Communication- and Management Protocols for the Distributed PEACE Operating System*

L. Eichler, J. Nolte, T. Patzelt
F. Schön, W. Schröder, W. Seidel

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

ABSTRACT

This paper introduces the protocols for network wide inter-process communication and process management inside the PEACE operating system. A hierarchy of four operating system specific protocol layers is described. This hierarchy is constituted by the *naming protocol*, the *remote procedure call protocol*, the *dispatching protocol* and the *data transfer protocol*. The bottom layer of the PEACE communication system is defined by a *virtual network interface*, by which means a uniform network device access is given. The inter-relationship between the protocol layers is illustrated in terms of interactions taking place at certain interfaces. The functionality of each protocol is explained and the necessity of each of these protocols for each distributed operating system is manifested.

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

Table of Contents

Chapter 1

Introduction

The operating system for SUPRENUM, a high-performance distributed multi-computer system for numerical applications [Behr et al. 1986], is based on an application oriented process execution and communication environment, for which "PEACE" is the acronym[1]. The building blocks of the PEACE operating system, as well as for the application systems running on top of it, are processes [Schroeder 1986]. The interactions between the user- and system processes are based on the same basic mechanisms for inter-process communication [Schroeder 1987]. These mechanisms use synchronous message-passing and provide for a semantic of a *remote invocation send* [Liskov 1979].

1.1. The Principle of a Family of Communication Systems

The basic communication sub-system of PEACE, the message-passing kernel, is designed for optimal support of process structured operating- and application systems. The main design aspects of the message-passing facilities of this sub-system were given by the requirement of runtime efficiency and, therefore, were not based on such requirements as security, network transparency, concurrency and fault-tolerance. These latter mentioned requirements are integrated into PEACE as separate layers on top of the basic communication sub-system.

More comprehensive communication services are provided in an application dependent manner referring to the principles for the construction of program families [Parnas 1975]. On the lowest level, very elementary and high-performance mechanisms for inter-process communication are provided. This level is common to all communicating processes of PEACE. No design decision has been met on this level that would prejudice a process with respect to the runtime efficiency of inter-process communication.

More enhanced aspects of inter-process communication, such as security and reliability, are implemented using services of the low-level message-passing kernel. Basically there are two strategies in PEACE to provide for high-level communication services. One strategy is to use proper libraries, i.e. applying the principle of *remote procedure calls* [Nelson 1982]. The other strategy is to use dedicated system processes and automatically route remote inter-process communication to these processes. This

[1] Undoubtedly, PEACE with its conversational meaning by far is much more important than its technical meaning. Nowadays it, again, is of significant importance to remember PEACE instead to accept rearmament in silence.

strategy, although subject to less communication performance than the previously mentioned one, has the advantage of being able to dynamically integrate and/or exchange network software interfaces or protocols. In addition to that, using the route facility of the PEACE message-passing kernel, the usage of certain network interfaces may be enforced for processes not applying the remote procedure call library.

The communication system described in this paper is closely related to the demands given by the SUPRENUM architecture and given by the numerical application programs running on top of it. The primary design goal for this communication system is to optimally support numerical applications and to achieve a maximal utilization of the underlying network hardware. The consequence of these demands, and of the performance specification of SUPRENUM, is that the communication system interface should be easy to use and should not represent a bottleneck in the system design. Therefore the services of the basic PEACE communication system are very elementary in nature, i.e. there is little operating system overhead associated with these services. Thus, this communication system, more specifically the protocols developed for it, may be regarded as a basis for other communication protocol systems. In this sense, for instance, VMTP [Cheriton 1986] will be a candidate for the extension of the PEACE communication system to achieve a more general communication system interface.

1.2. Standards versus Non-Standards

The past decade has shown a lot of activities in the design of protocol standards and the implementation of communication systems. These works deeply influenced the design of operating systems. Taking advantage of the work done on communication systems has been a "natural" consequence, because one of the operating system's tasks is to manage the network devices and/or software interfaces. It has been recognized that the design of operating systems itself can take advantage of network-oriented systems and that it should not merely provide an interface or abstract machine to drive decentralized/distributed application systems. Presently many decentralized and/or distributed operating systems are available. See [Balter et al. 1986] for a general overview.

The work on communication systems led to the reference model for open systems inter-connection [ISO 1985]. There was a general agreement in the distributed systems community that the inter-connections of different distributed systems, or solely the integration of new systems into an existing complex of distributed systems, would be an awkward task if there was no common philosophy in communication systems design. Not only a framework for the design of "open systems" was necessary, but rather requirements for protocol standards, in order to define a common basis for information inter-change, were stated. As an example of such a protocol standard for the european area see [CCITT 1984].

Although [ISO 1985] may be regarded as a milestone in communication systems design, little or no emphasis was shown within the community of distributed operating system designers to apply the principles of the reference model [Mullender 1986]. The main reason for following this strategy was not to lose system performance. Indeed, too many services are defined by the reference model and there is little flexibility in using protocol subsets without losing the attribute of being an "open system". Application systems are often forced to use services they actually don't need. In case of an operating system this would result in a general drop of system performance. Principles as for example manifested in [Parnas 1975] would overcome this problem. Therefore, high-performance decentralized/distributed operating systems today, as for example the V-System [Cheriton 1984], use their own special purpose, i.e. *problem-oriented*, communication protocols. In the case of V the problem-oriented communication protocol is VMTP, which generally replaces [ISO 1985] level-4 transport protocols.

1.3. Towards a High-Performance Communication System

In order to construct high-performance communication systems basically two preconditions must be regarded. Firstly, high-quality and easy to use communications hardware must be available. And secondly, a specific design principle must be applied, to avoid overloading of lower-level communication software and hardware. Using the results of [Lantz et al. 1984] as the basis, the largest effort toward the design of a high-performance communication system should being the development of high-speed network interfaces as well as network protocols. Even if standard networking hardware, as for example *Ethernet* [Metcalfe, Boggs 1976], is used, the actual network efficiency is limited by the performance of the higher-level communication protocols. The same applies to the lower-level network i/o system. It is still difficult to achieve the maximal possible utilization of *Ethernet* based systems and it is even more difficult to do so for SUPRENUM.

The main aspect of the SUPRENUM communication hardware design takes into account that no design decisions for special protocol strategies should be based on hardware, so as to avoid overloading of low-level hardware components. Thus, the communication hardware is application-oriented in the sense that a broad class of higher-level communication protocols is supported. In addition to this, the communication hardware itself is more powerful, because its design is more basic. Development as well as maintenance of such hardware components is improved, also.

From the software design point of view, the same principles are applied. These principles require postponing significant design decisions, i.e. striving to represent design decisions by certain building blocks and inserting these building blocks on the highest possible level in the system hierarchy [Parnas 1975].

Because of the reasons mentioned in the previous sub-section, optimal support of a high-performance network system requires the design and implementation of problem-oriented communication protocols, rather than the use of standard protocols, to establish a connection between peer processes. With respect to remote procedure calls, following this pattern is motivated by [Birrell, Nelson 1984]. To give an example, a performance gain of a factor of ten might be possible when designing communication protocols specifically for remote procedure calls.

The raw data transfer in PEACE is managed by a special *blast protocol* [Zwaenepoel 1985]. This protocol benefits from the high-quality and high-performance communication hardware designed for SUPRENUM. More specifically this means removing time consuming activities from the protocol as far as possible. The consequence for a communication protocol is to avoid mechanisms for selective retransmissions as well as complex timeout management and to reduce temporary buffering of messages.

The design of the PEACE communication system has been influenced by the observations done in [Saltzer et al. 1984]. The complexity of communication protocols may be reduced if one keeps in mind, that improving reliability and security is a context sensitive fact. Without the knowledge what the semantics of a message exactly is, no absolute reliable message transfer is possible. Reliability and/or security actually must be regarded to be of *end-to-end significance* with respect to the information producing and consuming site, i.e. processes. In this sense it is of significant importance that distributed applications designers are aware that they themselves are faced with typical problems from the communications systems area. The consequence of these applications is obvious: a distributed system never is really transparent to the applications running on top of it if these applications demand a high-performance communication system.

From the software engineering point of view the most significant aspect in PEACE for the construction of a high-performance communication system is to provide for elementary services, which are used to compose more enhanced and application-oriented services. The idea is to let each application use only those services it needs to fulfill its task. Thus, to avoid loss of communication performance, no application process should be impaired by services it never needs. An application process in this context is regarded to be any process that uses (applies) the basic services from the next lower level. In this respect each process of a PEACE environment, especially the system processes of the operating system, is supported by the most basic and/or most application oriented set of services it needs.

1.4. Overview

In Chapter 2 the different protocol layers are explained on the whole. Their meaning is illustrated on a conceptual level.

In Chapter 3 each protocol layer is regarded in more detail. The protocol phases and/or activities are explained with respect to the *protocol data units* exchanged by the different protocol entities.

In Chapter 4 the interactions between the different protocol layers are illustrated. These interactions are explained giving sequences of *service data units* exchanged in conjunction with typical communication requests in PEACE.

In Chapter 5 some concluding remarks about the PEACE communication system are given. A short retrospective over the design decisions completes this paper.

Chapter 2
Protocol Hierarchy

In this chapter the hierarchy of communication and management protocols inside the PEACE operating system is discussed. The design decisions for this protocol hierarchy are illustrated and it is shown, on what level and in what way more enhanced protocol and/or operating system services may be introduced.

2.1. Separation of Concerns

The communication system of PEACE is faced with four main problems. These problems are typical for a process structured and distributed/decentralized operating system and thus are not given by the system structure and organization of the PEACE operating system.

2.1.1. Bridging Address Space Boundaries

On the top level, i.e. on the level on which ordinary processes will interact with the operating system (or with parts of it), usual procedure calls must be passed to other address spaces. In general the different address spaces each are constituted by a single team, which in turn may be controlled by a set of processes [Schroeder 1987b]. Thus actually remote procedure calls must be executed, whereby the attribute "remote" more specifically denotes that a procedure residing in a different address space is to be called. In distributed/decentralized systems the different address spaces are spread over the entire network, instead of being fixed on one machine. See [Nelson 1982] for a more precise discussion of this topic.

With process structured and on message-passing based operating systems it is generally of secondary importance to distinguish between distributed and non-distributed systems when designing a *remote procedure call protocol*. The problems with which such a protocol is faced in each case are basically the same. In the non-distributed case there also are situations in which duplicated calls may occur. This would happen, for instance, if the procedure calling site (client site) has been broken out of an existing inter-relationship to the procedure implementing site (server site), and repeats the previous call [2]. More specifically, the communication system may be forced to flush a message buffer

[2] Remote procedure calls often are implemented using a synchronous request-response protocol. This means that such a call is executed, for example, during a rendezvous between the client and server process. An exceptional abort of this rendezvous breaks the client out of the existing inter-relationship to its server.

pool, which too may result in the retransmission of the same message-encoded procedure call request. Another aspect is authentication, since messages are used to carry remote procedure call requests. The receiving server process must identify the requesting client process by the message-encoded request instead of using some magic "*currproc*" pointer, which is the usual way by which procedure based operating systems perform authentication checks.

