Third International Conference on Supercomputing (ICS 88)
Boston, Massachusetts, May 15-20, 1988

PEACE: A Distributed Operating System for an MIMD Message-Passing Architecture*

Wolfgang Schröder

German National Research Center for Computer Science
GMD FIRST, Berlin, FRG

Abstract. **MIMD message-passing architectures are decentralized computer systems. These systems require a decentralized/distributed operating system, which has to provide a runtime environment for the MIMD programs being processed by the actual machine. With PEACE, a process execution and communication environment, which is particularly designed for message-passing architectures, is explained in this paper. This environment is layered on top of SUPRENUM, an MIMD message-passing super-computer, and is carefully tuned in order to make the performance of this system "direct" available to the application level.**

1. Introduction

Today, two major MIMD design principles are favorized. The one design principle is based on the traditional idea of inter-connecting multi-processor systems by *shared memory*. The other and modern design principle takes advantage of network systems and is based on *message-passing*. This latter mentioned approach is very close to the original idea of autonomous processing nodes in an MIMD architecture. Reason for that is the loosely-coupled nature of network systems and the natural constrain to design and implement "true" MIMD programs whose instructions/operations process completely different data sets. The major criterion raised against an application of message-passing systems is low system performance when compared with traditional shared memory systems. Achieving high performance is difficult, because the operating system designer for those machines has to deal with all conceptual and technical problems of decentralized/distributed computer systems [8]. Solutions for these problems generally imply more software overhead.

Nowadays, progress in designing distributed systems can be discovered - although it is still a non-trivial task to build these systems [16]. There are several experimental/working systems, see [27] and [2] for an overview, which especially serve as a basis for making experiences on distributed programming in the large [18]. These experiences manifest that network-oriented and message-passing systems are superior to systems based on shared memory, especially with respect to availability, scalability, functionality and reconfiguration. With those systems, loss of performance is not necessarily the consequence, as the *Cosmic Cube* [25] illustrates.

For making distributed systems work, a very careful operating system design is required [16] and the slogan must be to keep things as simple as possible [9]. It is essential to decide what ideas to exclude from the design and not what functionalities to include [12]. As introduced a decade ago, the design principle of a "*family of operating systems*", described in [20] and [7] shows how to correspond to these maxims. The distributed operating system described in this paper consequently follows that family concept. The result is a distributed "*process execution and communication environment*", PEACE, which makes the performance of MIMD message-passing architectures "directly" available to the application level. The hardware environment for PEACE is

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

SUPRENUM, an MIMD super-computer based on a distributed hardware architecture and following the principle of message-passing for inter-connecting different processing nodes [3].

In section 2, the SUPRENUM hardware architecture is explained. Section 3 discusses operating system requirements specific to SUPRENUM. The PEACE operating system structure is explained in section 4. In section 5, some performance measurements are presented. These measurements are related to the PEACE message-passing kernel running on a SUPRENUM prototype. Related works and concluding remarks are presented in section 6.

2. The SUPRENUM Hardware Architecture

SUPRENUM is a multi-computer system based on a distributed hardware architecture. The first release of SUPRENUM consists of 320 nodes, whereby 256 application nodes are made available to the user. The residual 64 nodes are used for system maintenance purposes. 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.

2.1. Building Blocks

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 *high-performance processor kernel*. Figure 1 depicts the principal SUPRENUM inter-connection structure.

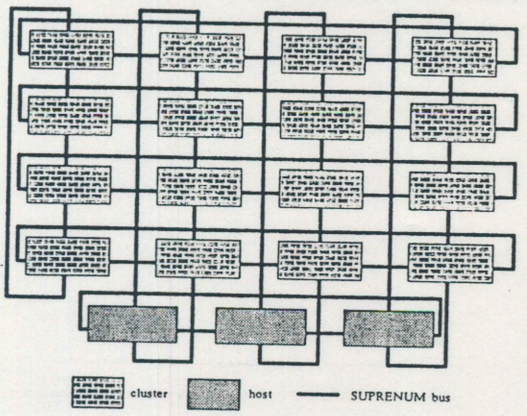

Figure 1.

A high-speed bit-serial ring bus, the *SUPRENUM bus*, inter-connects the various building blocks on a row/column basis. The physical bandwidth of this transmission media is approximately 16 Mbytes/sec. Each row/column of clusters form a so called *hyper-cluster*.

