CONPAR 88
UMIST, Manchester, UK, September 12-16, 1988

The Distributed PEACE Operating System and its Suitability for MIMD Message-Passing Architectures*

Wolfgang Schröder

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

1. Introduction

Excellent communication performance of highly parallel MIMD multi-computer systems is predominant. These systems are by definition message-based and distributed systems. In order to make distributed systems work, a very careful and modular operating system design is required [10]. In addition to that, unless good performance is achieved, the resulting system will not be used. Therefore, a successful operating system design for large MIMD multi-computer systems –i.e. for distributed computer systems, in general– has to reflect both aspects.

The distributed operating system described in this paper consequently follows the concept of a "*family of operating systems*", as described in [14]. 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 SUPRENUM, a MIMD message-passing super-computer based on a distributed hardware architecture [1].

The purpose of this paper is to explain the fundamental PEACE design concepts and to give an excerpt of the overall PEACE operating system structure. As a supplement to the scope of this paper, a general overview of the SUPRENUM architecture is given in [5] and the illustration of SUPRENUM hardware building blocks is presented in [2].

2. Fundamental PEACE Concepts

In the following subsections a rationale for fundamental PEACE concepts is given. For this purpose, several operating system requirements are discussed which are specific to SUPRENUM.

2.1. Reduction of Software Overhead by a Careful System Design

As analyzed in [6] for communication networks based on ETHERNET [8], 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

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

specified by the functionality of a network or direct memory access controller. On the software level, this bottleneck is specified by network driver functionalities just as the semantics of inter-process communication primitives.

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, which is followed with the PEACE design, 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. In this sense, hardware requirements should be stated late in software system design.

2.2. A Family of Operating Systems

Reducing overall operating system software overhead is the major goal to improve communication performance for MIMD message-passing architectures. First of all, this overhead is a question of the actual operating system structure. Tuning the implementation of a specific system module is of secondary importance, because only in case of a well defined system structure, selective tuning (i.e. re-implementation) is possible.

The design principle of a *family of operating systems* shows how to build an operating system in which the most common and frequently used system modules are by definition free of software overhead. In PEACE, 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.

2.3. Process Structuring

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, such as PEACE, 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 [15], 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 the PEACE operating system fault-tolerant.

2.4. A Family of Communication Protocols

MIMD message-passing architectures are based on a communication network which interconnects 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, interconnected 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.

2.5. Achieving Concurrency on the Basis of Processes

Many MIMD application systems are based on the asynchronous model of inter-process communication, i.e. the *no-wait send* [7]. Several application programmers claim that this model is more fundamental, more efficient and increases parallelism [9]. 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* or *synchronization send* semantic [7], 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* [13]. This technique implies *remote invocation send* semantics. Implementing remote procedure calls on the basis of *no-wait send* increases communication protocol overhead and, thus, is to the debit of operating system as well as application system performance.

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 [12], 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 [3] 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 [18], which supports concurrency within MIMD programs on the basis of *multiple threads*. With PEACE, similar concepts and design decisions are given.

3. Design Decisions for the PEACE Operating System

Generally, 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. In [16], a more detailed discussion of the PEACE operating system structure can be found.

3.1. Light-Weighted Processes and Teams

In PEACE, operating system services are provided by processes. These processes, i.e. the services, are encapsulated by *teams*. Following the pattern of THOTH, 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 [12], 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.

3.2. Two Separate Communication Mechanisms

According to a *synchronous request–response* model of communication, 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 (*hvdt*) is performed. Each message exchange just as data transfer is network-wide.

As illustrated with V [4], this principle avoids temporary buffering of message/data streams within low-level communication systems. 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 inter-connection between peer teams, thus exchanging *hvdt access right*. 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.

3.3. A Small and High-Performance 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.

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 the means of 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.

3.4. System-Wide Unique Process Identification and Naming

One of the most critical aspects of distributed systems is how to identify and address processes (i.e. objects). Identification and addressing mechanisms directly influence inter-process communication performance – and performance is predominant in PEACE.

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, process*}. 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.

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: *node*, *cluster*, *hyper-cluster* and *system name plane*. 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. On the Suitability for Message-Passing Architectures

So far, experiences drawn from implementing and running the first PEACE prototype completely confirm the overal PEACE design for a MIMD machine like SUPRENUM. The most important aspects covered by the prototype implementation are very high performance, good modularization and flexibility.

The PEACE message-passing performance, i.e. a complete *send-receive-reply* sequence, takes about 385 µsec. for local inter-team communication. For remote inter-team communication, a 10 MBit ETHERNET configuration takes 1.2 msec. and a configuration based on a SUPRENUM cluster bus coprocessor software emulation takes 2.03 msec. Using the final firmware-based communication coprocessor, it is expected to perform remote inter-team

communication in less than 750 µ sec. In [17] a more detailed analyzation of PEACE prototype parameters can be found.

These excellent performance parameters for a prototype implementation are the consequence of the design principle followed with PEACE. The most fundamental PEACE component, the message-passing kernel (more specifically, the *nucleus*), only is concerned with communication aspects. Most essentially, however, is the fact, that this design principle made it feasible at all to develop and complete an upward expandable process execution and communication environment for a distributed hardware environment, namely for SUPRENUM. At the one end, very early it was possible to drive first experiments, thus making first experiences with distributed systems in general and SUPRENUM in particular. At the other end, further system functionalities can be added, in terms of high-level and application-oriented system processes, without loosing any communication performance.

5. Concluding Remarks

With its fundamental design aspects, PEACE is comparable with V and AMOEBA. With respect to its message-passing performance, PEACE is superior to these systems [11]. 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] P. M. Behr, W. K. Giloi, H. Mühlenbein: **Rationale and Concepts for the SUPRENUM Supercomputer Architecture**, Gesellschaft für Mathematik und Datenverarbeitung (GMD), 1986

[2] P. M. Behr, S. Montenegro **The SUPRENUM Node-Computer**, CONPAR 88, Manchester, UK., 12th-16th September, 1988

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

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

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

[6] 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

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

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

[9] 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

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

[11] S. J. Mullender: **The AMOEBA System - a Retrospect**, International Workshop on "Experiences with Distributed Systems", Kaiserslautern (West Germany), Sept. 28-30, 1987

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

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

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

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

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

[17] W. Schröder: **A Distributed Process Execution and Communication Environment for High-Performance Application Systems**, Lecture Notes in Computer Science, 309, J.Nehmer (Ed.), International Workshop on "Experiences with Distributed Systems", Kaiserslautern (West Germany), Sept. 28 - 30, 1987

[18] 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