Parallel Computing 7 (1988) 325–333
North-Holland

PEACE: The distributed SUPRENUM operating system

W. SCHRÖDER

Gesellschaft für Mathematik und Datenverarbeitung mbH, GMD FIRST an der TU Berlin, D-1000 Berlin 12, Fed Rep. Germany

Abstract. This paper describes the fundamental concepts and structure of the distributed operating system, PEACE, for SUPRENUM. A large scale of distribution is achieved because of consequently encapsulating typical operating system services by processes. By this way an optimal and application-oriented mapping of the entire operating system onto the distributed SUPRENUM architecture is made feasible.

Keywords. SUPRENUM architecture, operating system PEACE, design aspects, communication performance, prototype implementation.

1. Introduction

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. Rationale and fundamental concepts of SUPRENUM are illustrated in [1], and in [6] the SUPRENUM architecture is explained. 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 the appropriate distributed operating system for the SUPRENUM processor kernel.

This paper describes a distributed '*process execution and communication environment*' for SUPRENUM which is called PEACE. From the software point of view, PEACE follows the design principle of a family of operating systems as motivated in [10]. The major foundation of PEACE stems from THOTH [3] and MOOSE/AX [12]. With respect to the distributed organization of PEACE, the main influences came from V [4] and AMOEBA [9]. Using processes as building blocks, which encapsulate dedicated functionalities and/or services, PEACE is qualified as an application-oriented and high-performance distributed operating system for SUPRENUM.

In the following sections a short overview of the basic PEACE system structure is presented. In Section 2 the PEACE operating system structure is illustrated and fundamental concepts are described. Section 3 explains the PEACE communication system and shows by what means the PEACE message-passing kernel works on a network-oriented hardware architecture, i.e. SUPRENUM. Some performance measurements are given in Section 4 and a final presentation of lessons learned during design and implementation of PEACE, given in Section 5, concludes this paper.

2. Design aspects

The following subsections are concerned with a brief illustration of fundamental PEACE concepts which serve as the basis for the implementation of an appropriate process execution and communication environment for SUPRENUM applications. For this purpose, the PEACE operating system structure is introduced and an excerpt of the functionality of the different operating system layers is given.

2.1. 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 1 and are described in [13], in more detail. With each layer of the PEACE operating system a set of server processes is associated. These server processes are responsible for the implementation

Table 1
PEACE operating system layering

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-pasing 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

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 [2]. With each remote procedure call entry an own service function is associated. 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 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 within 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 high-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.

2.2. Light-weighted processes

As introduced with THOTH, processes in PEACE are associated with *teams*. A PEACE team specifies a common execution and scheduling 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.

2.3. Synchronous request-response model of communication

The communication environment for PEACE processes 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. A 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, a cluster bus word. In addition to this, the data segment is restricted to be data item aligned.

3. Problem-oriented communication system

The following subsections are concerned with a functional description of fundamental PEACE communication and management protocols, especially the substantial design decisions are presented. A more complete explanation of these protocols is given in [5].

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.

3.1. Reducing buffer management overhead

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, i.e. for remote processes no virtual representation, on the basis of appropriate state information, in 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.

Potentially, buffer management is necessary if a client process requests a rendezvous from a remote server process by applying the *send* primitive. A receive request message is transmitted to the remote server's nucleus and the client blocks until a *reply* is issued. At the remote site, buffering within the nucleus would be necessary if the server process is temporarily unable to accept, i.e. receive, the incoming message. If client and server process both reside on the same

host, the client process simply is queued in the server's sender list. This works straightforwardly, because the resource 'client process' is known to the nucleus. If both processes reside on different hosts, either the resource 'client process' or 'server process' is unknown to the nucleus, respectively.

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, because a number of clients again and again will retry sending the receive request message to the same remote server, 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. This 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 is exhausted and a client state is not remembered, then the receive is usually to be rejected, too. In PEACE, the buffer pool at the server site exactly consists of one buffer for each process.