The first release of a 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[1]. At the one end, the main functionalities of these hosts are downloading of SUPRENUM applications, diagnosis 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.

2.2. Processing Units

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 total physical bandwidth of this bus is 160 Mbytes/sec – each 50 ns clock tick a 64 bit cluster bus word can be transmitted. Considering a point-to-point connection between two nodes, 40 Mbytes/sec are specified as the physical bandwidth. Figure 2 shows the SUPRENUM cluster structure.

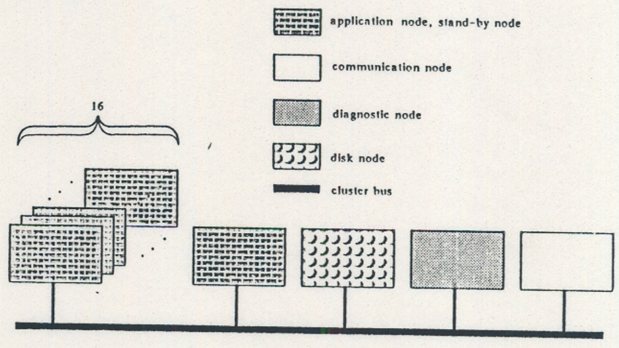

Figure 2.

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.

[1] UNIX is a registered trademark of *A T & T Bell Laboratories.*

Each cluster node is equipped with a 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.

3. Operating System Requirements

In the following subsections several operating system requirements are discussed. These requirements are specific to SUPRENUM, as a representative for MIMD message-passing architectures, and influenced the entire PEACE design.

3.1. About Communication Bottlenecks

Because communication performance is essential for MIMD message-passing architectures, the most critical hardware resource to be managed by an operating system seems to be the communication network. However, the network itself is not the major bottleneck.

As analyzed in [10] for communication networks based on ETHERNET [13], which are of low network bandwidth when compared with the SUPRENUM network system, the major communication bottleneck is the network interface. On the hardware level, this bottleneck is specified by the functionality of a network or direct memory access controller – similar to the legendary "*von Neumann bottleneck*" [1]. On the software level, this bottleneck is specified by network driver functionalities just as the semantics of inter-process communication primitives. Between both levels a strong inter-relationship exists. At the one end, maximal network utilization depends on software activities controlled by the central processing unit, i.e. the host. As an obvious rule of thumb, the more communication activities are controlled by a host the less network utilization will be. At the other end, complexity of network driver software depends on the network hardware interface capabilities. From the technical point of view, the complicated a programming model of a network interface is the more driver activities are necessary in order to send/receive a message onto/from a network interface.

Besides these hardware dependent aspects, there is also a significant influence on communication system complexity determined by the semantics of higher-level communication interfaces. The operating system designer is particularly faced with the problem of mapping higher-level communication interfaces onto lower-level network interfaces, and vice versa. Obviously, there is a *semantical gap* between these two interfaces and the optimal situation will arise if this gap is only of conceptual nature and will not be technically present, any more. On principal, the larger this semantical gap is the more mapping overhead is accounted within an operating system, i.e. the communication subsystem. As a consequence, overall communication performance just as processor utilization for application programs will drop.

One solution in order to overcome this problem is to claim that faster hosts are needed. The more sensible solution is trying to avoid these problems by a careful system design which is free of those performance bottlenecks. To give an example from the early days of operating system design, performance of paging systems was not significantly improved on the basis of faster disks, but new scheduling and paging algorithms and techniques were responsible for the performance gain. The same holds for improving performance of communication systems. The need for faster hosts is only justified if the designer is sure that there is no other way to go around the performance bottleneck. As a consequence, hardware requirements should be stated late in system design.

3.2. Separation of Concerns

Many MIMD application systems are based on the asynchronous model of inter-process communication, i.e. the *no-wait send* [11]. Several application programmers claim that this model is more fundamental, more efficient and increases parallelism [15]. Considering the implementation of this model in more detail, as done by a system programmer, one can observe that the aspect of being fundamental and efficient is no longer valid. In contrast to a synchronous model, with a *remote invocation send* semantic [11], message buffering is necessary, which requires additional memory management activities. In addition to that, message buffer often are completely copied instead of simply exchanging descriptors, in which case buffer descriptor management is required, only. In the most cases, reliable communication is expected, instead of simple datagram services, which significantly increases communication costs. All these activities are executed by the host, steeling processor cycles for solving the original MIMD application problem.

