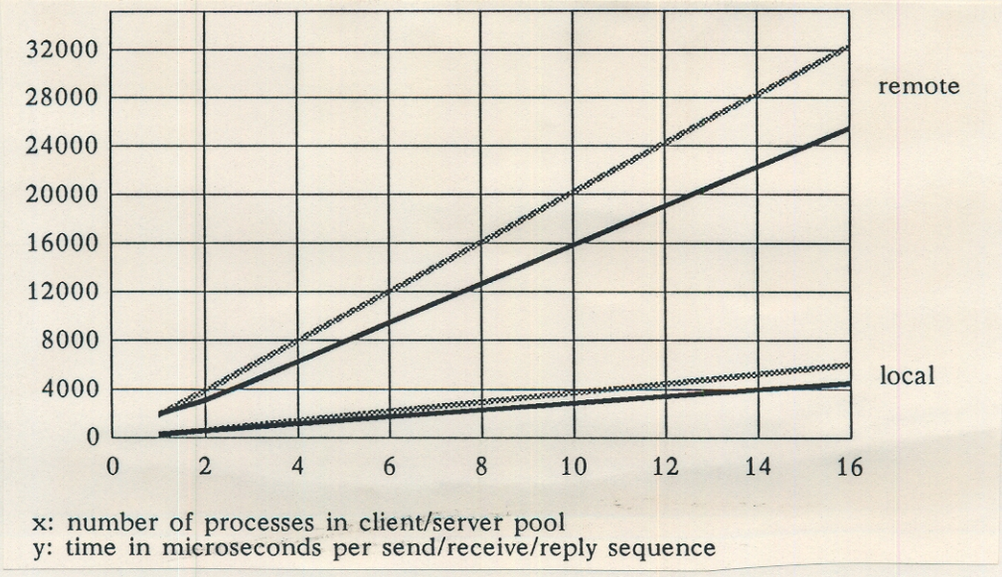

Figure 5.4: Message-Passing Comparison

With local rendezvous, this is due to a non-empty per-server sender queue. In this situation the server will not block because calling *receive*. As a consequence, no per-server dispatch strategy and, thus, no context switch is executed. A general performance gain between 8 % (12 %) and 15 % (25 %) is noticed for intra-team (inter-team) rendezvous. With remote rendezvous, the per-server sender queue is always empty, because for each remote client exactly one light-weighted server exists. The deviation of the rendezvous timing in this case is reasonable because of interrupt-driven protocol activities. Once started, the PEACE network driver, more specifically the *word transfer protocol*, is capable of receiving a sequence of messages, without being interrupted by further network events and without returning to the *dispatching protocol* each time a message has been received, thus reducing protocol switching overhead. This results in a general performance gain of approximately 21 %.

Further analysis of the general performance gain, for local as well as remote rendezvous, shows that the effective per-rendezvous performance gain is inverse proportional to an increase of the process pool size. As a matter of fact, large sender queues and server pools do not significantly improve the timing for a single rendezvous but rather improve the overall communication system performance in case of large system loads.

5.4.3. High-Volume Data Transfer

As with message-passing, the *dispatching protocol* controls remote *movefrom/moveto* sequences. On the one hand, these sequences are only applicable during a rendezvous and, thus, verification of the rendezvous relationship between client and server is

required. On the other hand, at the receiving site announcement of the arrival of an arbitrary sized data stream is sensible. This aspect is essential in PEACE, because it makes a true end-to-end data transfer between peer address spaces, each one residing on different SUPRENUM nodes, feasible without the need of buffering. The announcement of high-volume data means to setup a separate physical data transfer channel at the receiving site, actually programming the direct memory access controller of a SUPRENUM node.

Considering the measured performance of a remote high-volume data transfer, the interface penalty for a single *dispatching protocol* packet is the most limiting factor, again. In figure 5.5 a comparison of local and remote high-volume data transfer performance is given.