3.2. The use 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 exclusively can be 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.

3.3. Inter-connecting peer address spaces

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 [11]. Second, todays transmission media, especially for local area networks, are of high quality and loss of data packets is not only a problem because of transmission errors. There is also a significant packet loss because of operating system buffer management problems and network interface capabilities [7]. 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 [1].

The PEACE *data transfer protocol* follows the principle of a *blast protocol* [14]. The main purpose of this 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 [8], 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.

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.

3.4. Identifying peer processes

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. 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* and not the service encapsulating process—although in most cases it directly represents the server process, by default.

Names exported by processes constitute the PEACE *name space*. Actually, this name space is structured into one or more *name planes*, according to the SUPRENUM architecture. 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.

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 is 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 is hierarchically structured, thus building a *name tree*. The leaves of this tree are represented by name servers, i.e. name planes. The coupling between different subtrees is accomplished by dedicated system processes, so-called *domain server*. According to the four SUPRENUM name planes, the PEACE name tree consists of four layers. In order to locate PEACE services, the domain server applies a

sequential search strategy, from bottom up, and issues name lookup requests to the various name server processes.

4. A prototype case study

In order to early provide a process execution and communication environment for distributed SUPRENUM applications, 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. On this software basis, the communication performance for local as well as remote operation was measured. The results stress the high-performance of the PEACE communication primitives. In the following subsections a brief discussion of some performance parameters is presented.

4.1. Interface penalty

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 communication system. 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.

The *work transfer 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 μs if a single *dispatching protocol* packet is sent to peer protocol entities, not counted other nucleus management activities as interrupt handling and synchronization, process/protocol state observation, context switching and user/nucleus switching. Presently, this overhead determines the per-node *interface penalty* of a single message setup for the transfer of each one of these packets.

Note that this actual interface penalty should always be kept in mind if the communication performance of the PEACE message-passing kernel is assessed. Avoiding this 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. However, for general comparison purposes the actual timing on basis of this emulation package is valuable, also.

4.2. Message-passing performance

For local operation, message-passing performance differentiates with respect to intra-team and inter-team communication. For intra-team message-passing 345 μs and for inter-team message-passing 385 μs is accounted. The difference is due to team selection and dispatch overhead in order to execute a team switch.

For remote operation, only intra-cluster communication is available, now. A performance of 2.03 ms is accounted, including the additional interface penalty. First analysis show that a remote rendezvous timing of less than 730 μs is possible on basis of the actual PEACE nucleus implementation. Using a 10M6 Ethernet instead of the cluster bus prototype, PEACE performs inter-node message-passing within 1.2 ms.

4.3. High-volume data transfer performance

The PEACE message-passing primitives are applied in order to achieve a direct inter-connection between peer address spaces on a rendezvous basis, maybe crossing node boundaries.

During a rendezvous, the primitives *movefrom* and *moveto* are applicable, actually enabling end-to-end high-volume data transfer without any need of buffering within the communication system.

For local operation, the raw overhead of *movefrom* and *moveto*, i.e. user/nucleus switch and rendezvous verification, is about 80 μs. The transfer of 8 bytes, i.e. a single cluster bus unit, takes 90 μs and 64 bytes are transferred within 105 μs. A page, with a size of 4096 bytes, is transferred within 950 μs.

For remote operation, the performance of *movefrom* differentiates from that of *moveto*. The reason is one more *dispatching protocol* packet to be sent for an explicit rendezvous verification request. With *movefrom* 1.57 ms and with *moveto* 2.38 ms raw overhead are accounted. The transfer of 8 bytes takes 1.61 ms for *movefrom* and 2.42 ms for *moveto*, whereas for 64 bytes 1.63/2.44 ms are accounted, respectively. A page is transferred within 2.57/3.18 ms.

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. Generally, avoiding the interface penalty, a raw *movefrom/to* timing less than 565/856 μs is expected, respectively. More specifically, the transfer rate of single cluster bus units at least is comparable with that of local direct memory-to-memory transfers.