Just as application processes, executed by an MIMD message-passing machine, communicate by the means of messages, processes constituting the distributed/decentralized operating system will have to do so. This especially means that operating system requests are message-encoded. It is widely accepted to invoke these requests on the basis of *remote procedure calls* [19]. This technique implies *remote invocation send* semantics. Implementing remote procedure calls on the basis of *no-wait send* increases communication protocol overhead. This is to the debit of operating system as well as application system performance. Even worser, this design decision implies deadlock situations. For example, sending a message might be the result of a memory management service request which, in turn, is the result of queuing a message due to a *no-wait send*. In addition to those kinds of resource allocation deadlocks [26], hidden communication deadlocks are of particular meaning. It is well known that programming on the basis of asynchronous inter-process communication primitives is heavily error-prone and often leads to an unintelligible program behaviour. Because of security and reliability reasons these primitives are not appropriate in order to implement an operating system interface for a network environment.

With respect to the semantics of fundamental primitives for inter-process communication, there are two different concerns when considering higher-level distributed MIMD applications and lower-level distributed operating systems. Common to both levels is the communication aspect and the only difference is due to concurrency. In this situation, separation of concerns would mean to separate the communication mechanism from the concurrency mechanism. As noted in [17], it is sensible to implement concurrency by the means of processes instead of messages. Nearly all state-of-the-art operating systems follow this design rule by implementing the concept of *light-weighted processes* sharing the same address space. With THOTH [4] this concept was successfully implemented for a real-time environment, a traditional domain for asynchronous communication mechanisms. One of the most recent operating system examples is MACH [29], which supports concurrency within MIMD programs on the basis of *multiple threads*.

3.3. Postponement of Design Decisions

As noted in a previous subsection, in order to avoid a communication bottleneck a careful operating system design is mandatory. This design must follow the rule of reducing operating system complexity, i.e. designing simple low-level operating system components, only. Dependent on the functionality of operating system services required by application systems, multiple entry points into the operating system must be provided, whereby each entry point directly interfaces the application process with the service providing system component. With this approach, there is no operating system bottleneck through which all service requests are threaded.

From the operating system point of view, consequent modularization is required, i.e. considering each service as a special *object* just as building a multi-level hierarchy of these objects. From the application system point of view, an application-oriented operating system interface is built, with the consequence that only those system components are required which are necessary to support the application. On the lowest operating system level, simple system components are present, only, which serve as a common basis for all higher-level components.

This design principle of a *family of operating systems*, [20] and [7], removes complexity from lower-level system components. Design decisions, whose consequences are not yet clear, are postponed and fixed at a higher (more application-oriented) level in the system hierarchy. Maybe that these design decisions are not fixed at all within the operating system kernel, but rather are fixed directly within the application context, i.e. process. In each case, lower-level system components are more basic and can be used in various contexts. The most essential aspect for MIMD message-passing architectures is, that these components encapsulate less software overhead and, therefore, potentially consume less execution time while being active.

With respect to the discussion performed in the previous subsection, the most fundamental operating system component for an MIMD message-passing architecture will be a simple message-passing kernel. As a requirement for this kernel, a synchronous communication model is appropriate if dispatching of light-weighted processes is supported. On this basis, asynchronous communication models can be implemented on the runtime system library level, for example. Buffering of messages is local to this level, too, with the consequence that no address space boundaries are crossed while the message is copied into the message buffer and that memory resource allocation problems always are bound to the application process instead of being fixed within a central system component. Applications which do not need a *no-wait send* semantic for communication can directly go the fastest possible way an operating system can provide for in order to enable inter-process communication. This is the case for all operating system interactions.

3.4. Actual System Structure

The operating system family concept presupposes some kind of object-oriented system design. Service encapsulating system components are required to reside within an own context and these components must be addressable in a unique way. Processes are the appropriate tools for representing these components. Both requirements, maintaining a context and addressing a service, are fulfilled using a single mechanism. In order to design a decentralized/distributed operating system, this aspect is a natural consequence at all.