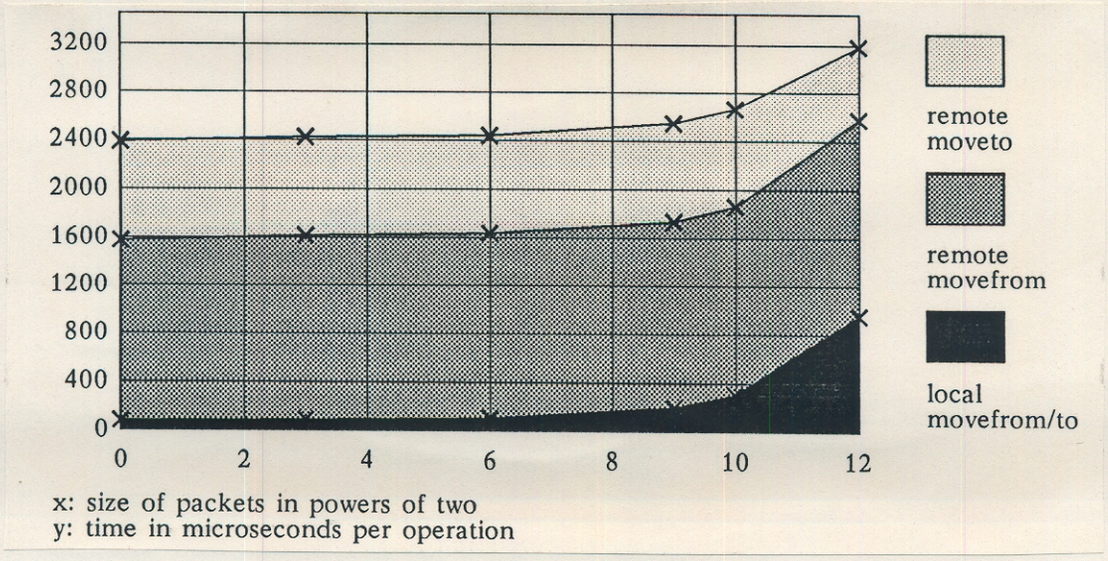

Figure 5.5: High-Volume Data Transfer Comparison

A remote *movefrom* accounts at least two setup times, namely issuing the *movefrom* request packet to the client site and processing this request as well as starting the data transfer at the client site. A remote *moveto* accounts one more setup time, because of an explicit rendezvous verification request packet sent to the client site. Approximately a total of 322 μsec is consumed for general nucleus management activities. Not counting the emulation overhead, the *dispatching protocol* processes a *movefrom/to* in less than 248 μsec, respectively. This timing reflects general nucleus overhead associated with protocol state observation, service collision detection, protocol switching, argument/object passing between protocol layers and checking for local/remote residing process objects. For message-passing, this timing is effective, too.

Remote bulk data transfer of 16/64 Kbytes takes 4.85/12.79 ms for *movefrom* and 5.67/13.62 ms for *moveto*, respectively. This timing results in a read transfer rate of 3.2/4.9 Mbytes and a write transfer rate of 2.7/4.6 Mbytes per second, approximately. Avoiding the interface penalty, it is expected to achieve a *movefrom/to* timing of less than 565/856 μsec, respectively. More specifically, in this situation the transfer rate of

single cluster bus units at least is comparable with that of local memory-to-memory copies.

6. Concluding Remarks

This paper introduced a process execution and communication environment, PEACE, for support of distributed application programs which are suited for the SUPRENUM super-computer. In the following subsections the lessons learned from design and implementation of PEACE are summarized. The conclusion is concerned with a *status quo* overview of SUPRENUM and PEACE.

6.1. Postponement of Design Decisions

The fundamental ideas of PEACE are exclusive application of system processes in order to encapsulate typical operating system services. Basing on processes for service encapsulation and following the design principle of a family of operating systems, decentralization and/or distribution of the PEACE operating system was achieved in a large scale. There is no doubt that this design principle significantly promotes project-oriented system development.

Without functional relief of the PEACE kernel, running first experiments with SUPRENUM would not be possible now. The first message-passing kernel prototype was completed within 6 month and was capable of supporting a numerical application for a cluster configuration of 9 nodes. This rapid prototyping was only possible because of a design philosophy, according to [Parnas 1975], which helps to concentrate on the substantial facts. Most importantly, the message-passing kernel functionality has not changed since that first prototype presentation and there is no idea what functionality to remove from and/or add to.

6.2. About Programming Languages and Compilers

The official programming language for SUPRENUM system programs is MODULA-2. Therefore, the first implementation of the PEACE message-passing kernel was done in MODULA-2. For comparison purposes with other message-passing kernels, such as V, a re-implementation in C followed. With this version a peak performance gain of 43 % was achieved.

This large performance gain solely was achieved because C explicitly knows "*register*" as a storage class designator for plain data objects. The nucleus implementation heavily uses this language feature. Surely, a MODULA-2 compiler with embedded register allocation/deallocation techniques will produce comparable results. However, the overall system performance and/or functionality of PEACE depends on the availability of such a compiler. For an operating system designer/programmer, this kind of dependence is not acceptable.

Besides the requirement of specific and valuable language features for operating system implementation, there is another important aspect concerning the quality of a compiler. More specifically, dependent on the underlying processor, alignment of data

objects may significantly influence the overall system performance. To give an example, the local inter-team rendezvous timing of the PEACE nucleus was improved by 13 %, from 440 μsec to 385 μsec, once proper mc68020 stack pointer alignment, in 32 bit units, was ensured. In addition to that, ensuring message buffer alignment result in a performance gain of approximately 5 % per rendezvous.