Independent of the underlying system topology, the remote procedure call protocol in PEACE handles marshaling of arguments, authentication of the requesting client and removes duplicate requests on the server site as well as duplicate responses on the client site. Besides these functionalities there are additional management activities to handle the disappearance of a server site as well as the abortion of rendezvous, which are used in PEACE to model the inter-relationship between client and server during a remote procedure call.

There is one more important aspect typically associated with remote procedure calls. This aspect deals with the machine independent representation of procedure arguments and/or results in the sense of the ISO *representation layer*. We do not need to regard the full complexity of these representational aspects, because SUPRENUM is not a heterogeneous multi-computer system. With this respect, object representation only is of little complexity and is reduced on the functionality to read/write *typed messages*, i.e. the protocol data units of the PEACE *remote procedure call protocol*.

2.1.2. Identifying the Server Site

Another aspect of the PEACE communication system considers the problem of locating the process which actually is able to handle a remote procedure call, i.e. which is used to bridge the gap between the two address spaces involved with a remote procedure call. Because processes are the focus in PEACE to handle system service requests and because with processes generally a dynamic behavior is associated, identification and addressing mechanisms must not be of statical nature. A typical naming problem is given, independently of the fact of an underlying distributed and/or non-distributed system architecture.

In PEACE, remote procedure calls are used in order to invoke specific system services. Identifying the service providing process is done by the name of the requested service. The known service names, together with the internal address of the corresponding remote procedure, constitute the PEACE name space. This name space is observable by a client, thus, giving a service name, the internal address of the remote procedure is returned.

The service name space is hierarchically structured. The structure reflects the SUPRENUM system topology, in which nodes, clusters, hyper-cluster and the entire system are addressable units. The name space itself is managed by a set of dedicated system processes. More specifically, there is at least one such process for each of these addressable units.

Using dedicated system processes for the management of name spaces is a common technique in distributed systems. One of the latest examples is illustrated in [Cheriton, Mann 1986]. The decentralized naming system for PEACE is described in [Schroeder 1987c] and for a more precise analysis of various naming conventions for distributed systems see [Terry 1984].

A special *naming protocol* in PEACE is used to allow for the dynamical binding of system services. Thus, before a remote procedure call is stated the first time, its controlling process is located looking up the actual name space. This step reflects the hierarchical structure of the PEACE name space, in that the actual name space automatically is expanded by the next surrounding one (the hierarchically above placed name space) if the requested service name is not contained in it. Besides that, service names dynamically are included into and excluded from the name space. This name space manipulation results in specific naming exceptions, which usually are handled in some application-oriented and/or problem-oriented context. For instance, trying to re-bind the service function in case that a service name has been removed from the name space.

2.1.3. Dispatching Remote Processes

Before a remote procedure call, i.e. system service, is executed, its controlling process must be dispatched. In a non-distributed system this is still an easy undertaking. The mechanisms in PEACE for this task are the basic primitives for inter-process communication, provided by the message-passing kernel. Sending a service request blocks the requesting client and directly may dispatch the receiving server, if it still waits on a message-encoded service request. Returning the result goes the other way round. The server replies the client, with the effect that the client is set ready and may be dispatched for execution, again.

In a distributed/decentralized environment these dispatching strategies are more complicated to materialize. Actually a remote residing process, client or server, is subject for dispatching. To make remote service invocation feasible, a problem-oriented *dispatching protocol* is provided in PEACE. This protocol bases on a set of processes capable of receiving incoming service requests and executing the actual procedure within the processes' address space, i.e. team.

This scenario, too, looks still easy. However specific checks are necessary at the remote site in order to verify that the addressed process actually resides there and that this process will be able to receive the incoming service request. It is this level in the communication system hierarchy of PEACE on which message-buffering may be introduced in order to store incoming and not yet processable service requests. However, for performance reasons as well as to avoid starvation and deadlock problems inside an operating system, and inside a communication system, it is a general principle to avoid buffering in lower-level system components. A *good* design is based on the principle to keep buffer resources still on the application level and solely manage its usage on the operating system level.

It is a question of the *dispatching protocol* model if buffering is necessary at all. If, for example, for each service requesting client a representative (*deputy*) at the remote site is available, whose task it would be only to wait on incoming service requests, then no buffering is necessary. An incoming message immediately can be delivered to the waiting representative. Moreover, the service requesting process blocks and therefore is unable to transmit another service request to the same remote representative. The only disadvantage of this model seems to be the obvious large number of representatives, i.e. processes, at the server site. Address space switching time really would not be the actual problem within PEACE, because all these processes would reside in a single server team. Following this pattern, i.e. consequently using server teams, is motivated by [Mullender 1986]: "... *good* distributed systems work requires a large team".

An alternative model, which only bases on one client and server and which, too, does not need any message-buffering, would be to follow a principle already known, for instance, since ALOHA [Abramson 1973]. This model controls the interactions between peer processes in that sending a message to a process not being able to receive this message is regarded as a collision of service requests. We regard this as a service collision, because in this situation the server already processes a service request. Thus, the main task of a protocol designed for this model is to resolve such collisions in a way that avoids starvation of the service requesting processes.

The model for remote process/service dispatching favorized in PEACE is a compromise between the two above mentioned strategies. On the server site a pool of processes inside the server team will be available. Each time a remote service request is received, it is tried to dispatch a process from this pool and delegate the service execution accordingly. This scheme works much in a same way as AMOEBA [Mullender, Tanenbaum 1986] does. A collision of service requests only will take place if the process pool is empty at the time the service request is received. The optimal number of processes inside the pool typically is problem-oriented. Therefore, the remote procedure call level in PEACE is responsible for balancing the load of processes inside the process pool.

2.1.4. Passing variable sized Messages

The lowest level of the PEACE communication system is concerned with bulk data transfer. This level directly interfaces to the communication hardware designed for SUPRENUM. The performance of the SUPRENUM communication network is physically specified with a transfer rate of upto 256 Mbytes/sec. for the *cluster bus* and approximately 20 Mbytes/sec. for the *SUPRENUM bus* [Behr et al. 1986]. As a consequence, there is little time to play with complex communication protocol strategies.

Besides the performance requirements, a main design goal with the PEACE *data transfer protocol* is to provide for an interface "directly" accessible by higher-level application processes, without the need to invoke intermediary system services each time a message is ready for transmission. This protocol implements typical services of an *unreliable transport protocol*, only. However, it guarantees the most available data transfer throughput and gives a chance for the maximal utilization of the SUPRENUM communication network.

As already noted in a previous sub-section of this paper, the network interface is a critical point in the design of high-performance communication systems. We claim that the requirement of an easy to use and high-performance network interface is fulfilled if basic services are available in order to construct more enhanced communication services. Thus, a consequent separation of concerns is the mandatory design principle by which overloading of lower-level protocol entities is avoided.

This principle is of significant importance for SUPRENUM, because it enables the system designer to postpone certain design decisions dealing with the question what functionality to put into hardware and what functionality should be represented by software modules. In addition to that, it allows for to migrate, step by step, software implemented design decisions into hardware. Precisely this strategy is followed with the design of the network communication system in PEACE. Thus, the *data transfer protocol* should be basic enough to be, as the final step, completely representable by hardware components, more specifically by a microprogrammable unit. Keeping the SUPRENUM performance specifications in mind, see [Behr et al. 1986], this is the only way of driving the communication hardware with maximal utilization.

2.2. Functional Hierarchy

From the software engineering point of view, the aspects of a PEACE communication system, so far discussed, are worth to be represented by dedicated and autonomous functional units. The inter-relationship between the different functional units can be best explained giving the <u>uses relation</u> [Parnas 1974] of the basic PEACE communication system. Figure 2.1 shows the resulting structure.

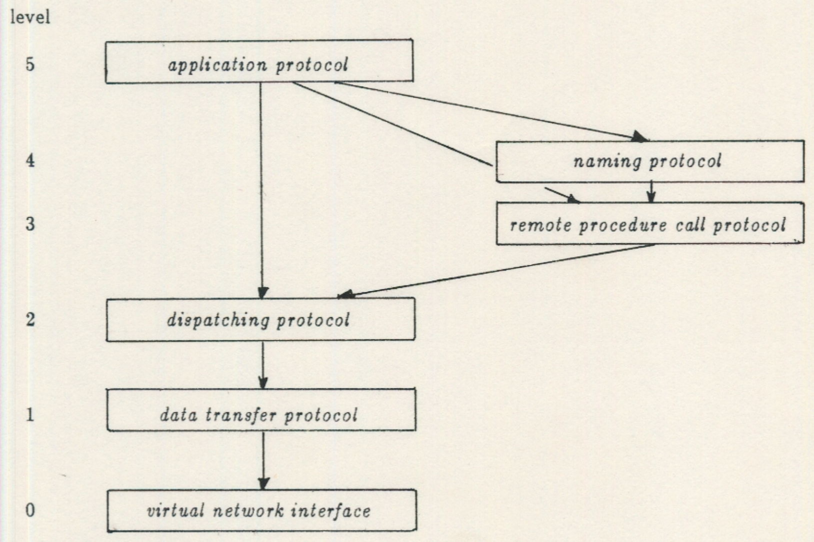

Figure 2.1: Functional Hierarchy of the Basic Communication System

It should be noted that the level numbering in this figure is not comparable with the level numbering of ISO communication protocols. It simply denotes a functional level of the basic PEACE communication system. In the following we explain the uses relation between the various functional units in a top-down fashion, i.e. starting with level 5.

2.2.1. The Uses Relation

Basically the application processes on level 5 either will state system service requests, such as the creation of a new process, or they request the communication with other application processes, maybe passing node boundaries in both cases. The system service requests are represented as remote procedure calls directed to dedicated (system/user) server processes[3]. To locate the corresponding server process for a remote procedure call, the naming services from the level 4 functional unit are used. Actually, remote procedure calls are handled by level 3. The functional unit of this level assumes that the location of the procedure, i.e. service, is already known. This location either is statically known on the application level, hence the direct inter-relationship between level 5 and level 3, or it is dynamically determined by the level 4 naming services.

The protocol specific part of level 3 upto level 5 is represented in PEACE on a library basis. Dedicated system processes only are used if certain system services, such as the initial lookup of name spaces to locate a certain server process, are requested. This means that the protocol activities are directly controlled by any PEACE process and that no address space boundaries must be bridged while running a level 3 upto level 5 protocol at the client and/or server site.

The next two lower levels, level 2 and level 1, represent the PEACE communication system kernel. Mechanisms for dispatching of remote accessible services/processes are provided on level 2. In this sense level 2 represents the problem-oriented part of checking the process pool on the server site and checking for end-to-end data transfers directly initiated by level 5 application processes. On level 1 the raw data transfer is controlled and the entire PEACE communication system is adapted at the underlying network hardware interface. This interface, i.e. level 0, hides the network hardware dependent facts from the higher-level communication/management protocols. In this sense, a virtual interface is defined, giving conform access to a various number of physical network interfaces.

2.2.2. Representational Aspects

Consequent structuring a system necessarily does not mean to design overhead-prone communication paths, which inter-connect the different functional units. In the case of the PEACE communication system presented here, not all functional units are used each

[3] The high-volume data transfer request may be represented by a remote procedure call to the respective communication partner. Thus, each process in PEACE implicetly provides for a message transfer service.

time an application process initiates some service request[4]. However, using the higher-level functional units of level 3 improves the reliability for the transmission of a service request. Variations in the usage of level 4 and/or level 3 services are possible on a peer-to-peer basis. However, both processes involved in the communication must use the same protocol functionalities to understand each other.