A process-oriented operating system has many advantages. It is not only appropriate to support the design of an application-oriented and distributed operating system, but also to support a fault-tolerant system design. As noted in [21], the *actual structure* of a system is one important requirement for making it fault-tolerant. Especially for SUPRENUM it is important to maintain operating system functionality even in the case of host crashes. Single system services, represented by autonomous processes, can be migrated onto other hosts, thus keeping the entire system working. Furthermore, there. is no fundamental difference in making application systems and parts of an operating system fault-tolerant. See [22] for more detail.

3.5. Communication Protocols

MIMD message-passing architectures are based on a communication network which inter-connects the various processing nodes. A communication system controls the interactions between processes distributed over these nodes. Generally, this communication system is constituted by a multi-level protocol hierarchy.

On the most basic level, simple message/data transfer functionalities are observed, whereas another level is responsible for network-wide process synchronization/dispatching. On top of this, a remote procedure call protocol is layered, at least controlling access to operating system services. Above that, a naming protocol is settled in order to address service encapsulating operating system processes, for example. Observing the SUPRENUM architecture in some more detail, inter-networking is required because of two different communication busses, inter-connected by the communication node. This makes another protocol level necessary which has to deal with SUPRENUM-specific gateway activities, especially maintaining routing tables up-to-date.

This 5-level example is typical for a machine like SUPRENUM. Each of these levels implement a *problem-oriented* communication protocol. In order to achieve good communication performance, i.e. reducing host-relative processing time of a communication request, traveling through all of these protocol layers must be avoided with each application-initiated communication request. Following the concept of a family of operating systems, each protocol level can be considered of being a service the communication system provides. Depending on the functionality required, the process decides what communication service to use, i.e. attaching what protocol level. This means to provide for problem-oriented communication services in an application-oriented way.

4. The PEACE Software Architecture

PEACE is a decentralized/distributed process execution and communication environment based on a process-oriented system architecture. A multi-level hierarchy of system processes provides for operating system services in an application-oriented way. This section describes by what means the PEACE operating system is constructed and gives an excerpt about the hierarchy of system processes. In [23], a detailed discussion of the PEACE operating system structure can be found.

4.1. Building Blocks

In PEACE, operating system services are provided by processes. These processes, i.e. the services, are encapsulated by *teams*. Following the pattern of THOTH [4], a PEACE team is the notion for grouping several *light-weighted processes*. These processes is given a common execution domain which is qualified by the same access rights onto all objects owned by the team. Objects are memory segments, files, devices just as processes and processors, i.e. resources typically managed by an operating system. In this sense, a team is the traditional *heavy-weighted process* which, for example, is scheduled as a single unit by an operating system.

In addition to the common execution domain, a PEACE team serves for two other purposes. First, it represents the unit of distribution. Second, it is the building block of the operating system, i.e. it encapsulates operating system services. As illustrated with AMOEBA [17], several light-weighted processes, encapsulated by the same team, may be responsible in PEACE for executing services provided by the team. On this basis, concurrent execution of services within a single team is enabled, because with each light-weighted process a unique thread of control is implemented. Switching between these processes is extremely fast. On SUPRENUM, an intra-team user level process switch, controlled by the PEACE kernel, takes about 50 μsec.

4.2. Inter-Process Communication

According to a *remote invocation send* model of communication [11], 64 byte fixed-size messages are exchanged between peer processes on a *send-receive-reply* basis. Once a process (the server) received a message from a peer process (the client) and has not yet replied to the client, arbitrarily sized data streams may be transferred between client and server team. This transfer activity is controlled by the server team on the basis of *movefrom/moveto* primitives. In terms of PEACE, a high-volume data transfer is performed. Each message exchange just as data transfer is network-wide.

As illustrated with V [5], this principle avoids temporary buffering of message/data streams within low-level communication systems. This fact is independent from local or network-wide communication. With respect to the address spaces of peer teams, transfer of message/data streams is always of end-to-end significance. Buffering of messages is only required if low-level network hardware interfaces impose certain alignment restrictions on the data items to be transferred.

Asynchronous communication principles are implemented using two fundamental PEACE mechanisms, namely light-weighted processes and the primitives for message-passing and high-volume data transfer. The light-weighted processes are used in order to establish an address space inter-connection between peer teams. Technically, a non-terminated rendezvous between peer light-weigthed processes, encapsulated by different teams, enables this connection. Because high-volume data transfer, on the basis of *movefrom/moveto*, is a non-blocking activity in PEACE, a *no-wait send* semantic of communication between peer teams is achieved.

