PEACE: A Distributed Operating System for High-Performance Multi-Computer Systems[*]

Wolfgang Schröder

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

ABSTRACT

The design of the distributed PEACE operating system is explained. Functionality and structure of the PEACE operating system is mainly influenced by the distributed and message-based SUPRENUM supercomputer architecture. In order to achieve a large scale of decentralization, the operating system is consequently structured as a set of cooperating and communicating processes. The inter-connection between the various PEACE processes is realized by a small and high-performance message-passing kernel, the most fundamental system component of PEACE. The concept of light-weighted processes is adopted in order to provide store-and-forward functionalities on gateway nodes.

1. Introduction

The PEACE operating system represents a versatile virtual machine for SUPRENUM [Behr et al. 1986]. It is this virtual machine that hides the fact of a complex multi-computer system from the application level. On the other hand, the network architecture of SUPRENUM is made visible and, thus, enabling the application level to take advantage of specific architectural features. Thus, network transparency is achieved as well as dedicated and network oriented applications are supported. In [Schroeder 1986] the basic concepts of PEACE are illustrated – actually, this paper is a summary of these concepts.

Network transparency in PEACE is realized by a specific set of system components. These components widely follow the design principle in PEACE of a process structured operating system. Distinct system processes are used to support efficient inter-cluster communication. In this paper, fundamental PEACE system components are explained. The operating system structure is shown and essential system concepts of PEACE are discussed.

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

2. Network Architecture

SUPRENUM is a multi-computer system which is based on two different inter-connection systems. The one inter-connection system is given by the so called cluster bus which inter-connects upto 20 nodes. These nodes are based on the mc68020 and together form a cluster. The cluster bus is a high-speed parallel bus and supports transmission speeds of upto 128 Mbytes/sec. The other inter-connection system is given by the SUPRENUM bus which connects a couple of clusters. The SUPRENUM bus is a slotted-ring bit-serial bus with a transmission rate of upto 25 Mbytes/sec. With this inter-connection system as the basis so called hyper-clusters are constructed by connecting a couple of clusters on a row/column basis. A cluster represents the SUPRENUM hardware building block for the construction of a specific multi-computer system with a performance to any extend desired by a customer. See [Behr et al. 1986] for more detail.

In order to allow flexible inter-connection of clusters, in each cluster a qualified node is identified for that task. This node, termed the SUPRENUM communication node, acts as a gateway, i.e. it bridges the two different SUPRENUM inter-connection systems. Additionally, specific communication nodes are connected to a host, running UNIX[†] on top of it. In contrast to that, the PEACE operating system runs on top of the SUPRENUM processor kernel, only, which is constituted by the cluster matrix. The logical inter-connection between UNIX and PEACE is achieved by a PEACE "guest layer" on top of UNIX.

The SUPRENUM architecture identifies three host computers, each one associated with some specific controlling task. Figure 2.1 shows a SUPRENUM configuration consisting of 16 clusters, with a total of 320 nodes, and the 3 hosts.

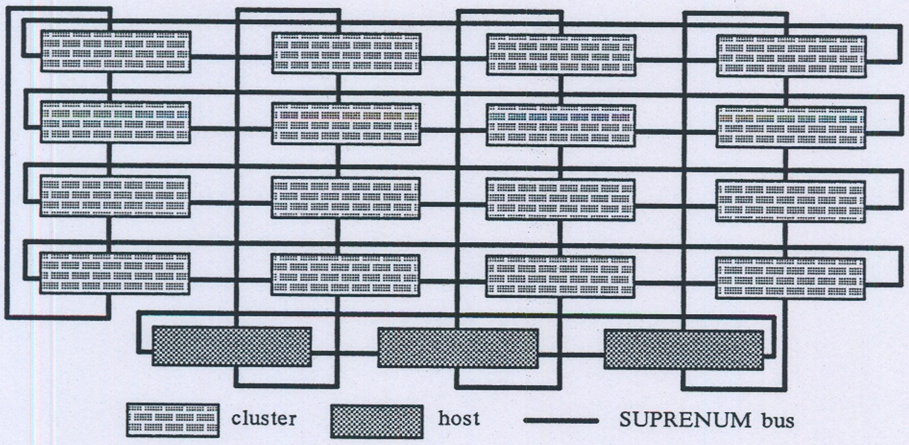

Figure 2.1: SUPRENUM Hardware Architecture

This specific network architecture requires a very efficient communication model with respect to maximal network performance. In order to fulfil this major requirement, a