The most reliable communication path is given if the *remote procedure call protocol* services are applied. The variation with the most expected throughput, however the least reliable one, is given if the *dispatching protocol* services are directly applied. What service interface to use is the decision of the single application processes and is controlled on the level of runtime libraries, i.e. on level 5.

The standard scenario we prefer is the following. Each call, directed to another process, results in using level 3 services. The process which accepts the call has previously been made known using level 4 services. These services, however, are used in two situations, only:

a) to locate the process to which the call is directed and

b) to handle a signaled naming exception.

Both situations is in common that a search through the hierarchical structured PEACE name space is requested from a dedicated system process (*name server*). However, this search is initiated only once. If the remote procedure call implementing process has been located then with its internal address (process identification) is worked further on. The consequence is that, in the normal/usual case, the application level directly uses the PEACE *remote procedure call protocol* services (level 3).

The general overhead of the PEACE communication system is dictated by level 0 upto level 3 functional units. The protocols on these levels, except level 0, are designed in such a way that overhead will be produced in exceptional situations, only. This is the same principle as discussed above, i.e., in the normal case, minimal protocol overhead is given, considering the transmission of a message-encoded remote procedure call request to some server process.

Normal protocol processing is concerned with manifesting some protocol states, interpreting and sending/receiving a *protocol data unit*. Timeout controlled supervisory functionalities, if any, are considered of being exceptional conditions. The way these conditions are handled in PEACE is a straightforward task if light-weighted processes are applied. These processes reside inside the application team on level 3 and/or level 4, invisible from the application processes on level 5, and are conditioned to receive *probes* from a problem-oriented protocol supervisor, a special system process. Sending *probes* is a common technique to ask for attention at the receiving site, see [Birrell, Nelson 1984] and [Cheriton 1986]. In case of the PEACE *remote procedure call protocol*, for instance, *probes* are used to trigger timeout management. With this strategy retransmission

[4] More specifically, it should be noted that a uses relation does not mean actually calling some lower-level procedure [Parnas 1974]. This may only be one valid technical representation, as well as it would be in case of a macro invocation or inline coding.

overhead due to a timeout at the client site is not produced along with the normal transmission of messages. More specifically, a common basis is given with which protocol exceptions may be handled, i.e. protocol exceptions signaled from lower levels and propagated properly to higher levels.

2.3. Inter-Networking

Four different network systems are subject for consideration with the PEACE communication system design. These network systems are represented by the *cluster bus* and *SUPRENUM bus* [Behr et al. 1986], the *pipeline bus* [Franke 1986] and *Ethernet* [Metcalfe, Boggs 1976].

The *cluster bus* always is the inter-connection system inside a SUPRENUM cluster. The inter-connection of different clusters, however, is represented by any of the other three network interfaces. Which of these interfaces actually is used depends on the development stage of SUPRENUM.

Because of the situation that always two different network interfaces are necessary to construct SUPRENUM, the design and implementation of the proper communication system is faced with typical inter-networking problems. These problems, for instance, are present with protocol conversion aspects [Lam 1986], timeout controlled protocol functionalities, due to the different transmission speeds of the network systems, as well as with time consuming *store-and-forward* techniques, to balance the message throughput on the different network systems. Protocol conversion is avoided in PEACE, because the same *data transfer protocol* runs on all SUPRENUM inter-connection systems. However the other problems, especially the *store-and-forward* problem, still exist.

To make the adaptability at different network interfaces for the PEACE communication system feasible, the level 0 functional unit is introduced. This unit less is concerned with communication protocols, but rather with the abstraction from specific network hardware interfaces. It actually encloses the network device driver modules. The idea is that the various SUPRENUM network systems merely are visible as a specific low-level i/o system. In this way, level 0 represents a *virtual network interface* which makes the higher-level communication system independent from any special network hardware.

Chapter 3

Communication and Management Protocols

The previous chapter gave a brief overview of the structure of the basic PEACE network communication system. A motivation for this layering was given as well as an explanation of the principle functionality associated with each level. This chapter, now, is concerned with the communication and management protocols used to implement the various functional units of the PEACE communication system. In a top-down fashion we will figure out the different protocol sequences.

3.1. The Notation of Protocol Sequences

We describe the protocol sequences in terms of the protocol data units exchanged by the peer protocol entities. The description itself is based on a compact and formal notation, basically a subset of RSPL-Z [Hahn, Herrtwich 1983]. The constructs of this subset and the semantic of each construct, as applied to describe the PEACE protocol sequences, are shown in table 3.1.

Construct	**Meaning**
↑	accept (receive) the following protocol function (data unit)
↓	deliver (send) the following protocol function (data unit)
→	enforce sequencing (*state transition*)
*	skip or repeat at least one times
+	repeat at least one times
( )	enclose alternatives
[ ]	qualify alternatives

Table 3.1: Constructs for Protocol Sequence Specification

The protocol sequences are represented by a specific grouping of protocol functions. These functions actually denote some protocol data units, which are exchanged between peer protocol entities to model a certain protocol behaviour. The grouping of protocol functions either is achieved by function sequencing, i.e. expressing a protocol state transition, or by giving alternatives in function processing.

Using this subset, $\uparrow a$ denotes acceptance of protocol data unit a. On the other hand, $\downarrow a$ denotes delivery of protocol data unit a. With $a \rightarrow b$ we express a protocol sequence in which processing of b is only initiated if a already has been processed. Simple alternatives are expressed writing $a\,b$, which denotes processing of either a or b. Complex alternatives are composed using brackets. Thus $(\uparrow a \rightarrow \downarrow b)(\uparrow b \rightarrow \downarrow a)$ requires either the acceptance of a and then the delivery of b or it requires the acceptance of b and then the delivery of a. Alternatives, simple as well as complex ones, may be qualified by appending *qualifier parameters* embraced by "[" and "]". These parameters usually represent specific members of a protocol data unit and they are bound to each function belonging to the qualified alternative. Repetition is expressed by appending either "*" or "+" at an alternative expression. For instance $(\uparrow a \rightarrow \downarrow b)*$ means skipping or repeating the entire alternative, i.e. the embraced sequence expression.

3.2. Common Functionalities

The *remote procedure call protocol*, the *dispatching protocol* and the *data transfer protocol* are typical communication protocols in the sense that protocol data units are exchanged by peer protocol entities using some lower level communication service. The *naming protocol* is somewhat different, in that it merely regulates the way services are located within PEACE. Actually no protocol data units are exchanged using this protocol.

3.2.1. Exchange of Protocol Data Units

The exchange of protocol data units is accomplished in two ways: confirmed and non-confirmed. The *confirmed exchange* improves the reliability and security of the respective protocol, however at the expense of general communication performance. The *non-confirmed* exchange, on the other hand, promotes general communication performance, however at the expense of reliability and security. What principle to prefer depends on the requirements associated with the communication system.

The basic communication protocols, except the *naming protocol*, implement confirmed as well as non-confirmed exchange of protocol data units. Providing both strategies is a demand we impose on each PEACE protocol implement. Without this demand, control and observation of certain protocol activities is not feasible. More specifically, adding reliability mechanisms for communication protocols later is difficult and may slow down communication performance [Mullender 1986].

The confirmed exchange of protocol data units is a pure management activity of the corresponding protocol entity. With except of timeout handling at the sender site, no specific protocol state transitions will occur. The confirmation actually is represented by the header information taken from the received protocol data unit.

3.2.2. Dynamical Verification

An important management protocol data unit associated with each PEACE communication protocol is the *probe*. The *probe* is used to poll peer protocol entities for the return of certain protocol state information. Using *probes* enables the dynamical verification of PEACE communication protocols. They are applied to detect the availability of remote nodes or they are specifically used to observe the functioning of certain protocol levels and/or entities.

The coding of *probe* protocol data units, as well as the returned *state* protocol data unit, is protocol dependent. However the protocol sequence for sending/receiving *probes* and *states* are the same for each PEACE protocol. This sequence is as follows:

$$((\downarrow probe \rightarrow \uparrow state)(\uparrow probe \rightarrow \downarrow state))*$$

Once *probe* has been delivered, processing the first (leftmost) alternative, then *state* is accepted as one of the next received protocol data units. On the other hand, if *probe* has been accepted, processing the second (rightmost) alternative, then *state* is delivered. The entire sequence may be repeated arbitrarily.

As with *probe*, the function *state* is strictly protocol dependent and, too, is represented by a protocol data unit on its own. This protocol data unit is used to read and to observe the state of a peer protocol entity. Typical informations associated with *state* is the actual execution state of the protocol state machine. The execution state reflects the actual *protocol phase* and indicates the next protocol data unit to be delivered and/or which protocol data unit is expected for acceptance. The basic protocols described in this paper will response at least with the protocol execution state each time *probe* has been accepted.

3.2.3. Exception Handling

The protocol sequences given in the following sub-sections each describe the expected behaviour of a certain PEACE communication protocol. All protocol functions not listed with these sequences are considered of being exceptional conditions. Usually, no specific protocol activities are performed in this situation, i.e. the protocol state actually remains unchanged. Solely, specific management activities may be initiated within the protocol entity where the protocol exception occurs. The consequence of such activities might be the propagation of protocol exceptions to higher-level protocol levels.

A specific situation arises if *probe* is received. In this case, too, the exceptional condition is present, but instead of ignoring or propagating this condition, the protocol function *state* is executed and the corresponding protocol data unit is returned to the *probe* initiating protocol entity. Thus, this exceptional condition completely is handled by the actual protocol entity, independently of the actual protocol state.

3.3. Naming Protocol

As noted in a previous sub-section of this paper, the *naming protocol* is somewhat different from the other PEACE protocols. Instead of controlling the communication activities, the *naming protocol* merely regulates the way a service is located in the PEACE operating system and what route to take in order to invoke the service by some client process. In the following sub-sections the *naming protocol* is described as it appears within the PEACE protocol hierarchy. A more detailed description of the entire naming system will be given in [Schroeder 1987c].

3.3.1. Hierarchically Structured Name Space

According to the SUPRENUM architecture, the *naming protocol* bases on a hierarchically structured name space. This name space is constituted by five different name scopes. Figure 3.1 shows the corresponding naming hierarchy.

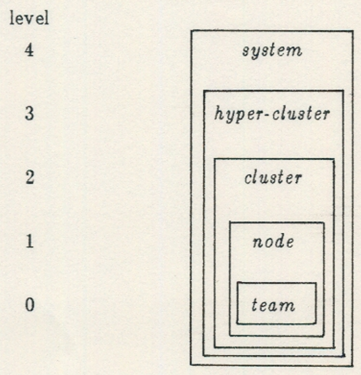

Figure 3.1: Naming Hierarchy

In this hierarchy, the *team name space* actually is implemented by a compiler/linker of the used software development system. On this level, the procedure call binding already has been done statically at compile/linkage time. The other, outermost, name scopes are implemented by dedicated system processes in PEACE, so called *name server*. These system processes serve for a dynamical procedure call binding, actually supporting the implementation of remote procedure calls.

3.3.2. Modeling a Name Space

In order to enable network transparent identification of service providing processes, service names must be properly inserted into the PEACE name space. The scope level associated with a service name defines the visibility of the corresponding service and, more specifically, of the actual server process. As an example, PEACE services may be bound local to a team, only, or they may be bound local to a cluster, thus being global to and therefore visible from upto 20 SUPRENUM nodes.