4.3. Message-Passing Kernel

The fundamental PEACE system component is the message-passing kernel. The major functionality of this component is to provide for network-wide inter-process communication and high-volume data transfer. In addition to this functionality, dispatching of light-weighted processes just as scheduling of teams is supported. Processor multiplexing is always bound to a single object, which means that with each process an own dispatching and with each team an own scheduling strategy can be associated. What strategies are actually known to the kernel is fixed at configuration time. During runtime, dispatching and scheduling strategies may be dynamically changed for a process and team, respectively. There is no need in PEACE for a host-relative, central and overhead-prone scheduler/dispatcher.

Another aspect considered is low-level device management. For this purpose, the massage-passing kernel implements two abstraction mechanisms, namely:

- trap/interrupt propagation on a message-passing basis
- device driver interaction on a remote procedure call basis

The reason for both mechanisms is to propagate hardware-specific events, which are represented by traps and interrupts, just as to interact with low-level device drivers on a network-wide basis. Remote device access mechanisms are provided in a uniform way, independent if device management really is a host-relative activity.

Basically, a device is represented by a process, thus possessing a system-wide unique process identifier. This process is called in PEACE *device deputy* and is source of a message-passing activity in case of interrupt propagation. In order to propagate an interrupt, the device deputy is requested to send a message to some interrupt-serving (more application-oriented) system process.

The remote procedure call interface of the PEACE message-passing kernel is implemented by at least one light-weighted process, the *ghost*. This process is always present on each host and constitutes the kernel team. Depending on kernel configuration parameters, additional light-weighted processes, each one implementing a remote procedure call interface on its own, may be encapsulated by the original kernel team. Each of these processes is responsible for providing configuration-dependent kernel services, such as host management, low-level address space and process management, network driver management, low-level device management, and so on. The main functionality of these low-level kernel services is to provide for an abstraction level from hardware-specific details, only.

In addition to the remote procedure call interface(s) of the kernel team, there is a local interface which is comparable to typical system call interfaces of procedure-oriented operating systems. By this local interface, message-passing just as high-volume data transfer primitives are accessible. These primitives are provided by the *nucleus* of the PEACE message-passing kernel and are directly accessible by user as well as system processes. The *nucleus* is the most basic PEACE system component whose services are not provided on the basis of remote procedure calls – these services are used in order to implement remote procedure calls. All other PEACE services are accessible on a remote procedure call basis, only.

4.4. Process Identification and Naming

One of the most critical aspects of distributed systems is how to identify and address processes. Identification and addressing mechanisms directly influence inter-process communication performance – and performance is predominant in PEACE. In the following it is explained by what means processes are identified and addressed, and on what level network-transparency is achieved.

4.4.1. Absolute Addressing

Processes are addressed by unique process identifiers. Following the pattern of V, a PEACE process identifier is represented as a low-level pathname and consists of the triple {*host, team, task*}, whereby the single members have the following meaning:

host	is a system-wide unique host identification, denoting either a physical or logical host (processor), depending on the actual system configuration;
team	is a host-relative unique team identification, denoting a heavy-weighted process, i.e. a team;
task	is a team-relative unique task identification, denoting a light-weighted process.

For the ease of handling, a PEACE process identifier always fits into a single register of the underlying processor. For SUPRENUM, this means a 32-bit process identifier, evaluating the triple with the maximal accounts of {*8192, 64, 256*}. As a consequence, with this maximal configuration, PEACE can support upto 2^{27} processes – surely enough for the support of very massive parallel programs.

Given a process identifier, the team and host membership of a process can be directly determined. 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. All PEACE message-passing primitives use a process identifier in order to select the corresponding communication partner.

4.4.2. Relative Addressing

Abstraction from team and host membership of a process is achieved by the PEACE naming facility. This facility associates *symbolic names* with process identifiers.

Names exported by processes constitute the PEACE *name space* which is structured into one or more *name planes*, according to the SUPRENUM architecture. This means the presence of at least four different name plane types:

- 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;
- the *hyper-cluster name plane* contains names relative to a SUPRENUM row/column of clusters;
- the *system name plane* contains all names unique within SUPRENUM, including the UNIX host machines.