6.3. Use of Large Teams

The team concept of PEACE influenced the entire operating system design in every respect. This holds not only for the actual operating system but also for the application system, more specifically the system library.

Mechanisms for asynchronous inter-process communication are provided on a library basis using light-weighted processes. In a similar fashion, propagation of system exceptions is supported, even within a network environment. The remote procedure call system applies light-weighted processes at the server site in order to maintain service connections and to significantly improve the invocation delay of remote procedure calls. The interrupt system does so in order to represent devices as processes and, thus, logically enabling a device to send a message to some, maybe remote residing, server process. A process can be represented as a *sandwich* [Parnas 1976] in order to avoid deadlock situations which will occur in PEACE if processes use each other by appropriately requesting rendezvous.

All these examples show that there may be a large number of light-weighted processes inside a single team. However, this is not considered as a drawback. The essential aspect is that these process resources, especially the different runtime stacks, are bound to the team and that the team is subject for access control and/or scheduling of resources allocated by the single processes. The operating system, essentially the nucleus, only is concerned with a large process table which, actually, makes no harm. Presently, PEACE supports upto 256 processes and 64 teams per node and this actually suffices.

6.4. About Network Transparency

Especially with distributed systems, in which, for example, process migration introduces a significant aspect of system dynamic, the process identification mechanism has to provide some means of network-transparency. However, what the meaning of transparency actually is, depends on higher-level system/user application functionalities. For instance, if a decentralized/distributed application expects a specific process mapping for the underlying network architecture then migrating a process out of this interdependent environment may have a drawback on the overall application performance. The communication delay may become worser and, thus, even in the case that process identifiers implement some kind of location independence, real network and/or topology transparency is not achieved. Using forwarding addresses as in DEMOS/MP [Powell, Miller 1983] or relay processes in order to continuously reach the original process can be a temporarily solution, only. Rather the dynamical events within an interdependent and dedicated application environment should be considered as exceptional conditions and, thus, appropriately signaled by the operating system.

A similar example, which potentially may introduce loss of transparency on the application level, is given with network communication systems, especially if the capabilities of low-level network interfaces are considered. The essential design decision in this conjunction is what basic data transfer unit specification to use at the message-passing interface. For example, basing on a byte stream oriented interface a SUPRENUM user would realize that his distributed application performs better if 64 bit aligned data segments are exchanged instead of byte aligned ones. This will be due to temporarily buffering of a portion of the data segment in order to enforce alignment within the message-passing kernel. Thus, loss of transparency will be deliberately accepted by the user in order to achieve a general communication performance gain. The stream i/o library of UNIX is another typical example of this situation. As already pointed out in [Parnas, Siewiorek 1972], such transparency considerations are of significant importance when specifying the functionality of and/or designing a service interface.

With this respect, designing more intelligence and/or functionality into the message-passing kernel would not really solve the transparency problem. Rather, postponement of the critical design decisions is necessary. An application system should have the chance to define transparency according to its own demands. For example, it is a simple affair either by higher-level user/system components or by runtime libraries or by a compiler and/or linker to ensure proper alignment for communication data segments. In the case of process and/or service migration, handling of signaled migration exceptions may result in rebinding a service access point, i.e. applying the naming facility, again.

6.5. Network Interfaces

The functionality of network interfaces significantly determines the overall communication protocol performance. However, it is a sophism that primarily protocol functionalities should be migrated into low-level hardware and firmware components. As right highlighted in [Lantz et al. 1985], *network bandwidth is rendered virtually insignificant* and/or *faster hosts are needed*. The main performance bottleneck is the network interface and the fact that, usually, the host is busy because of interrupt handling, synchronization, queuing, buffering, and so on. Thus, in order to improve communication performance the first step is to reduce host-bounded operating/communication system activities and the second step, if at all, might be the migration of protocol functionalities into low-level hardware.

Lessons learned from the PEACE communication system design show that a high-performance network interface, above all, should be supported by a clever direct memory access controller. From the communication system point of view the controller should provide for three main services. First, a set of segment descriptors, each one designating possibly variable sized message segments, should be manageable. Second, a differentiation between system channel and user channel should be made feasible. Third, some means of multiplexing/demultiplexing of a single physical transfer channel should be possible, using logical channel numbers generated on behalf of the communication system. The controller merely should consider a channel number as a hash-key in order to locate a specific segment descriptor at the receiving site. The announcement of the

hash-key at the receiving site again is a functionality of the higher-level communication system.

Generally, the main functionality of a network interface controller should be to significantly reduce message-transfer setup times and to enable end-to-end data transfer between address spaces residing on different nodes. Thus, primarily management aspects should be addressed instead of communication protocol aspects.

6.6. About Communication Reliability

Although communication performance is significantly limited by the interface penalty of the low-level *word transfer protocol*, basing on a software emulation was highly informative. In fact, intra-cluster communication is highly reliable and there is little to improve by higher-level communication protocols.