Modeling the PEACE name space is of strictly dynamical nature. Names may be inserted into and removed from the name space. This facility supports the implementation of dynamical configuratable operating and/or application systems so as of being able to create and destroy server processes dynamically, without the loss of service linkage. Table 3.2 shows the basic PEACE functions for modeling the name space.

function	**Meaning**
assign	insert a service name into the name space and associate a certain name scope with it. Additionally, signal the <u>name</u> <u>assign</u> <u>exception</u>.
remove	remove a service name from the name space and, additionally, signal the <u>name</u> <u>remove</u> <u>exception</u>.
replug	change the mapping from logical service name to server process and/or internal representation and signal the <u>name</u> <u>replug</u> <u>exception</u>.

Table 3.2: Functions for Name Space Modeling

With each function a *scope* argument denotes the name server responsible for managing a certain name space. This name server will receive the corresponding naming function and, thus, is able to model the name space it controls. The identification of this server process itself is taken from the name space. In addition to that, a *name* argument represents the logical service (and/or process) name in terms of an arbitrarily character string. With *assign* and *replug*, an internal representation, *repr*, of this service name is associated. This representation, for instance, is used by the *remote procedure call protocol* to construct a *service identifier* 5). Additionally, the function *replug* is supplied with the process identification, *pid*, of that server process which is responsible for executing the service, designated with *name*.

Executing the listed naming functions results is signaling a PEACE system exception. In case of *replug*, a <u>name</u> <u>replug</u> <u>exception</u> is raised and in case of *remove*, a <u>name</u> <u>remove</u> <u>exception</u> is raised. Both exceptions are propagated to all client processes actually being connected to the service, designated with *name*. In case of *assign*, a <u>name</u> <u>assign</u> <u>exception</u> is raised. This exception is used by the decentralized naming facility of PEACE in order to support the management of replicated name space informations. More specifically, dedicated system processes are notified about a new service name and, thus, being able of checking the integrity and uniqueness of this service name, with respect to a specific application and/or system context.

5) The meaning and structure of a *service identifier* is explained in the next sub-section, which describes the *remote procedure call protocol*.

3.3.3. Observing a Name Space

Basically, each PEACE name server implements a flat name space. The hierarchically structured name space then is constructed using the flat name spaces and arranging a tree-structured inter-relationship between them. Based on this hierarchically structured naming tree, different naming scopes are qualified much in the same way as within block-structured programming languages. Identification of a certain flat name space is achieved by identifying the corresponding controlling name server. More specifically, controlling a certain flat name space is a service a name server provides and thus a service name is used in PEACE for representation of name scopes.

Traveling through the PEACE name space is initiated by firstly locating the name server responsible for service name resolution. The second step is to request from this initial name server a connection to the service implementing server process. The final step is to request a disconnect from the service implementing process, which usually takes place if the service requesting process terminates. Table 3.3 shows the basic functions for a search through the PEACE name space.

function	**Meaning**
locate	return the process identification (scope identifier) of the server for the requested service name, without establishing a service connection.
attach	request a service connection and return the service identifier for later service requests.
detach	relinquish a service connection.

Table 3.3: Functions for Name Space Searching

In order to reduce name resolution time, the function *locate* is used. This function generally returns the *scope identifier* for the requested service *name*. Moreover, it is this scope identifier which designates the name server used for further name resolution. Although general applicable for process localization, *locate* primarily is used by the *naming protocol* to address a specific name server and, thus, only returns a hint where to start with name resolution.

The functions *attach* and *detach* manage a logical connection between client and server processes based on service names. This connection enables the name server to raise name replug/remove exceptions to service using processes.

Using these basic naming functions in order to locate a service providing process, a multi-stage strategy is implemented with the *naming protocol*. The design decision we took is to remove any hierarchy-specific management activities from the basic name server. Thus, each PEACE name server itself is unaware of being used as a building block for the construction of a hierarchically structured name space. With this design decision a more flexible principle must be applied on behalf of the service requesting client process to enable traveling through different name scopes, instead of asking only one name server. The principle is based on a namer list, maintained at the client site. Establishing a service connection is requested, step by step, from the server processes contained in the namer list. These processes are assumed of providing the basic naming functions. The sequential search is finished once the actual selected server process returns a positive response, i.e. indicates a name match. In this case, the service connection has been established and the corresponding service providing process is addressable. If no server process responses with a result indicating a name match then the requested service presently is unknown. In this situation, the requesting client process either may decide to wait on the service or to continue execution, with an indication that the service name is unknown.

The server processes contained in the namer list basically are separated into two classes. One class embraces the basic name server processes, each one controlling a flat name space. The other class is constituted by *scope server* processes, whose task it is to control the name space search on demand of the service requesting client processes. Actually, the scope server requests for some client a service connection from some name server. Thus, on the one hand, a simple naming strategy is supported on a library basis when processing the namer list and, on the other hand, more enhanced naming strategies are supported if scope server processes are contained in the namer list. To what class such a server process belongs is invisible by the client processes. These server processes are located by the initial name server on behalf of a specific service name and, therefore, are logically addressed. As a consequence, the namer list merely contains the process identification of the corresponding name/scope server.

3.3.4. Initializing a Name Space

The main problem in observing the name space is to locate the initial name server by a service it provides. Once this name server is known, locating other name server processes, as well as ordinary system/user processes, is straightforward. Another aspect is making name server globally known in the sense of bridging node boundaries when the name space is observed for a certain service name. A generalized view is to make services globally known, at all, such that the construction of a decentralized/distributed operating system is possible. All these aspects are handled in the same way using *replug*.

A *scope manager* is used in PEACE to associate service names with specific name scopes, i.e. with specific name server processes. This scope manager applies *replug* with an argument list in such a way, that a specific name server is requested to remember a service name with an explicitly named process. The specific name server is addressed by the scope identifier and the named server process is addressed by its process identification. Both identifiers are of the same type and, more specifically, represent the system-wide unique address of the corresponding process. Applying this principle, an arbitrarily name server within the PEACE environment is requested to remember a service name, its internal representation and its controlling server process.

3.4. Remote Procedure Call Protocol

Primarily the complexity of the PEACE *remote procedure call protocol* is specified by the model with which duplicate request/response messages, authentication and timeouts are handled. We will discuss each of these facilities in the following sub-sections and then describing the protocol data units as well as the protocol sequences.

3.4.1. Duplicate Suppression

A *remote procedure call protocol* is faced with the problem of duplicate requests received on the server site and duplicate responses received on the client site. The reason for duplicates may be manifold. Administrative steps local to one site, for instance the termination and re-creation of a client and/or server process, may cause this situations as well as node failures, which are handled by proper recovery actions. With either of these problems PEACE is confronted.

Regarding a synchronous request-response communication between a client and server process, three situations may cause duplicate messages. These situations are:

a) the server receives the same request several times and has not yet answered with a response;

b) the client receives the same response several times;

c) the server receives an already responded request at least one times.

Not considered are situations provoked by *intruder* server processes, i.e. that the same response is returned by different server processes. These situations are handled by the *dispatching protocol*, as explained in one of the following sub-sections. Basically, the rendezvous used to model the synchronous request-response inter-relationship during a remote procedure call can be terminated only by that server process to which the request has been directed.

The cases a) and b) are handled in PEACE following the pattern of [Birrell, Nelson 1984]. A *call identifier* is transmitted with each request and response. Each new stated request at the client site, by the *user stub*, is supplied with a different call identifier. This identifier is used at the server site, by the *server stub*, to detect replayed requests (calls) and it is used at the client site, by the *user stub*, to detect replayed responses (results). On both sites the most recently created call identifier must be remembered to perform the necessary checks. About the typical functionality of "user stubs" and/or "server stubs" see [Nelson 1982] and [Birrell, Nelson 1984], for instance.

In order to handle case c), the server site is forced to perform additional functionalities. The last responded request must be remembered for being able to detect a request which already has been processed. This is accomplished by storing the last transmitted response and thus storing the corresponding call identifier of the request, too. It is necessary to store the entire response message because the server site is not aware of the previous response being accepted by the client and that the replayed request, for instance, is the reaction of some user-initiated timeout associated with the service request.

Obviously a call identifier always must be related to a client process. This is done in PEACE by structuring this identifier as a tupel {*process identification, call number*}. The *process identification* member uniquely represents in PEACE a system-wide identification of a process. The *call number* member is a monotonic sequence number. This number is used to generate different call identifiers for the same process. See [Birrell, Nelson 1984] for more detail.

3.4.2. Authentication

Each request packet of the PEACE *remote procedure call protocol* contains the identification of the process which acts as the service requestor. This strategy has the advantage that a third process is able to request services for a specific process. As noted in [Randell et al. 1978], recovery of process states is made possible if an audit trailed service history is read and the services are requested again, with the proper process identification associated. The problem given with this technique, however, is to prevent a process from unauthorized service requests by *intruder* processes. The server site must be in the position to detect unauthorized service requests such that no process illegally is able to request services for any other process.

In order to implement authentication we use the notion of a *service access point* or, more specifically, *channel*. Actually, a channel is used to transmit requests and/or responses and may be classified as *privileged* and *non-privileged*. A channel identification is returned to the client process if it initially imports the services provided by a server process[8]. This identification is generated by the server process using the basic inter-process communication primitives of the PEACE message-passing kernel. Each primitive of the message-passing kernel returns a channel identification (i.e. process identification). In order to export services, the server must receive the import request from the client site using some basic inter-process communication primitive. The channel identification returned to the server, more specifically returned to the server stub, is responded to the requesting client.

For authentication of a client process, the server process compares the process identification contained in the request message with the channel identification returned from the message-passing kernel upon message reception. If they do match, the request is assumed of being valid. If they do not match then the channel identification must denote a privileged channel to enable further processing of the received request. If the channel is not privileged, the exceptional processing of the received request is initiated.

Distinguishing a privileged channel from a non-privileged one is achieved by a basic mask operation on a channel identification object. It is not possible for a non-privileged *intruder* process to force the PEACE message-passing kernel to return privileged channel identifications. Likewise, because channel identifications actually are process identifications, it is not possible for a non-privileged *intruder* process to create a

[8] In [Birrell, Nelson 1984] a similar principle is described.

privileged process whose process identification may be used as a channel identification. Therefore the channel identification (process identification) returned as the result of the basic massage-passing primitive always is correct.

3.4.3. Timeout Management

Timeouts are only applied with the *remote procedure call protocol* if confirmed exchanges of protocol data units are requested. There will never be a timeout associated with a certain service request, for instance the creation of a process, because the remote procedure call level does not know the semantic of these requests. Thus, setting timeouts is of strictly problem-oriented nature and the timeout controlled problem to be solved within the *remote procedure call protocol* is to await confirmations for transmitted protocol data units.

There is another important aspect which additionally makes timeout management strictly dependent on the underlying network topology. As already mentioned, the PEACE communication system is faced with inter-networking. Two different inter-connection systems, offering different transmission speeds, are used to construct SUPRENUM. This makes timeout management difficult if end-to-end confirmations are considered. The timeout interval depends on the route the request/response message takes to reach the destination node.