5. Concluding remarks

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, a very high degree of decentralization and/or distribution of the PEACE operating system was achieved. There is no doubt that this design principle significantly promotes project-oriented system development. Without functional simplicity 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, described in [10], 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.

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 [7], *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 and 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 by a careful system design and the second step, if at all, might be the migration of protocol functionalities into low-level hardware. As a consequence, synchronous message-passing and separate mechanisms for high-volume data transfer found the adequate basis for inter-process communication in PEACE. Within the operating systems area it is widely accepted that this approach encourages the implementation of high-performance communication systems. With V [4] an exemplary system design and implementation has been presented; and PEACE follows this pattern.

Compiler quality has significant effect on the overall performance of SUPRENUM, maybe this is the most important aspect at all. Dependent on the underlying processor, alignment of data objects significantly influences the overall system performance. To give an example, the

local inter-team rendezvous timing of the PEACE nucleus was improved by 13%, from 440 μs to 385 μs, once proper mc68020 stack pointer alignment, in 32-bit units, was ensured. Ensuring message buffer alignment, a performance gain of approximately 5% was achieved. In addition to that, using a compiler providing for embedded register allocation/deallocation features, a peak performance gain of 43% was achieved for the nucleus implementation. Indeed, it is a well-known story that system performance rises and falls with compiler and/or programming language quality. In each case, the PEACE design is optimal for SUPRENUM and reflects the state of the art in operating systems design. Experiences with the first PEACE prototype show that performance bottlenecks within the implementation will be especially due to compiler and/or programming language quality.

Acknowledgment

Presently, the distributed PEACE kernel runs on a SUPRENUM cluster. This was only possible because of excellent team work of the SUPRENUM software crew. From the large list of the PEACE designers and programmers, L. Eichler, J. Nolte, B. Oestmann, Th. Patzelt, F. Schön, and W. Seidel essentially influenced the actual PEACE design and made the prototype implementation feasible, at all.

References

[1] P.M. Behr, W.K. Giloi and H. Mühlenbein, Rationale and concepts for the SUPRENUM supercomputer architecture, Gesellschaft für Mathematik und Datenverarbeitung (GMD), 1986.

[2] A.D. Birrell and B.J. Nelson, Implementing remote procedure calls, *ACM Trans. Comput. Systems* **2** (1) (1984) 39-59.

[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 and W. Zwaenepoel, The distributed V kernel and its performance for diskless workstations, *ACM Operating Systems Rev.* **17** (5), *Proc. 9th ACM Symposium on Operating Systems Principles*, Bretton Woods, NH (1983).

[5] L. Eichler, J. Nolte, T. Patzelt, F. Schön, W. Schröder and W. Seidel, Communication- and management protocols for the distributed PEACE operating system, Technical Report, GMD FIRST an der TU Berlin, 1987.

[6] W.K. Giloi, SUPRENUM: A trendsetter in modern supercomputer development, *Parallel Comput.* **7** (1988) 283–296, this issue.

[7] K.A. Lantz, W.I. Nowicki and 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.

[8] R.M. Metcalfe and D.R. Boggs, Ethernet: Distributed packet switching for local computer networks, *Comm. ACM* **19** (7) (1976) 395-404.

[9] S.J. Mullender and A.S. Tanenbaum, The design of a capability-based distributed operating system, *Comput. J.* **29** (4) (1986).

[10] D.L. Parnas, On the design and development of program families, Forschungsbericht BS I 75/2, TH Darmstadt, 1975.

[11] J.H. Saltzer, D.P. Reed and D.D. Clark, End-to-end arguments in system design, *ACM Trans. Comput. Systems* **2** (4) (1984) 277-288.

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

[13] W. Schröder, Concepts of a distributed process execution and communication environment (PEACE), Technical Report, GMD FIRST an der TU Berlin, 1986.

[14] W. Zwaenepoel, Protocols for large data transfers over local networks, *Proc. 9th IEEE Data Communication Symposium* (1985).