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

high-performance message-passing kernel, specifically tuned for operation within network environments, founds the basis for the PEACE operating system. The comparison to other message-passing systems, particularly V [Cheriton, Zwaenepoel 1983], AMOEBA [Tanenbaum, van Renesse 1985] and AX [Schroeder 1986b], shows that the PEACE kernel takes the leading position with respect to its message-passing performance. A local message interaction, expressed by a *send receive reply* sequence and based on 64 bytes fixed-size messages, takes about 400 μsec. The remote communication model benefits from the runtime efficiency associated with the local inter-process communication mechanisms. A prototype implementation for ETHERNET [Metcalfe, Boggs 1976] performs a remote message interaction within 1.2 msec. Presently, a bulk data transfer rate of 4/5 Mbytes/sec over the cluster bus is achieved for 16/64 Kbyte message segments and based on a cluster bus protocol software emulation package.

3. Fundamental Concepts

The design of PEACE was mainly influenced by THOTH [Cheriton 1979]. The slogan for PEACE is process structuring, in order to construct the operating system, and synchronous message-passing on a *send-receive-reply* basis, as the fundamental inter-process communication mechanism. In addition to that, processes are encapsulated by THOTH-like teams, i.e. adopting the concept of *light-weighted processes* sharing the same address space. In order to provide for some means of asynchronous and bulk data transfer, a separate mechanism is provided on the basis of THOTH-like *movefrom/moveto* primitives.

With respect to the network-oriented nature of PEACE, the concepts of V [Cheriton 1984] and AMOEBA [Mullender, Tanenbaum 1986] especially influenced the design of the message-passing kernel. This means that all message-passing and bulk data transfer activities are network-wide.

The fundamental ideas about how to hierarchically structure an operating system are taken from [Parnas 1975] and [Habermann et al. 1976]. With MOOSE [Schroeder 1986b] it has been exemplified how to construct a hierarchically and process structured operating system in an application-oriented manner. The basic mechanisms of MOOSE, especially the idea of consequent service structuring and service encapsulation by processes, are used to allow for the dynamical reconfiguration of PEACE. These facilities are fundamental to the distribution of applications running on top of PEACE and of the PEACE operating system itself.

4. Degree of Distribution

Due to the SUPRENUM hardware architecture at least a decentralized operating system is required. In order to allow for interactions between peer user and/or system components, it suffices to use the remote procedure call (RPC) paradigm as described in [Nelson 1982]. With respect to system services, there is no cogent necessity for a really distributed PEACE, e.g. by replication of primary data structures of the operating system.

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

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

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

5. Degree of Fault-Tolerance

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

The fault-tolerance of the system is implemented by system processes which provide proper functionalities (e.g. checkpointing and recovery). The *nucleus* and very few low-level system processes of PEACE are the only system components which must reside on each SUPRENUM node. The functionality of these components is basic inter-process communication and low-level aspects of naming, process creation/destruction and address space management. Other typical operating system functionalities are provided by high-level system processes, which may be located on a specific subset of all available nodes, only. Examples of these functionalities are management, control and loading of jobs, deadlock detection and resolving, scheduling, propagating and handling of system specific exceptions, file handling, i/o management, etc.

In PEACE, each system component is given its own address space, thus building "fire walls" around them. With that design decision a basis for implementing secure systems is given, because potential malfunctions of system components, e.g. the destruction of address spaces, are mostly of local nature. More specifically, the design of PEACE follows the idea of a security kernel, thus, once a formal specification is given, providing the chance of verification of the most crucial operating system components [Millen 1976].

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

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

6. Functional Hierarchy

In the following the minimal set of PEACE system components is introduced. These system components represent functionalities as typically discovered in other modern operating systems. Figure 6.1 shows the actual hierarchical structure of PEACE, which identifies 10 levels (from bottom-level 0 upto top-level 9). This figure reflects the PEACE operating system structure in terms of process/team inter-relationships.