Because confirmed exchanges of request/response message are optional, timeout management is optional, too. Thus, using non-confirmed messages, any timeout management overhead is avoided. Additionally, in this situation the inter-networking problems associated with timeout handling are not present at all.

Timeout management at the *remote procedure call protocol* level is controlled using a light-weighted process residing in the client and/or server team. This model consequently separates the normal flow of control, i.e. transmitting request/response messages, from the exceptional flow of control, i.e. re-transmitting request/response messages. More specifically, because a dedicated process is used for encapsulation of timeout management, the dynamical as well as decentralized ("from remote") control of timeout intervals is supported.

3.4.4. Tracing

Because all PEACE system activities are initiated using the paradigm of remote procedure calls, the *remote procedure call protocol* represents the central system component where to insert general tracing functionalities. The strategies, however, what events should be traced, is problem-oriented. With this respect, the trace facility of the *remote procedure call protocol* merely enables monitoring of the protocol behaviour on behalf of the server team, more specifically by a specific library package linked to the program controlled by the server team.

Depending on what trace strategy should be followed, the proper library package is linked during program generation time. Actually, if tracing is enabled, the library package is called each time a protocol data unit has been received by the *remote procedure call protocol* level, i.e. by the server stub.

The most important aspect of the tracing facility in PEACE is to enable audit trailing and timeout controlled execution of service requests. With audit trailing one specific mechanism for the construction of fault-tolerant systems is implemented. The time controlled execution of service requests supports several performance analysis and/or supervisory concepts. Both principles is in common, that tracing is a server-defined and service-related activity. As a consequence, the trace package usually will know the semantic of the specific remote procedure call.

The alternative to the tracing facility controlled by the server stub is applying the PEACE monitoring concept [Schroeder 1986]. Instead of adding a special purpose trace package to the server team, a monitor team is placed in front of the server team's interface. As with the server stub alternative, the monitor team is server-defined and service-related. Each time it receives a remote procedure call request, the proper trace functionalities are performed. Finally, the monitored request is relayed to its original server process.

The advantage of the monitor concept is apparent. With respect to the server team implementation, there are no special purpose interventions necessary in order to achieve service-related tracing. The drawback of this concept, however, is additional process switching time and the inability of directly accessing service-related data structures maintained by the server team. Because process switching time in PEACE is very low[7], this aspect is negligible when considering the time it needs to execute a service (e.g., reading from a file and/or creating a process image). The fact, that service-related data structures are not directly shared with the monitor team, however, may reduce the ability of performing audit trailing and/or dynamical verification, for instance. Independently of any *pros and cons*, both tracing models having its place within PEACE in order to implement specific supervisory functionalities.

3.4.5. Protocol Data Units

Basically two classes of protocol data units are supported, as it is the normal situation with each communication protocol. One class only serves for specific protocol management activities while the other class actually is used to transmit user data. Common to both classes is the same protocol header, which is constituted by the tripel {*type, service identifier, call identifier*}. The *type* member identifies the actual protocol data unit and the *call identifier* identifies the service requesting client process and is used to implement duplicate suppression. The *service identifier* represents the internal coding of the remote procedure call. This identifier actually is represented by the tupel {*process*

[7] Actually, the rendezvous is slowed down by approximately one third of its general execution speed. Instead of performing the remote procedure call within 400 μsec, approximately 530 μsec are used.

identification, *service number*}, whereby the *process identification* member uniquely represents the service providing server process and the *service number* indicates the service procedure which is to be called at the server team.

The size of the remote procedure call protocol data unit is limited by the facilities provided by the basic message-passing kernel of PEACE. These facilities, for example, are the ability to transfer fixed-size messages, only. The communication services of the message-passing kernel are designed to transfer a complete protocol data unit without the need of segmentation [8)]. More specifically, the request/response messages, at least those ones dedicated to system processes, are handled as *expedited data* by the lower-level communication protocols. In table 3.4 the protocol data units of the PEACE *remote procedure call protocol* are listed.

Type	**Meaning**
request	state a remote procedure call request in order to invoke either a corresponding service the server provides or an internal service of the peer *remote procedure call protocol* entity.
response	return the result of a remote procedure call request.
probe	poll a peer *remote procedure call protocol* entity to return its actual protocol execution state.
state	return the actual protocol execution state as response to *probe*.
trace	control tracing of remote procedure calls accepted at the server site.

Table 3.4: Remote Procedure Call Protocol Data Units

Usually the *request* and *response* types carry user data, i.e. they are used to implement service data units. In addition to this, these two functions are used to transfer management service requests to peer protocol entities. Management services request and system service request, i.e. those services implemented by the server process, both are coded in the *service number* member of the *service identifier*.

8) Presently fixed-size messages are limited to 64 bytes and the size of the protocol header is limited to 16 bytes. Thus, actually 48 bytes user data information, the marshaled remote procedure call arguments, are available.

3.4.6. Protocol Sequences

Writing the basic protocol sequence to carry a remote procedure call request from the client site to the server site is straightforward. The same is true for carrying results from the server site back to the client site. The protocol sequence from the client's point of view is as follows:

$$(\downarrow \mathit{request} \longrightarrow \uparrow \mathit{response})[\mathit{ci}]$$

The entire sequence is bound to the same *call identifier*, *ci*, and thus designates the flow of control associated with one specific remote procedure call. The client actually initiates this sequence by transmitting *request* with a new generated and unique *ci*. The next step is awaiting the corresponding *response*. Because a response message may be received twice, although sent only once, duplicate suppression must be applied. This is expressed by the qualify parameter *ci*. All response messages, excepted the first one, qualified with a *ci* which is different from the *ci* associated with the actual request, must be discarded by the client site.

The protocol sequence for the server site is somewhat different. The following expression represents the corresponding protocol activities:

$$(\uparrow \mathit{request} \longrightarrow \uparrow \mathit{request}* \longrightarrow \downarrow \mathit{response} \longrightarrow (\uparrow \mathit{request} \longrightarrow \downarrow \mathit{response})*)[\mathit{ci}]$$

This sequence too is qualified with the same *call identifier*, *ci*, as it was the case with the corresponding sequence for the client site. The specific aspect of this sequence is the multiple occurrence of *request*. The first *request* actually corresponds to the *request* delivered by the client site, i.e. it designates the actual remote procedure call request. The second *request* denotes a duplicate request message received before the response message has been transmitted. This alternative may be skipped, i.e. the corresponding *response* is transmitted, or arbitrarily repeated, i.e. the duplicates actually are discarded. The third *request* represents request messages received although the response message already had been sent. In this situation the response message must be re-transmitted, which is expressed by the corresponding *response* function. This last sub-expression may be skipped, which indicates that *response* was accepted by the client site, or it may be repeated in case that one more *request* with the same *ci* came in. The additional delivery of *response* packets eventually produces duplicate response message received at the client site. These duplicates, however, are discarded, as it is observable from the client's protocol sequence expression.

3.5. Dispatching Protocol

In this sub-section, the level 2 protocol will be introduced. The functionalities of this protocol layer are characterized by synchronization of remote processes and communication between address spaces located on different nodes. A dedicated *dispatching protocol* is applied for implementation of these functionalities.

3.5.1. Inter-Process Cooperation

The main characteristic of the *dispatching protocol* is the way it enables synchronization and cooperation between processes residing on different nodes. There are a couple of primitives provided by the PEACE message-passing kernel in order to achieve inter-process cooperation. For a general discussion of these primitives, see [Schroeder 1987].

Some of the basic message-passing kernel primitives are of local nature, only, because the execution state of the actual running process is manipulated. To give an example, *receive* will block the actual running process, without readying some other process – although some other process will be dispatched for execution. In contrast to this primitive, the application of *send*, *reply*, *relay*, *accept* and *signal* may set some other process ready, thus manipulating the execution state of the corresponding process. If this process resides on the same node, readying is a straightforward task. However, if the process resides on a different node, special activities are required in order to perform a remote dispatching functionality. These activities are discovered with the *dispatching protocol* of the PEACE communication system.

3.5.2. Service Collision Detection

Besides the dispatching functionality, most of the PEACE message-passing primitives actually enable the transfer of message objects. The consequence is, that the actual message transfer potentially is of remote nature. The main problem in this situation is the fact, that the remote server process may be unable to immediately receive an incoming message. In the local case, the sending client process blocks on the server process, i.e. the client is queued into the server's receive list.

The remote case, however, requires a somewhat different strategy, because the client process actually does not reside on the server processes' node. A solution would be to queue message objects, instead of process objects, into the server's receive list. However, this strategy produces additional overhead if the receiver list is processed in order to determine the sending process. At least one more indirection is required in order to locate the proper management structure which represents that process. Another solution would be the introduction of *virtual process objects*, as discovered with other decentralized/distributed operating systems, in order to continue with the usual receiver list management strategy. For each process that resides on a different SUPRENUM node one such object is necessary. Obviously, this leads to significant resource management problems. However, reducing the number of these objects does not really solve the original problem. What should be done if all objects are used at the time a new remote request comes in?

As outlined in the previous chapter, the *dispatching protocol* is based on a communication principle known since ALOHA [Abramson 1973]. Independently of a server's execution state and/or buffering capabilities, each client may pass a rendezvous request, using *send*, to the server. Maintaining a process and/or message buffer pool on peer nodes is not necessary, if all unsuccessful requests are consequently rejected to the originating site. Thus, whenever a *send* is initiated to a remote server process which is not ready to immediately receive this message, then this send request is rejected and the client is requested to retry it again. We consider this situation as the collision of service requests, because the server process presently executes some other service at the time a new request came in [9]. Instead of buffering this request at the server site, the client process will retry its request after a specific delay. The duration of this delay is determined by some empirical parameters returned along with the retry indication.

The retry request returned to the client site contains a collision counter, which actually represents the number of service collisions produced by the received *send* request. At each client site, the same delay interval is used to control the retry activity. This interval is scaled-up with the collision counter. Because each client will receive a different collision counter value, the entire retry delay is different, too. Applying this principle, the receiver list actually is spread (distributed) over the network and the client's position in this list is determined by the specific retry delay. More specifically, because the collision counter monotonically increases with each detected service collision, from the server's point of view the same queueing strategy, *first-come-first-serve*, is realized when compared to the local and per-server maintained receive list.

One drawback is accounted with this strategy. If the same server process is used for local as well as for remote requests, then starvation of processing remote requests may occur. Starvation in this case is unavoidable, because the server maintains a receive list of local clients, only. Remote clients virtually are queued at nodes which are different from the server's one and, thus, processing these clients is not directly controllable by the server. The replayed remote *send* request may overlap with a *send* request issued by a local client.

The starvation problem can be solved when a remote server is added to the original server team and when a specific dispatching strategy is associated with that additional process. Applying this principle, a remote dispatching activity is consequently separated from a local one and, additionally, uniquely bound to a specific process. This enables problem-oriented re-scheduling strategies based on processes, instead of considering special purpose network state informations.

Considering this principle in more detail, if for each remote client a representative is present within the server team, then starvation completely is avoided. Of course, this solution, again, is confronted with resource management problems at the server site. However, instead of maintaining virtual process objects within the message-passing kernel, a pool of processes is maintained within the server team. The starvation situation exists because of some problems accounted with the server team and, therefore, is only solvable in a problem-oriented way, i.e. designing the server team in such a way, that service-related starvation problems are reduced or avoided. This is a typical

[9] A service request in this situation is considered of having a twofold meaning. Firstly, a remote process is requested to serve an incoming message. And secondly, the message usually encodes some remote procedure call request, which, in the sense of PEACE, in turn designates a specific service request submitted for execution by the remote residing process.

example in PEACE of postponing certain design decisions, in this case the decision, on what level to place the queue for remote client.