The PEACE naming facility is applied by the remote procedure call system to name services provided by processes. Asking for a service name results in the delivery of the associated process identifier. The process identifier then denotes a *service access point* which is the target of a message-encoded remote procedure call request.

4.5. Process Hierarchy

The actual structure of PEACE consists of several system processes, each one providing dedicated services to higher-level system/application processes. A hierarchical and process-oriented operating system structure is accounted. This structure identifies 10 functional layers, as shown in figure 3.

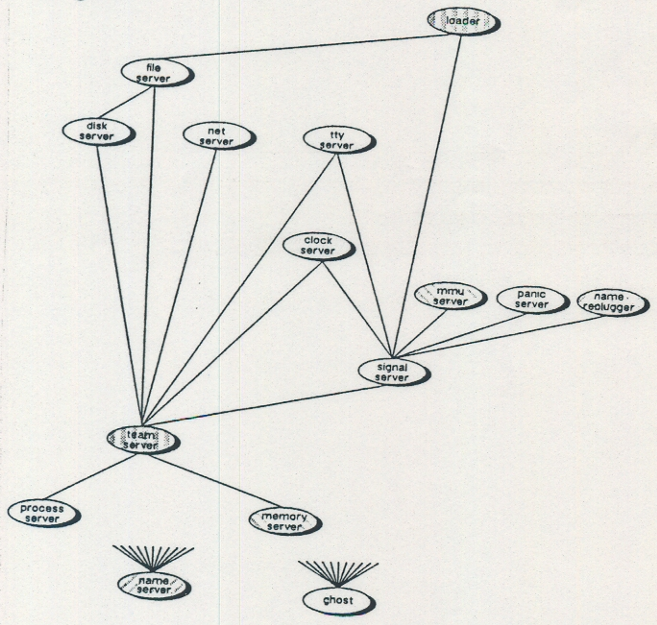

Figure 3.

Interactions between processes are based on remote procedure calls and, thus, being network-transparent. The actual structure of PEACE is expressed in terms of a *blocking graph*, as illustrated with the THOTH design. For the ease of illustration, the inter-relationships to the *name server* and the *ghost* are intimated, only, because all higher-level PEACE system processes apply level 1 and level 0 system services. In the following, the functionality of each of these system processes is explained, short.

The *ghost* assists the PEACE message-passing kernel in providing a scaled down remote procedure call interface. Level 0, i.e. the kernel team, of the PEACE system hierarchy represents the kernel space, which in turn means privileged mc68020 supervisor mode. All other system processes run in the so called team space, which in turn means non-privileged mc68020 user mode.

The *name server* provides basic naming services and allows for the manipulation of the mapping between a service name and the corresponding server. In PEACE this functionality is termed name replugging and is signaled as a system specific exception. Signaling this exception will be performed by the *name replugger*. Both processes are encapsulated by the same team, thus sharing the same knowledge.

The *process server* and *address space server* provide low-level and per-node process management and address space management functionalities. The *MMU server* handles MMU traps, propagated by the kernel, and therefore is placed side by side with the *address space server* in the same team.

The *team server* represents the interface to the application level for process and address space management. The *loader* does so for loading new programs or program fragments. Additionally, this system component controls termination of processes/teams and signals these events as well as creation of processes/teams in form of system specific exceptions. Both processes are encapsulated by the same team.

The *panic server* catches all non-served traps as propagated by the message-passing kernel and initiates the proper distribution of these exceptional events.

The *clock server*, the *tty server*, the *net server*, and the *disk server* are responsible for enhanced device management. Each of these processes is assisted by at least one device deputy, thus being able to receive messages, i.e. interrupts, from devices. The *file server* implements a UNIX-like file system interface. It is designed to allow for remote file access from any node.

The *signal server* enables propagation of specific exceptions relative to a UNIX-like process group, i.e. a job. Just as traps and interrupts are propagated on a message-passing basis, propagation/distribution of user/system defined exceptions, i.e. signals, is handled in the same way. Because message-passing is network-wide, signal propagation is also.

5. Performance Measurements

In this section, performance of the PEACE message-passing kernel is illustrated. Foundation for the measurements was a SUPRENUM prototype consisting of 5 nodes. Each node was equipped with a 16 MHz Motorola mc68020 with 1.5 wait states for each memory access. In addition to that, the cluster bus communication coprocessor interface was software emulated. The following subsections present the results of benchmark sequences executed on this SUPRENUM prototype. In [24] a more detailed analysis of these results is given.