6.1. Basic Components

The *nucleus* (not shown in this figure, because it is no process/team) is the most basic system component and defines the bottom level of the PEACE system structure. The main functionalities are inter-process communication based on synchronous message-

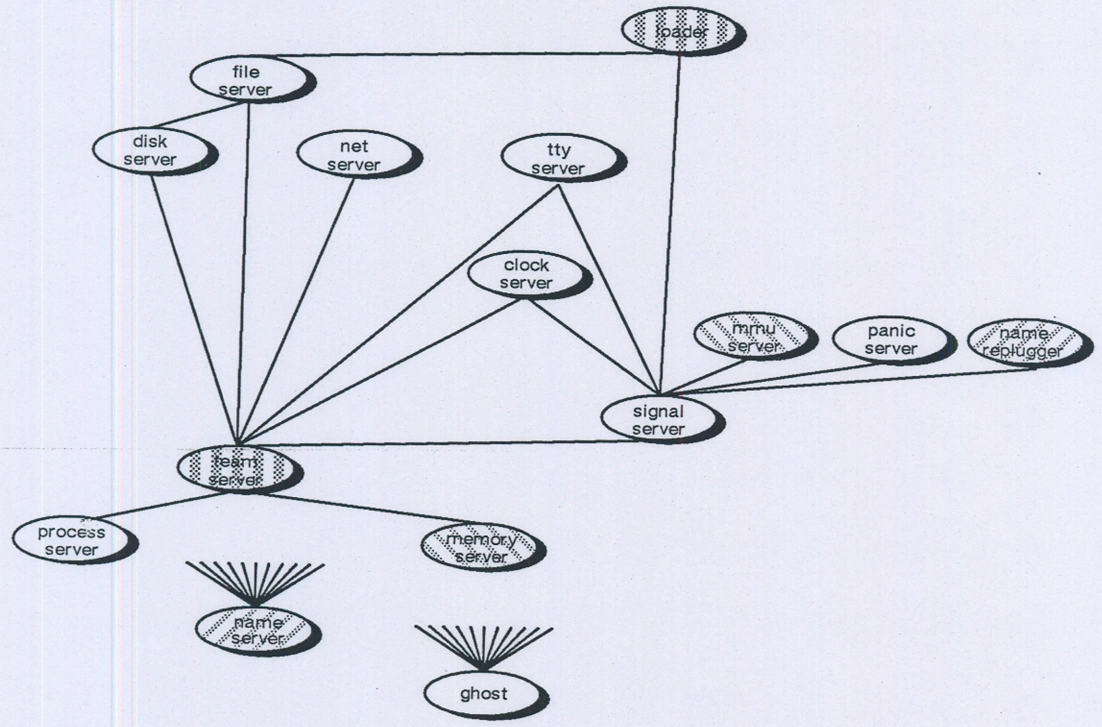

Figure 6.1: The Actual Structure of PEACE

passing, process dispatching based on the concept of light-weighted processes and teams, address space switching and, finally, trap/interrupt propagation as based on the principles discovered by MOOSE [Schroeder 1986b].

The *nucleus* is assisted by the *ghost*, whose task it is to provide an RPC interface for the message-passing kernel. Level 0 of the PEACE system hierarchy represents the *nucleus 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. Signalling this exception will be performed by the *name replugger*. Both processes are encapsulated by the same team, thus sharing the same knowledge, and together represent the PEACE name administrator.

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 and therefore is placed side by side with the *address space server* in the same team. These two processes constitute the address space administrator.

All system processes on level 2 and above uses *name server* as well as *nucleus* services.

6.2. Constructive Components

The *nucleus* as well as all system processes mentioned so far are placed on each SUPRENUM node. The following system processes are the entities in PEACE for arbitrary distribution over the SUPRENUM network architecture. 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 it controls the 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 and represent the PEACE team administrator.

The *panic server* catches all non-served traps as propagated by the *nucleus* 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. They represent the so called device administrators of PEACE.

The *file server* implements a UNIX-like file system interface. It is designed to allow for remote file access from any node. In case of remote file access, on any node remote from the *file server* node a *file deputy* acts as the local *file server* representative. The *file deputy* and the *file server* together constitute the file administrator.