3.5.3. Inter-Connecting Address Spaces

The basic primitives for inter-process communication are used to model a rendezvous between client and server processes, only. A rendezvous in PEACE is expressed either by a *send - receive - reply* or by a *send - accept* inter-relationship maintained between client, executing *send*, and server, executing *receive*, *reply* or *accept*. The actual communication activity, in the sense of transfering arbitrarily sized data streams, is performed by a different set of primitives. See [Schoen 1987] for more detail.

An important aspect is, that the primitives for high-volume data transfer enable the end-to-end message exchange without logically blocking the executing process. The rendezvous is used to connect two address spaces (i.e., teams) and, thus, enables the application of DMA principles for the transfer of arbitrary sized messages. Really initiating the data transfer always is the privilege of the server team, whose address space is connected to the client's one, i.e. a rendezvous is established. In contrast to the server, the client is blocked in this situation. More specifically, the client's execution state is completely controlled by the server team and, therefore, the server team is allowed to initiate high-volume data transfer to/from the client's address space.

The advantage of the rendezvous concept in order to maintain a permanent data flow connection is apparent, because the construction of high-performance end-to-end data transfer channels is made feasible. Once a connection has been established, data transfer actually is performed without any operating system intervention. The data transfer operations are directly mapped onto proper communication protocol activities within the PEACE message-passing kernel. As a consequence, application-oriented communication semantics in PEACE are implementable on a library basis.

In order to support many different communication principles, it is helpful for the client team to receive some kind of notification about the completion of a data transfer activity on the channel. This notification, of course, may be achieved releasing the connection, i.e. terminating the rendezvous. As a consequence, each high-volume data transfer activity dynamically is embraced by connection setup and release activities. This significantly would slow down the overall end-to-end data transfer performance.

In order to take advantage of a permanent data flow connection, the notification about the completion of a data transfer activity is produced locally at the client site. This notification is sent to a process residing inside the client's team. Especially in case of remote data transfer activities, additional network traffic is avoided for delivery of these events. The request for notification actually is *piggy-backed* with the high-volume data transfer request. More specifically, the notification itself is implemented as a *signal* upcall, which is to be executed by the peer PEACE message-passing kernel.

3.5.4. Security

Especially considering the principle of inter-connecting address spaces in PEACE, security aspects are of significance. The high-volume data transfer activity, initiated by a server process, may manipulate the entire address space of a client process. With this respect, authentication is required in order to guarantee that only a qualified server process may use the data flow connection to the client's address space.

Again, the rendezvous concept provides for sufficient authentication semantics. Obviously it suffices to guarantee, that the data flow connection is used only by that process, which the client has been asked for the establishment of a rendezvous. Because at each given point in time a client may stay only in a single rendezvous, the authentication check is straightforward.

Releasing a data transfer connection is controlled in the same way. Actually, the corresponding rendezvous must be terminated. This termination request is only granted to a server process, to which the respective rendezvous inter-relationship is still valid, i.e. the client either awaits *reply* or *signal*. A similar situation occurs with *relay*, which requests for a client a new rendezvous inter-relationship, maybe to another server process.

It is one of the most important functionalities of the *dispatching protocol* to control the access rights on address space inter-connections, or, more generally, to guarantee the integrity of rendezvous inter-relationships. This functionality, for instance, let the *remote procedure call protocol* level be sure, that, once received, a response message in each case has been transmitted (replied) by that server process from which it is expected to do so.

3.5.5. Protocol Data Units

The general protocol header for all level 2 protocol data units consists of the tripel {*type, client identifier, server identifier*}. The *type* member identifies one of the protocol data units listed in table 3.5. The other two members uniquely identify the client and server process involved in a rendezvous. The system-wide unique process identification is used for that purpose.

The *send*, *repeat* and *reply* protocol data units additionally carry fixed-size user data fields, actually with a length of 64 bytes. The protocol functions for high-volume data transfer, *movefrom* and *moveto*, are supplied with additional header information, which addresses a specific memory segment of the client's address space. This information consists of the tupel {*address, size*}.

3.5.6. Protocol Sequences

Although a lot of protocol functions are defined with the *dispatching protocol*, writing down the protocol sequences for the client and server sites is straightforward. The protocol sequence from the client's point of view is as follows:

Type	Meaning
send	exchange a fixed-sized message between client and server.
retry	return a *send* because the server process was not ready to receive the message.
repeat	represents the repetition of a previously *send*.
reply	terminate the rendezvous and deliver a message to the client process.
signal	terminate the rendezvous without delivery of a message.
reject	indicate that a previous *send* was not accepted by the server.
relay	represents the request of a *relay* from the server to the client of the actual rendezvous.
moveto	transfer a data stream to a remote node and verify the rendezvous.
movefrom	read a data stream from a remote node and verify the rendezvous.
probe	poll a node to return the actual protocol execution state of the *dispatching protocol*.
state	return the actual protocol execution state as response to *probe*.

Table 3.5: Dispatching Protocol Data Units

$$(\downarrow send \longrightarrow ((\uparrow retry \longrightarrow \downarrow repeat)(\uparrow relay \longrightarrow \downarrow send)\ \uparrow movefrom\ \uparrow moveto)*\ \uparrow reply\ \uparrow signal\ \uparrow reject)$$

Once *send* has been delivered, the most critical situation arises in case that *retry* is received. This situation indicates a service collision. Actually, with *repeat* the previously delivered send request is stated, again, after the client's delay time has been elapsed. Another important aspect is the acceptance of *relay*. In this case a previously sent message is transmitted again by *send*, however this time to a new server process. With *reject*, the non-existence of a specific server process is indicated. The functions *signal* and *reply* are used to terminate a rendezvous, more generally, to achieve remote dispatching of processes. With *movefrom* and *moveto*, the high-volume data transfer is initiated, for instance setting up the local i/o system, after the specified address space inter-connection has been verified. This verification also is performed with *signal* and *reply*.

The protocol sequence from the server's point of view is analogous. This sequence is as follows:

$(\uparrow send \rightarrow ((\downarrow retry \rightarrow \uparrow repeat)\ \downarrow movefrom\ \downarrow moveto)*\ \downarrow reply\ \downarrow relay\ \downarrow reject\ \downarrow signal)$

The main difference is given by the fact, that delivering a *relay* does not enforce the acceptance of the replayed message by the same node. Actually, a server residing on node B is capable of relaying a client residing on node A to another server residing on node C. The optional message supplied to *relay* is carried within the user data field of the corresponding protocol data unit. Once *relay* is received by the client site, the message immediately is forwarded to the new server process. The protocol sequence for the client reflects this functionality.

3.6. Data Transfer Protocol

The lowest level of the PEACE communication system, the *data transfer protocol*, is concerned with the typical functionality of basic network transport protocols. The purpose of this protocol is to transfer raw data streams with the most effective utilization of the underlying network system. In contrast to the *dispatching protocol*, which, in each case, addresses process objects as source and destination for the message transfer, the *data transfer protocol* uses node objects as the addressable units for the raw transfer of data streams.

The main problem, however, with which the design of the *data transfer protocol* is faced, is to cope with different qualities of the underlying transmission medium while keeping in mind such important requirements as minimal overhead and maximal effectiveness. The design decisions we met to fulfill these requirements are basically concerned with the strategy by which segmentation and retransmission of messages are handled. We will describe these design decisions in the following sub-sections.

3.6.1. Segmenting and Blocking

The typical requirement for segmentation of large messages is given by the fact that retransmission time of faulty received messages can be reduced. This is true for long-haul networks, but it needs not be true for local-area networks [Zwaenepoel 1985].

The need for message segmentation within the PEACE *data transfer protocol* is motivated by two aspects. The first aspects deals with retransmission of segment groups, in case of a faulty received message segment belonging to the corresponding group. This strategy is implemented with VMTP [Cheriton 1986] and represents a compromise between pure blast protocols and ordinary transport protocols. The second aspect deals with the fact, that not every basic network interface is able of transfering variable sized messages. This aspect is the main motivation for segmentation of messages with the *data transfer protocol*.

The design decision to consider segmenting and blocking within the *data transfer protocol*, and not to consider it of being part of the *virtual network interface*, is manifold. Firstly, the portability and/or adaptability of the entire PEACE communication system is improved when designing all significant protocol functionalities independent from a specific network hardware interface. Secondly, the lower-level network interface modules are more basic and, as a consequence, can be used to support a variety of high-level communication systems. Thirdly, the size of a single message segment both is of problem-oriented and of network interface-specific nature. In order to design a high-speed communication system, it is important of being able to map higher-level communication objects, possibly each having different sizes, onto lower-level services, straightforwardly. And finally, the design of various data transfer strategies is made feasible, with the consequence that the most powerful one can be selected by comparison of each strategy applied to the same basic network interface.

The most significant motivation for considering segmenting and blocking inside the PEACE communication system, however, is to support inter-networking on a very basic level. Because of using two different network systems for the construction of SUPRENUM, the PEACE communication system is faced with typical inter-networking problems. These problems are completely reduced to the need of buffering, as well as flow control strategies for managing the message buffer pool, on the corresponding network gateways, i.e. the communication nodes of a SUPRENUM cluster. The same *data transfer protocol* runs on both network systems and, thus, protocol conversion problems do not arise. In order to reduce buffering overhead on the gateway nodes, segmentation of messages is necessary. Additionally, using fixed sized segments, time consuming memory management strategies are avoided.

3.6.2. Dynamically Alterable Segment Groups

The SUPRENUM network system, especially the *cluster bus*, provides for a very broad communication bandwidth. This fact motivates a non-segmented message transfer following the principles of a *blast protocol* as described in [Zwaenepoel 1985]. Even in the case of transmission failures, simply transfering the entire message takes little time when compared to the management activities involved with segmenting on the client site and blocking on the server site and with selective retransmission of single message segments [10]. Especially solving the sequencing problem associated with segmented message transfer may cause a significant loss of overall communication performance. The principles of a blast protocol, however, only are applicable if the underlying communication hardware provides for high-performance and, more importantly, high-quality data transmission services. Thus, retransmission of messages is considered of being a very rare system activity.

Obviously, a blast protocol is very basic in nature. This aspect is a significant precondition for being able of representing these protocols in firmware, for instance as a dedicated microprogram. Precisely this step is planed with the *data transfer protocol* for

[10] Remembering the SUPRENUM performance specification, [Behr et al. 1986], to transfer 1 Kbyte data over the *cluster bus* takes about 8 μsec!

intra-cluster communication within SUPRENUM. It is well known that the migration of software components into firmware and/or hardware is only successful if the software components are well structured and of elementary nature. With this respect the PEACE *data transfer protocol* is designed.