5.1. Timing

Basically, communication performance of PEACE is determined by two separate mechanisms, namely message-passing and high-volume data transfer. In addition to this, timing parameters must be considered which are related to general operating system activities. For example, these parameters determine the overhead for context switches between user and supervisor mode, argument passing, interrupt management, process dispatching/scheduling, and so on.

In PEACE, the nucleus call overhead produced while switching from user to supervisor space, including argument passing, is accounted with 25 μsec. The same timing holds for the activation of device driver modules in case of incoming interrupts. Total interrupt management overhead, including synchronization and propagation activities and excluding device driver activities, takes 100 μsec. A user level process (context) switch is performed within 50 μsec. Not considering the nucleus call overhead, 25 μsec are accounted for the actual process dispatching activity within the PEACE nucleus.

A local message-passing activity distinguishes between intra-team and inter-team inter-process communication. Exchanging 64 byte fixed-size messages on a *send - receive - reply* basis takes 345/385 μsec for intra/inter-team communication, respectively. This timing includes three nucleus calls just as two dispatching activities, with a total overhead of 125 μsec. The 40 μsec difference between inter-team and intra-team message-passing performance is due to team scheduling/dispatching.

Presently, for remote message-passing activities two different network interfaces are available with the SUPRENUM prototype and are supported by PEACE. A *send - receive - reply* sequence over ETHERNET is accounted with 1.2 m sec, whereas 2.03 m sec are measured for the actual SUPRENUM cluster bus. That cluster bus communication is of relative low performance is due to a software emulation package which virtualizes the cluster bus communication coprocessor.

Although cluster bus communication presently is based on a software-emulated and low-level *word transfer protocol*, a net bandwidth of 3.2/4.9 Mbytes/sec for a *movefrom* and 2.7/4.6 Mbytes/sec for a *moveto* is accounted in case of exchanging 16/64 Kbyte segments over the cluster bus. The raw remote *movefrom/moveto* overhead is determined with 1.57/2.38 m sec, respectively. For a local activity, 80 μsec of overhead is accounted for each of these primitives.

5.2. Analysis

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 message-passing kernel 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. Considering the message-passing performance, this would mean a rendezvous timing of approximately 730 μsec. In a similar way, the *movefrom/moveto* setup timing is expected of being less than 565/856 μsec, respectively.

In the SUPRENUM prototype presented so far, inter-cluster communication is achieved by a PEACE message-passing kernel based on ETHERNET. This kernel performs a remote rendezvous, i.e. a *send - receive - reply* sequence, within 1.2 m sec, without being carefully tuned on the network driver level. A comparison of this kernel release with the fastest message-passing kernels (namely V and AMOEBA) which are based on similar processor architectures, i.e. various mc68020 host processors inter-connected by ETHERNET or ETHERNET-like networks, shows that PEACE is one fo the fastest message-passing systems [18].

6. Related Works and Concluding Remarks

Since the first SUPRENUM prototype, in the meantime a system configuration consisting of two clusters, with a total of 17 nodes, was introduced. In this system, the PEACE message-passing kernel and *name server* provides a runtime environment for an *ETHERNET server*, *UDP server*, *graphic server* and a distributed application computing fractale Mandelbrot sets. This application is structured into a total of 32 computing processes, distributed over 16 nodes. Each result of these processes either can be displayed on a graphic workstation or can be transmitted on a remote file access basis to UNIX.

Basing on this PEACE release, dynamical process and address space management services are tested, now. Following the pattern of [28] and [30], team migration concepts are developed for PEACE [22], which provide for a common basis for load balancing just as checkpointing and recovery. A hardware/software diagnosis system [6] is designed for SUPRENUM/PEACE maintenance purposes, respectively. Based on a similar approach as illustrated in [14], a monitor system is going to be developed for making on-line performance analysis, load balancing, remote procedure call monitoring and debugging feasible in PEACE. The PEACE communication system is expanded by inter-networking functionalities, especially with the maxim of maintaining network-transparency on a remote procedure call basis. The naming system is expanded, too, step-by-step as complexity of distributed/decentralized user/system applications will increase. And finally, a UNIX host interface is integrated into PEACE just as providing a UNIX-like system call interface.