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

7. Signal Distribution

Exceptions in PEACE are represented by expedited messages, so called signals, whose transmission and reception is guaranteed by the system. Because of the message-oriented nature of signal propagation in PEACE, there is a specific model necessary. This model stems from MOOSE and is shown in figure 7.1. In general, there are four processes involved with signal propagation. On the raising side, the *signal notifier* raises or notifies the occurence of an exception. On the receiving side, the *signal client* serves and awaits propagated signals, originally dedicated to the *signal applicant*. In general, *signal applicant* and *signal client* both are encapsulated by the same team. On the propagating side, the *signal server* directs and distributes raised signals to all clients which requested to be informed about the occurence of a specific exception. The *signal server* and the *signal client* both constitute the PEACE signal administrator.

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

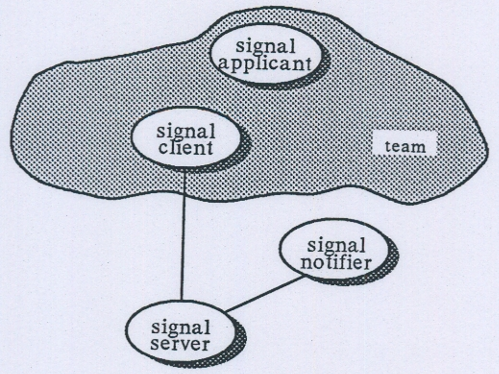

Figure 7.1: Propagation of Signals

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

The existence of the *signal client* is motivated due to the fact, that signals must be propagatable to a specific application process (in this model represented by the *signal applicant*) independently of its actual processing state. Specifically this is ensured in PEACE on the basis of the light-weighted *signal client* process. Because of the fact that signals are represented by messages and that *signal clients* represent the receiving process of a propagated signal, signal propagation is possible in PEACE across node boundaries.

7.1. Multi-Casting

Basically, signal propagation in PEACE means multi-casting an expedited message. This multi-cast mechanism is based on a team group identification associated with a couple of *signal clients* and/or *signal applicants*. A signal, raised by the *signal notifier*, is bound to a team group identification. All *signal clients* belonging to this team group may receive the same signal, respectively the same message.

For the PEACE operating system, signal multi-casting is a fundamental technique for the distribution/notification of system specific events. Examples for such PEACE events are termination/creation of processes and detaching services from server processes, as done in conjunction with process migration and service replugging just as with replacement and reconfiguration strategies for achieving fault-tolerance.

One of the significant applications of multi-casting in PEACE is checkpointing, consistence checking of specific data structures, test pattern generation and recovery. A fault-tolerant application in PEACE uses the *signal client* model for the encapsulation of application-oriented management activities, triggered at the responsibility of dedicated system processes, in PEACE e.g. a *checkpoint server*. For example, this system process controls writing of application dependent checkpoints and directs the *signal client* to actually write out critical data objects of its team. The team itself is part of the application to be checkpointed and it is bound to a unique team group for that application. Initiating checkpointing would lead to reactivation of all the *signal clients* acting as *checkpoint clients* of the corresponding team.

8. Remote Service Invocation

All interactions between user and system processes in PEACE are controlled on an RPC basis. Because services are the focus of PEACE operating system activities, this basis allows for a remote service invocation. The PEACE RPC model closely follows the pattern of [Birrell, Nelson 1984].

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

Generation of the protocol data unit as well as "marshalling", i.e. generation of code for the definition and interpretation of a protocol data unit, is done automatically. Given an interface specification of the requested service function, a PEACE utility arranges the necessary steps for generating a framework to cope with the definition and interpretation of RPC protocol data units. In terms of [Nelson 1982] and applied to the PEACE service model, this framework is represented by the *service stub* on the client's side and by the *call stub* on the server's side[1)].

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

The principle of abstract data types, as e.g. exemplified by [Liskov, Zilles 1974] strongly is followed with the service interface design in PEACE. Not only high-performance is achieved when invoking a service. More importantly is the fact of abstraction from the way a service is encoded by a message, the way the corresponding server is identified by asking the *name server*, the way the service is decoded from the received message and, finally, the way the client/server interaction is realized by a specific communication protocol and system. It is this mechanism of remote service invocation which enables the network transparency in PEACE. For example, service replug exceptions are served on the RPC stub level, thus hiding the corresponding *signal*

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

client from the application level. Handling this kind of system specific exceptions within the application context, always can guarantee direct addressing of service providing processes.