As noted in the previous sub-section, the *data transfer protocol* is forced to use some kind of segmented message transfer in order to support buffer management on network gateways. With this respect, the bulk transfer of messages is dedicated to segment groups. Only one acknowledgement per segment group is returned. More specifically, confirmations explicitly are requested by the sending protocol entity and are not implicitly bound to a specific number of received message segments. This strategy, a management activity of the sending protocol entity, enables the problem-oriented construction of segment groups and/or can be used to generally improve communication reliability, because of additional inserted confirmation requests. A segment group may be constructed of one segment, only, or it may represent the entire message as a single segment. Additionally, the number of segments building a segment group is dynamically alterable, without leading to inconsistent protocol states in both protocol entities. The principle of explicitly requesting a confirmation from the receiving protocol entity makes this feasible. The transmission of a protocol data unit, whose confirmation explicitly is requested, completely is considered of being a service/management activity of the sending protocol entity.

3.6.3. Upcall of Service Functions

An important aspect of the *data transfer protocol* is its feasibility of transfering upcall requests for service functions to peer protocol entities. These upcalls actually are coded by the user data field of a specific *data transfer protocol* function and they are directly passed to the peer *dispatching protocol* level, using the upcall facility of the peer *data transfer protocol* entity.

Basically, the peer *data transfer protocol* activity in this situation is interrupted and a specific *dispatching protocol* activity is resumed. This latter mentioned activity is determined by the user data field of the protocol function qualified of carrying an upcall request. Once the peer *dispatching protocol* activity terminates, the peer *data transfer protocol* activity is resumed, again, and requested to return a specific result to the originating protocol entity.

This facility has its significance in driving specific protocol sequences which are closely related to specific service activities implemented by the next higher-level communication protocol. Typical examples for the PEACE communication system are the implementation of remote rendezvous as well as remote high-volume data transfer. Actually, using service function upcalls, traffic on the underlying transmission medium is reduced.

3.6.4. Protocol Data Units

Each protocol data unit is prefixed with the same header information. A header information is defined by the tripel {*type*, *route identifier*, *message identifier*}. The *type* member encodes one of the protocol functions listed in table 3.6. In order to identify both nodes involved in the data transfer, the *route identifier* is used. This identifier actually is structured as the tupel {*from*, *to*}, which denotes the source and destination node considered with the message exchange, correspondingly. The *message identifier* serves for two purposes. Firstly, is is used to distinguish message segments belonging to different segment groups. And secondly, it is used to check in-sequence acceptance of received message segments. Because of these two aspects, a *message identifier* itself is structured by the tupel {*group number*, *sequence number*}.

Type	**Meaning**
prepare	request a transfer channel to the peer protocol entity.
grant	indicate the allocation of a transfer channel.
refuse	indicate that no allocation of a transfer channel was possible.
finish	indicate the completion of a transfer activity and release the corresponding transfer channel.
data	transfer a variable sized message segment over a transfer channel.
probe	poll a peer *data transfer protocol* entity to return its actual protocol execution state.
state	return the actual protocol execution state as response to *probe* .

Table 3.6: Data Transfer Protocol Data Units

The protocol functions *probe* and *state* already has been explained, previously. The functions *prepare*, *grant*, *refuse* and *finish* are used for channel management purposes. Allocation of a transfer channel with *prepare* typically means to ensure the presence of enough memory in order to store the transfered message at the receiver site. With this respect, a transfer channel may be considered of being a DMA channel between different nodes. Actually, with each message transfer a *channel number* is associated, which is part of the *group number* member of the corresponding *message identifier*. A channel number is allocated upon request, by issuing *prepare*, and with *grant* the allocated channel number is returned to the *prepare* invoking site.

Service function upcalls are implemented by *prepare* and *grant*. The corresponding protocol data unit is qualified with the *upcall* attribute if its user data field contains a service primitive of the *dispatching protocol* level. This step is an additional service functionality of the *data transfer protocol* level and does not influence the normal protocol activities on this level.

3.6.5. Protocol Sequences

The protocol sequence expression describing the *data transfer protocol* is very simple. Actually, the transmission of message segments is embraced by a management activity, which verifies the transfer channel between memory segments residing on different nodes. Once the transfer channel is valid, i.e. the remote node (virtually) holds enough memory resources to store the incoming data stream, all message segments are blasted to the receiver site. The following protocol sequence describes the protocol behaviour for the sender site:

$$(\downarrow \mathit{prepare} \rightarrow \uparrow \mathit{refuse}\,(\uparrow \mathit{grant} \rightarrow \downarrow \mathit{data}[\mathit{grp}]* \rightarrow \uparrow \mathit{finish})+)(\downarrow \mathit{grant} \rightarrow \uparrow \mathit{data}[\mathit{grp}]* \rightarrow \downarrow \mathit{finish})$$

The allocation of a transfer channel is requested by *prepare*. Two alternatives are possible as the result of this protocol activity. The transfer either may be rejected, by *refuse*, or it may be allowed, by *grant*. The receive of *refuse* indicates that the allocation of a transfer channel was not possible by the peer protocol entity, i.e. (virtually) there will be not enough memory to store the message at the remote site. The receive of *grant*, however, indicates that the blast transfer of data streams is possible. The data stream is transfered by an arbitrary sequence of *data* and the entire transfer is acknowledged by a single protocol function, *finish*.

Each newly transmitted message segment uniquely is identified by a *message identifier*, which is supplied to *data*. Segment groups are constructed using the confirmed exchange of specific *data* units. With each new segment group a unique *group number* is allocated by the sending protocol entity.

The protocol for the receiver site works analogous. The following protocol sequence describes the protocol behaviour for the receiver site:

$$(\uparrow \mathit{prepare} \rightarrow \downarrow \mathit{refuse}\,(\downarrow \mathit{grant} \rightarrow \uparrow \mathit{data}[\mathit{grp}]* \rightarrow \downarrow \mathit{finish})+)(\uparrow \mathit{grant} \rightarrow \downarrow \mathit{data}[\mathit{grp}]* \rightarrow \uparrow \mathit{finish})$$

The decision for a *refuse* will take place if there is no chance of storing the message, whose transfer is initiated by *prepare*. Actually, the transfer size is delivered with *prepare* to enable the necessary checks at the receiver site. This transfer size is remembered in order to decide the delivery of *finish*.

Generally, a faulty received message segment results in signaling to the lower-level communication sub-system the abortion of further message receives of the same group. The consequence of this strategy is a corresponding indication at the peer interface to the sending protocol entity. This indication is considered of being a protocol (and/or service) exception and, thus, the sender's normal protocol sequence is interrupted. Handling this kind of exceptions results in the abortion of further message transfers and enables the retransmission of the actual considered segment group.

It should be noted that these *data transfer protocol* activities completely are controlled using specific services of the next lower-level communication sub-system. In case of the *cluster bus*, the word transfer level, actually residing on level 0 of the PEACE communication system, implements these services. Thus, no specific protocol data units of the *data transfer protocol* are exchanged to stop further transfer of messages.

Both protocol sequences take care of the fact, that flow-control is allowed in order to circumvent certain management difficulties at the receiving site. The flow-control semantic is simply expressed by a series of *grant* and *finish*, likewise embracing the data transfer activity of the presently receivable message segment. In addition to that, the second (rightmost) main alternative of each of these protocol sequences describes the protocol functioning once a permanent transfer channel has been established. The transfer activities over this channel need not be enabled by a leading *prepare*. This facility is used in order to map level 2 *movefrom* requests, efficiently.

3.7. Virtual Network Interface

In order to adapt the PEACE communication system at different network hardware interfaces a specific abstraction level is introduced. This level, the *virtual network interface*, hides network specific hardware characteristics from the higher-level communication system.

With SUPRENUM, four different network systems are considered. Besides the main inter-connection bus-systems, the *cluster bus* and *SUPRENUM bus*, two alternatives are present for replacing the *SUPRENUM bus*. These alternatives are *Ethernet* and the *pipeline bus*. Additionally, the *cluster bus* interface actually is a multi-stage design. The consequence is, that, in the early project phase, the PEACE communication system bases on a much more basic network interface than given with the finally *cluster bus* interface. Basically, the word transfer level of the *cluster bus* defines this network interface. A software emulation of the finally *cluster bus* interface bridges the gap between the present and future hardware interfaces.

In this sub-section we describe the functionality of the *virtual network interface*. This description less is concerned with the illustration of specific communication protocols, but rather with the illustration of specific functionalities (services), whose fulfillment are required by level 0 of the PEACE communication system. Of course, in order to provide these services, specific communication and/or management protocols might be necessary. However, these protocols are not mandatory by the PEACE communication system. To give an example, the word transfer protocol is represented by software components as long as the design and implementation of the entire *cluster bus* interface is not completed. Another example is the *SUPRENUM bus*, whose firmware/hardware enables the transfer of variable sized messages by small containers, a principle as typically discovered with ring-bus systems.

3.7.1. Expedited Data Flow

It is of significant importance for the continuous functioning of a distributed operating system to deliver and/or accept system service requests, independently of the actual network system load. With non-distributed operating systems this problem does not arise, because, in order to request system services, no network communication is necessary.

In order to guarantee delivery and/or acceptance of service requests, at least two different data channels are considered with the PEACE communication system. One channel controls a *normal data flow*, i.e. the transmission of non-critical messages. An example is the communication between different application complexes and/or reading (writing) of data streams from (into) files. The other channel controls an *expedited data flow*. In contrast to the previously mentioned one, this channel guarantees the transmission of messages, even in the presence of certain bottlenecks inside the communication system. For example, independently of any buffering problem, a message may be passed to a peer protocol entity. Additionally, certain sequencing and/or flow control strategies inside the communication system are bypassed.

The PEACE *data transfer protocol* is designed for using at least two different data channels for message exchange. These channels also are visible at the interface between *dispatching protocol* and *data transfer protocol* in terms of virtual data channels. The *dispatching protocol* transfers system messages using the expedited data flow channel. The normal data flow channel is used by this protocol to transfer ordinary user messages and to manage high-volume data transfer between different nodes.

3.7.2. Exception Propagation

Although expedited data flow channels are available with the PEACE communication system, there can't be an absolute guarantee that communication between peer protocol entities still works correctly in the sense that protocol data units are exchanged between the corresponding parties. For reliability reasons, a communication system should be designed in such a way that, independently of any protocol state, certain exceptional conditions may be propagated to a peer protocol entity. With this demand, even in the case of, for instance, a deadlock within the *data transfer protocol*, peer protocol entities can be notified about specific protocol conditions. The notification is achieved in using lower-level communication services, instead of trying to exchange protocol data units by an, eventually, out of order higher-level protocol machine.

The propagation of exceptional conditions to peer protocol entities, using specific services of a lower-level communication system, is one of the main functionalities of the *virtual network interface*. Using these propagation services, further transmission of message segments is inhibited once a faulty received message segment has been detected on the *data transfer protocol* level. The propagation of exceptional conditions to peer protocol entities actually interrupts the protocol activities of these entities. A problem-oriented exception handler may be declared on the *data transfer protocol* level in order to handle the (from remote) signaled protocol exception.

3.7.3. Descriptor Chaining

Overhead in programming a network interface can be significantly reduced if a set of message segments are manageable by the lower-level network hardware, instead of processing one i/o request at a time – the i/o request is stated once for the entire set. On the other hand, the hardware will process this segment set, i.e. empty and/or fill data, independently from any activity taking place inside the higher-level communication system. The finally *cluster bus* interface will provide for such functionalities, and there are many other examples of state of the art i/o controller interfaces offering similar services.