Considering the fact that a future microprogram implementation will not lower communication reliability, directly interfacing the *dispatching protocol* to the cluster bus interface is the adequate solution for SUPRENUM. The *data transfer protocol* will be used for inter-cluster communication, only. Thus, consequently separating different concerns – namely transportation of messages and access control of remote process/team objects – by different protocol layers, was the right design principle in order to implement a high-performance communication system for SUPRENUM, above all.

6.7. Status Quo

Basing on the distributed PEACE kernel, the PEACE operating system is going to be completed, step by step. Presently, layer 0 and 1 are running and layer 2 upto layer 6 have been implemented. Integration and testing of these layers is done from bottom up, now. Additionally, the development of mechanisms for remote file access from UNIX is in progress. Program loading, i.e. layer 9, is made feasible as far as the file i/o interface has been implemented.

With respect to hardware, the completion of the cluster bus communication interface is in progress and the communication node is going to be equipped with an ETHERNET controller. A second SUPRENUM prototype, consisting of two clusters and a UNIX host inter-connected by ETHERNET, is planed to be completed by fall of this year (1987). For this purpose, gateway functionalities are integrated into PEACE, now.

Acknowledgments

Only because of excellent team work the design of PEACE and the implementation of the presently available software configuration was possible. Essential conceptual work was done by F. Schön and W. Seidel. The bootstrap procedure was implemented by B. Oestmann. J. Nolte developed the *dispatching protocol* and a remote procedure call interface for MODULA-2. Last but not least, L. Eichler and Th. Patzelt were concerned with design and implementation of the *word transfer protocol*.

References

[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

[Birrell, Nelson 1984]

A. D. Birrell, B. J. Nelson: **Implementing Remote Procedure Calls**, ACM Transactions on Computer Systems, Vol. 2, No. 1, 39-59, 1984

[Cheriton 1979]

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

[Cheriton, Mann 1986]

D. R. Cheriton, T. P. Mann: **A Decentralized Naming Facility**, Technical Report STAN-CS-86-1098, Department of Computer Science, Stanford University, 1986

[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

[Clark 1985]

D. D. Clark: **The Structuring of Systems Using Upcalls**, ACM Operating Systems Review, 19, 5, Proceedings of the Tenth ACM Symposium on Operating Systems Principles, Orcas Island, Washington, 1985

[Eichler et al. 1987]

L. Eichler, J. Nolte, T. Patzelt, F. Schön, W. Schröder, W. Seidel: **Communication- and Management Protocols for the Distributed PEACE Operating System**, Technical Report, GMD FIRST an der TU Berlin, 1987

[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

[Lampson 1983]

B. W. Lampson: **Hints for Computer System Design**, ACM Operating Systems Review, 17, 5, Proceedings of the Ninth ACM Symposium on Operating Systems Principles, Bretton Woods, New Hampshire, 10-13 October, 1983

[Lantz et al. 1985]

K. A. Lantz, W. I. Nowicki, M. M. Theimer: **An Empirical Study of Distributed**

Application Performance, Technical Report STAN-CS-86-1117 (also available as CSL-85-287), Department of Computer Science, Stanford University, 1985

[Liskov 1981]

B. H. Liskov: **Report on the Workshop on Fundamental Issues in Distributed Computing**, ACM Operating Systems Review, 15, 3, 1981

[Metcalfe, Boggs 1976]

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

[Mullender 1986]

S. J. Mullender: **Report on the Workshop on Making Distributed Systems Work**, ACM Operating Systems Review, 21, 1, 1986

[Mullender, Tanenbaum 1986]

S. J. Mullender, A. S. Tanenbaum: **The Design of a Capability-Based Distributed Operating System**, The Computer Journal,, Vol. 29, No. 4, 1986

[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

[Parnas, Siewiorek 1972]

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

[Powell, Miller 1983]

M. L. Powell, B. P. Miller: **Process Migration in DEMOS/MP**, ACM Operating Systems Review, 17, 5, Proceedings of the Ninth ACM Symposium on Operating Systems Principles, Bretton Woods, New Hampshire, 1983

[Saltzer et al. 1984]

J.H. Saltzer, D.P. Reed, D.D. Clark: **End-To-End Arguments in System Design**, ACM Transactions on Computer Systems, Vol. 2, No. 4 (November), 277-288, 1984

[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

[Tanenbaum, van Renesse 1985]

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

[Watson, Mamrak 1987]

R. W. Watson, S. A. Mamrak: **Gaining Efficiency in Transport Services by**

Appropriate Design and Implementation Choices, ACM Transactions on Computer Systems, Vol. 5, No. 2 (May), 97-120, 1987

[Zwaenepoel 1985]

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