9. Naming and Addressing

The naming service of PEACE is provided by the *name server* and the *name replugger*. The named objects are the services a process provides to other processes. On behalf of a server, these services are associated with arbitrary character strings and made known to the *name server*. Moreover, a server may remove a service name from the *name server*. In this case, a name release exception is signaled by the *name replugger* to all clients connected to the service represented by the released service name.

9.1. Service Representation

Along with a service name, the *name server* remembers two keys. The one key represents a service access point [Schindler 1980] which gives the way to the server associated with the service name. The other key is the per-server internal representation of the service. Actually, this key is generated automatically by the *call stub* of the corresponding server. It is used solely by the *service stub* and the *call stub* to encode and decode the service function, respectively.

The service access point actually is represented by a process identification. In case of a local provided service, this process identification directly designates the real server. In case of a remote provided service, however, the process identification either may designate a process which merely routes the service request to the actual (remote residing) server, or it may directly designate the remote residing server. In order to efficiently distinguish a remote request from a local one, the process identification is structured as 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 light-weighted task which actually represents the process.

The client logically addresses some server by a service name and receives an identification about what route to take for the service request. The client is not aware about the physical position of the service providing process. The information returned from the *name server* enables the *service stub* to uniquely address a service primitive by the tuple (*server,service*).

9.2. Name Planes

The SUPRENUM architecture naturally leads to a hierarchically structured name space. This name space is composed by different name planes. The lowest (innermost) name plane is defined by a single node. The next encapsulating name plane is that of a cluster. The hyper-cluster name plane corresponds to a row/column of clusters and, finally, the outermost system name plane embraces all hyper-cluster name planes.

Locating the server from a given service name means observing the different name planes from bottom-up. More precisely, as with usual scope rules discovered in many programming languages, a service name of the innermost name plane overrides the same service name of some outermost name plane. Thus, services are addressed in a certain neighbourhood fashion. Apart from the fact that this principle allows for a high degree of flexibility, it is an important one to achieve the best utilization of the SUPRENUM network performance. As already mentioned, the cluster bus is about a factor of 10 faster than the SUPRENUM bus. The more communication activities must be performed over the SUPRENUM bus, the more the overall system performance will drop. Because this inter-connection system combines a couple of clusters to hyper-clusters, using a service from the hyper-cluster name space is more expensive than using one from the cluster name space.

Setting up different name planes is performed on behalf of server processes by instructing qualified *name servers* of the system. Basically, each name plane is encapsulated/implemented by a single *name server*. Service names may be explicitly associated with a certain name plane, thus binding the name to a specific scope. For example, crossing node boundaries is only possible if a service name is known by a cluster controlling *name server*. The same holds for clusters and hyper-clusters. Thus, inter-node/intra-cluster service invocation is supported by a proper service name associated with the cluster name plane.

10. Inter-Node Communication

In PEACE, inter-node communication is based on a model with minimal process overhead. It is important to note that the actual underlying network hardware architecture does not influence the functionality associated with that model. A virtual network device, implemented by the *nucleus* and *ghost*, realizes the abstraction from the specific physical characteristic of the networking hardware.

10.1. Remote vs. Local Server

It is a principle in PEACE that a server, which provides remote accessible services, at least is constituted by two light-weighted processes encapsulated by the same team. The one process, the *local server*, is responsible for receiving and processing local service requests, whereas the other process, the *remote server*, is responsible for receiving and processing remote service requests. In either case, the requested service (coded in the message) immediately may be processed, because *local server* and *remote server* reside within the same server team. Moreover, there is no lost in efficiency while processing a service, because services are still received on a client-by-client basis without the necessity of any specific client/server synchronization. The *local server* only executes if the *remote server* does not, and vice versa. Each process will block if there are no more requests pending, thus, due to the dispatch strategy locally to one team and enforced by the *nucleus*, enabling the execution of the other, non-blocked process.

10.2. Invoking Remote Services

In order to complete the scenario of inter-node inter-process communication in PEACE, the client side is considered in the following. In general, a client is unaware about the localization of the server. In case of a remote residing service, the corresponding service access point is represented by a *remote server*. Sending a message-encoded service request to such a service access point, leads to the activation of the low-level PEACE network communication system directly implemented by the message-passing kernel.