With its fundamental design aspects, PEACE is comparable with V [5] and AMOEBA [17]. With respect to its message-passing performance, PEACE is superior to these systems [18]. Although V as well as AMOEBA are used within completely different environments than associated with PEACE, the fundamental synchronous message-passing principle common to each of these systems proves to be the most efficient one in an MIMD message-passing architecture. In addition to this, at least with V, just as AMOEBA, experiences have shown that synchronous message-passing is more fundamental and more efficient when compared to asynchronous message-passing, and that parallelism should be increased on the basis of light-weighted processes, instead of overloading a communication mechanisms. Because all these experiences, there is no doubt that PEACE is the appropriate process execution and communication environment for SUPRENUM.

Acknowledgments

Several excellent computer scientists and students contributed to the successful design and implementation of the PEACE prototype. Essential conceptual support came from Friedrich Schön and Winfried Seidel. Without the support of Jörg Nolte and Lutz Eichler, the network-oriented PEACE message-passing kernel would not exist at this point in time. An ETHERNET interface was implemented by Bernd Oestmann and Michael Sander, which made inter-networking experiments by Thomas Patzelt and UNIX access possible, at all. Last but not least, presently there is a residual crew of 12 excellent software engineers, all together working towards completing and maintaining PEACE.

References

[1] J. Backus: **Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs**, Comm. ACM, 21, 8, 613-641, 1978

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

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

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

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

[6] L. Eichler: **AUDIS - Automatic Diagnosis in the SUPRENUM System**, International Conference on Supercomputing, Saint Malo, France, July 4-8, submitted for publication, 1988

[7] 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

[8] W. H. Kohler: **A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems**, ACM Computing Surveys, Vol. 13, No. 2 (June), 1981

[9] 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

[10] 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

[11] B. H. Liskov: **Primitives for Distributed Computing**, Proceedings of the Seventh ACM Symposium on Operating Systems Principles, 33-42, 1979

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

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

[14] J. Mogul, R. Rashid, M. Accetta: **The Packet Filter: An Efficient Mechanism for User-level Network Code**, ACM Operating Systems Review, 21, 5, Proceedings of the Eleventh ACM Symposium on Operating Systems Principles, Austin, Texas, 1987

[15] H. Mühlenbein, O. Krämer, F. Limburger, M. Mevenkamp, S. Streitz: **Design and Rational for MUPPET - A Programming Environment for Message Based Multiprocessors**, Gesellschaft für Mathematik und Datenverarbeitung (GMD), 1987

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

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

[18] J. Nehmer: **Experiences with Distributed Systems**, Proceedings to be published in Computer Science Lecture Notes Series, Springer-Verlag, Workshop, Kaiserslautern (West Germany), Sept. 28-30, 1987

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

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

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

[22] F. Schön: **Migration and Fault-Tolerance in the Distributed PEACE Operating System**, to be provided, GMD FIRST an der TU Berlin, 1988

[23] W. Schröder: **Concepts of a Distributed Process Execution and Communication Environment (PEACE)**, Technical Report, GMD FIRST an der TU Berlin, 1986

[24] W. Schröder: **A Distributed Process Execution and Communication Environment for High-Performance Application Systems**, to be published in the Computer Science Lecture Notes Series, Workshop on "Experiences with Distributed Systems", Kaiserslautern (West Germany), Sept. 28 - 30, 1987

[25] C. L. Seitz: **The Cosmic Cube**, Comm. ACM, 28, 1, 22-33, 1985

[26] A. C. Shaw: **The Logical Design of Operating Systems**, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1974

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

[28] M. M. Theimer: **Preemptable Remote Execution Facilities for Loosely-Coupled Distributed Systems**, Ph.D. thesis, Technical Report STAN-CS-86-1098, Department of Computer Science, Stanford University, 1986

[29] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, R. Baron: **The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System**, ACM Operating Systems Review, 21, 5, Proceedings of the Eleventh ACM Symposium on Operating Systems Principles, Austin, Texas, 1987

[30] E. Zayas: **Attacking the Process Migration Bottleneck**, ACM Operating Systems Review, 21, 5, Proceedings of the Eleventh ACM Symposium on Operating Systems Principles, Austin, Texas, 1987