Distinguishing a remote service from a local one is done once the service provider has been asked for by instructing the *name server* to locate the corresponding service name. The process identification delivered by the *name server* designates the actual server for the requested service, independently of the server's node membership. In case of a remote residing server, the host member of its process identification is different from that of the client's process identification. The *remote server* process identification always is returned to a client which is remote from the service provider. In contrast to that, the *local server* process identification is returned to a client which is local to the service provider.

11. Inter-Cluster Communication

A general problem arises with remote inter-process communication if network boundaries must be passed. In this case, typically, a network gateway must be installed, whose responsibility it is to shift all communication activities between different network architectures. This means an additional overhead, because converting protocol data units [Lam 1986] as well as programming of other network devices will be necessary. Moreover, it may be required to drive another protocol and to perform some other time consuming management activities as, for example, buffering of data items. Actually with that situation the overall communication system for SUPRENUM is confronted.

Inter-cluster communication in PEACE means driving the SUPRENUM bus. In order to compensate the lower transmission speed of the SUPRENUM bus, a *store-and-forward* model is adopted on each communication node. Figure 11.1 explains this model, in short.

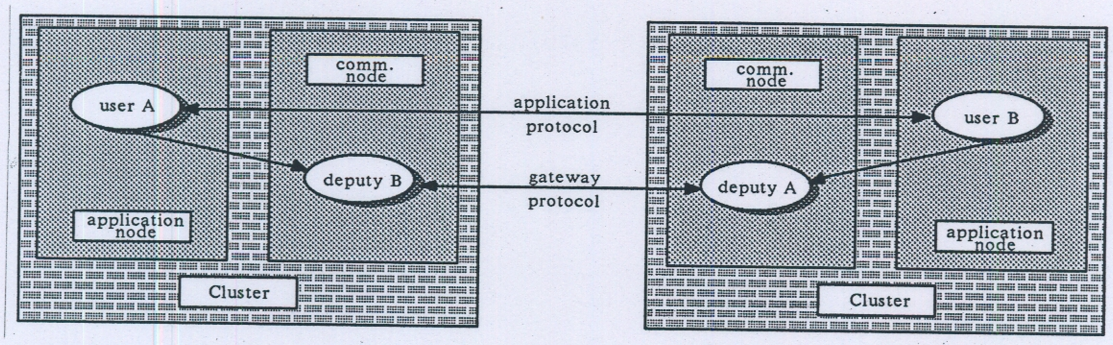

Figure 11.1: Inter-Cluster Communication using Deputies

The fundamental idea behind this model is to use a so called deputy process/team which represents the original peer application process/team. In case that the application is distributed over several clusters, a deputy of the peer application process/team is placed on all communication nodes on the route between the original peer application processes/teams. The sole function of this deputy is to store and forward messages (logically) exchanged between the peer application processes. The advantage of this technique is that the message setup time for inter-cluster communication is identical to intra-cluster communication, because there is always a cluster-local deputy for a cluster-remote application process/team. As long as there is enough buffer capacity on the communication node, from the application point of view inter-cluster communication is performed with the same speed as intra-cluster communication.

12. Communication Protocol

Performance is the major requirement associated with inter-node and inter-cluster communication. The main goal is to achieve the maximal utilization of the physical communication bandwidth as specified for the SUPRENUM inter-connection architecture. In order to fulfil that requirement to a great extend, from the communication software point of view very optimistic assumptions about the underlying network hardware must be met. Especially, this means that time consuming recovery procedures, for example given by complex retransmissions strategies, must be excluded from the communication protocol design, as far as possible.

In [Zwaenepoel 1985] measurements about several communication protocols, well suited for high-performance local area networks, are given. Although these measurements are based on 10 Mbit ETHERNET interfaces, it can be taken as a basis for deciding what kind of communication protocol will be the best for SUPRENUM. As described in [Lantz et al. 1985], significant loss of performance was given because the need for transferring data by the central processing unit from/to network interfaces (the ETHERNET controller). As a consequence of these observations, the major design goal for SUPRENUM communication protocols should be to avoid time consuming and host-bounded management activities as, for example, dynamical memory and/or buffer management.

Maximal utilization of the SUPRENUM network hardware can be achieved by using a so called *blast protocol*. The principle of this protocol is to transmit all data packets belonging to the same message in sequence, with only a single acknowledgement for the entire packet sequence. Thus, network traffic with respect to transmission of acknowledgement packets is minimized. Moreover, the retransmission strategy applied either leads to the complete retransmission of the entire packet sequence or allows for selective retransmission of erroneous packets. What strategy actually would be the one with the best performance is to be investigated once both protocol variants have been realized for SUPRENUM. At first sight, it seems that the complete retransmission of an entire packet sequence will be the best solution. Due to the speed of the cluster bus and SUPRENUM bus, selective retransmission by a specific protocol functionality may significantly drop the overall communication performance.

For SUPRENUM, it is assumed that physical transmission errors are very rare. For example, the error rate for an intra-cluster message transfer over the cluster bus is comparable with the error rate of local memory-to-memory copies. As a consequence of this situation, the main functionality of the *blast protocol* should be the management of the physical network interface. The primary functionality should be to enable a direct end-to-end message transfer between peer user (team) address spaces. On this basis, low-level buffer and/or memory management within the network communication system is reduced.

13. Conclusion

Although PEACE is designed as an operating system for SUPRENUM, it founds a proper basis for various distributed/decentralized applications. The slogan is to design and implement a server process/team with a functionality as required by the specific application. Thus, the PEACE operating system represents one more competition for other distributed operating systems as, for example, listed in [Tanenbaum, van Renesse 1985] and [Balter et al. 1986].

Presently, the PEACE message-passing kernel is implemented and runs on a SUPRENUM prototype consisting of 9 nodes within a single cluster. Two different clusters are interconnected by ETHERNET and inter-cluster communication is tested, now. The name administrator functionality is completely available and most of the fundamental operating system server processes/teams are implemented.

The design aspects of PEACE, especially process structuring, light-weighted processes and synchronous message-passing, made it feasible that very early during the SUPRENUM project phase a common basis was available, onto which the execution of distributed SUPRENUM applications already was possible. In addition to that, without functional relieve of an operating system kernel, running first experiments with SUPRENUM would not be possible, now. There is no doubt that the design principle behind PEACE significantly promotes project-oriented system development.

14. Acknowledgements

I would like to thank Friedrich Schön and Winfried Seidel for their co-operation in working out the concepts of PEACE and for critical reading the manuscript of this paper. In addition, Jörg Nolte receives my acknowledgement for the fruitful discussion about the PEACE inter-node communication model and because of the rapid realization of the remote service invocation model.

15. References

[Balter et al. 1986]

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

[Behr et al. 1986]

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

[Birrell, Nelson 1984]

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

[Cheriton 1979]

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

[Cheriton 1984]

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

[Cheriton, Zwaenepoel 1983]

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

[Fabry 1973]

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

[Goodenough 1975]

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

[Habermann et al. 1976]

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

[Lam 1986]

S. S. Lam: **Protocol Conversion – – Correctness Problems**, Proceedings of the SIGCOMM '86 Symposium, Stowe, Vermont, August 5 - 7, 1986

[Lantz et al. 1985]

K. A. Lantz, W. I. Nowicki, M. M. Theimer: **An Empirical Study of Distributed Application Performance**, Technical Report STAN-CS-86-1117 (also available as CSL-85-287), Department of Computer Science, Stanford University, 1985

[Liskov, Zilles 1974]

B. H. Liskov, S. Zilles: **Programming with Abstract Data Types**, SIGPLAN Notices, 9, 4, 1974

[Metcalfe, Boggs 1976]

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

[Millen 1976]

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

[Mullender, Tanenbaum 1986]

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

[Nelson 1982]

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

[Parnas 1974]

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

[Parnas 1975]

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

[Popek et al. 1981]

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

[Randell et al. 1978]

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

[Schindler 1980]

S. Schindler: **Distributed Abstract Maschine**, Computer Communications, Vol. 3, No. 5, 1980

[Schroeder 1986]

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

[Schroeder 1986b]

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

[Tanenbaum, van Renesse 1985]

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

[Zwaenepoel 1985]

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