The *data transfer protocol* considers segment groups as the i/o request unit. Actually, a descriptor queue represents the interface between this protocol level and the *virtual network interface* level. At this interface arbitrarily sized data segments are exchanged. The elements of the descriptor queue, i.e. the single descriptors, are designed of being basic DMA descriptors, as they are processable by the finally *cluster bus* interface.

A segment descriptor is used to address data which is ready for transmission and to denote a free memory area into which data, received from the network, is transfered. From the SUPRENUM application point of view it is important to note, that a descriptor is used to transfer regular structured data segments. A typical application is the transfer of a row, column and/or diagonal of multi-dimension arrays, without any intermediate formatting activity except the creation of the proper message descriptor. Using different descriptors for the transfer of representation information associated with each object, reduces additional buffering overhead while assembling a specific transfer segment. Of course, this strategy is only favourable if relative large data objects should be transfered. Generally, descriptor allocation/deallocation time is high when compared to the time it would need to assemble (copy) small data objects.

Chapter 4

Service Functions and Mappings

The previous chapters generally gave an illustration of the basic PEACE communication system and a description of the corresponding communication and management protocols. In this chapter, the interactions between the different protocol layers are explained in terms of service functions invoked at dedicated interfaces. It is explained, how service data units, for example the fixed-size messages transfered by the PEACE message-passing kernel, are mapped onto lower-level protocol data units.

In order to keep the description of the service functions and/or mappings small and clear, a subset of the PEACE communication system is considered, only. This subset represents the communication system kernel of PEACE and, thus, is of main interest for any assessment of the inter-node and/or inter-cluster communication within PEACE.

4.1. Hierarchy of Service Data Units

In order to illustrate the mapping of service data units, we will show how a *remote procedure call protocol* data unit, step by step, is represented by lower-level communication protocols. This protocol data unit is always represented by a fixed-size message object, which, in turn, is directly transportable using the inter-process communication primitives of the PEACE message-passing kernel. Figure 4.1 shows the mapping of these message objects.

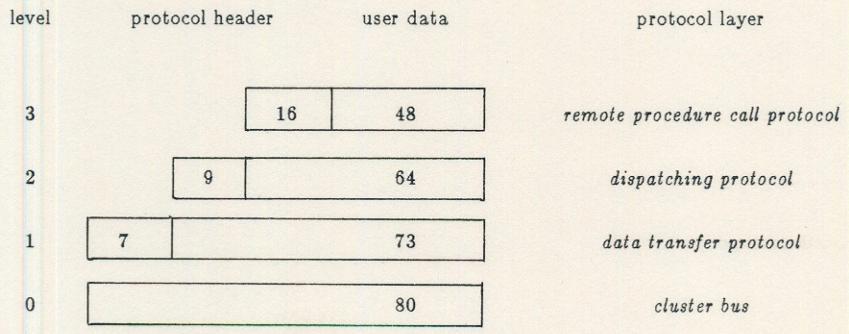

Figure 4.1: Mapping of *dispatching protocol* Service Data Units

On level 3, the *remote procedure call protocol,* the 64 byte fixed-size message object is structured into a 16 byte protocol header, consisting of the tripel {*type, service identifier, call identifier*}, and a 48 byte service data unit. This service data unit contains marshaled procedure arguments and/or results, i.e. its contents is of strictly system call (system service) specific nature. The entire level 3 protocol data unit is considered of being a service data unit on level 2, the *dispatching protocol*. A 9 byte protocol header, the tripel {*type*, *client identifier*, *server identifier*}, is added to this service data unit. The level 1 protocol data unit in this mapping example, i.e. the representation of a level 3 protocol data unit on the *data transfer protocol* level, additionally requires a 7 byte protocol header. This header is represented by the tripel {*type*, *route identifier*, *message identifier*}.

This mapping example shows, that the PEACE communication system kernel produces additional protocol header information overhead of 16 bytes. Thus, a remote procedure call request totally requires 80 bytes to be transfered by level 0[11].

4.2. Protocol Mappings

Each hierarchically structured communication system requires a mapping of service functions, for instance *send*, onto lower-level protocol sequences in order to execute, or more precisely to implement, the corresponding service function. In the following subsections we will describe such mappings and, in each case, figure out the general protocol overhead necessary to implement a specific service function.

4.2.1. Fundamental Transfer Units

Considering the standard SUPRENUM inter-connection bus systems, the *cluster bus* and *SUPRENUM bus*, basically two different sized transfer units are visible on the physical level. The *cluster bus* is capable of transfering 64 bit wide words at a time, whereas the *SUPRENUM bus* uses actually 64 bytes or 128 bytes per container.

In order to transfer a series of 64 bit words, the *cluster bus* transfer activities are controlled by a special purpose *word transfer protocol*. This protocol is very close to the hardware capabilities of the *cluster bus* and is represented by a specific microprogram. The functionality of the *word transfer protocol* is very low. Actually, it may be considered of being a datagram protocol. In order to drive this protocol, and, thus, making variable sized transfer units available on the *cluster bus*, single 64 bit wide words are used by the hardware. The consequence is a general transfer overhead of 3 *cluster bus* cycles, when considering the transfer of a single *data transfer protocol* data unit. This protocol data unit, maybe of variable size, always is a level 0 transfer unit (service data unit). Therefore, in addition to the 3 *cluster bus* cycles, at least 1 more cycle is required in order to transfer a *data transfer protocol* data unit which exactly would fit into a 64 bit wide *cluster bus* word. In other words: each transfer of a *data transfer*

[11] More precisely spoken, because the *cluster bus* transfers 64 bit wide words at a time, a total of 10 words on a *cluster bus*, i.e. 10 cycles, suffices in order to carry these requests. The size of one *SUPRENUM bus* container is 128 bytes, which means that a single container is used to transfer a remote procedure call request.

protocol data unit at least takes 4 *cluster bus* cycles.

4.2.2. General Data Transfer

The basis of all *dispatching protocol* activities is the presence of a transfer channel. This channel is maintained by the *data transfer protocol*. In order to enable the transfer of arbitrarily sized data streams, at least two protocol activities are mandatory. These activities are *prepare*, to request and/or verify the channel allocation, and *grant*, which indicates that the transfer may be started. Once the data stream has been transfered, *finish* is executed by the receiving protocol entity. These three *data transfer protocol* activities are performed along with each transfer of a *dispatching protocol* data unit. Thus, a general transfer overhead of

$$\alpha = \text{sizeof}\,(\,\mathit{prepare}\,),\ \ \beta = \text{sizeof}\,(\,\mathit{grant}\,) + \text{sizeof}\,(\,\mathit{finish}\,)$$

is accounted for the *data transfer protocol* in order to manage a transfer channel. The additional overhead for each data transfer activity is determined by

$$\gamma = \text{sizeof}\,(\,\mathit{data}\,).$$

With α, the overhead for transfer channel allocation is expressed, whereas β indicates the overhead for enabling and disabling data transfer on a single transfer channel.

The overhead accounted with pure data transfer, in case of segmentation, is determined by formulas representing the transfer of complete segments as well as the transfer of a residual data stream. With each message segment, the *data* protocol header represents the additional transfer overhead of the *data transfer protocol*. Let

$$\nu = \text{sizeof}\,(\,\mathit{message}\,)\ \ \mathbf{div}\ \ \text{sizeof}\,(\,\mathit{segment}\,)$$

be the number of complete segments and

$$\kappa = \text{sizeof}\,(\,\mathit{message}\,)\ \ \mathbf{mod}\ \ \text{sizeof}\,(\,\mathit{segment}\,)$$

be the size of the residual data stream. Then the transfer overhead for blasting an entire message is

$$\sigma = \nu * (\,\gamma + \text{sizeof}\,(\,\mathit{segment}\,)\,) + \gamma + \kappa.$$

The design of the *data transfer protocol* guarantees, that each of its protocol data unit header informations are transferable using a single *cluster bus* transfer unit, i.e. exactly one *cluster bus* cycle is required for transfering a *data transfer protocol* header and, thus, actually driving the *data transfer protocol*. For example, in order to transfer a 4 kbyte message, for instance based on 1 kbyte message segments, a total overhead of 16 *cluster bus* cycles[12] is produced during the transfer phase, i.e. in addition to the 512 *cluster bus* cycles accounted for the entire message transfer.

[12] Remember the total of 4 *cluster bus* cycles in order to transfer the *data* protocol header.

The dynamical allocation and/or verification of a transfer channel is necessary even in such situations, where connection-oriented bus-systems, for instance the *SUPRENUM bus* and *pipeline bus*, are used as the underlying transmission medium. A transfer channel actually is a management functionality of the *data transfer protocol*. This management functionality enables end-to-end transfer of data streams between peer address spaces, without the need of buffering inside the communication system. The end-to-end transfer of data streams is related to the data stream producing and consuming processes, i.e. teams in the sense of PEACE. In case of inter-networking, i.e. performing inter-cluster communication within SUPRENUM, the transfer channels are used to implement specific flow-control strategies, in order to support problem-oriented management of network gateway buffer space.

4.2.3. Remote Rendezvous

A rendezvous is modeled using the PEACE message-passing kernel primitives *send*, *receive*, *reply* and/or *accept*. These primitives are considered of being *dispatching protocol* service functions. Executing these primitives without passing node boundaries, i.e. performing a local rendezvous, enables the transfer of 128 bytes user data information – 64 bytes with *send* and 64 bytes with *reply* and/or *accept*.

Passing node boundaries with a rendezvous, means driving the *dispatching protocol* and the *data transfer protocol*. The *dispatching protocol* overhead is determined as the exchange of

$$\text{sizeof}\,(\,\textit{send}\,) + \text{sizeof}\,(\,\textit{reply}\,)$$

bytes of protocol header information. Each of these protocol data units are service data units for the *data transfer protocol* level. Considering the protocol sequence for this level, i.e. taking into account the overhead produced by the exchange of *data transfer protocol* header information, a complete rendezvous requires the transfer of

$$2 * (\,\alpha + \beta + \gamma + \text{sizeof}\,(\,\textit{message}\,)) + \text{sizeof}\,(\,\textit{send}\,) + \text{sizeof}\,(\,\textit{reply}\,)$$

bytes of data over the underlying transmission medium. Inserting the actual size for each protocol header, a total of 208 bytes for the entire rendezvous is accounted. Actually, 104 bytes are transfered for a *send* and/or *reply* over the *cluster bus*.

These accounts consider the 8 byte fixed-size *cluster bus* transfer unit. As a consequence, 26 *cluster bus* cycles are used for the entire rendezvous, i.e. 13 bus cycles for each *send* and/or *reply*. The pure transfer overhead, as it is dictated by the different communication protocols, is determined with 31 bytes for a single fixed-sized message of the PEACE message-passing kernel, instead of 40 bytes overhead because the size of a single *cluster bus* transfer unit.

4.2.4. Remote High-Volume Data Transfer

Besides the primitives for modeling a rendezvous, the primitives for high-volume data transfer are of significant importance for the overall performance of the PEACE message-passing kernel. The primitives, which are of interest for a remote high-volume data transfer, are *movefrom*, in order to read out of a peer address space, and *moveto*, in order to write into a peer address space. Both primitives, however, are only applicable during a rendezvous, i.e. during a synchronized inter-relationship between the two address spaces.

In order to implement remote high-volume data transfer, the upcall facility of the *data transfer protocol* is considered. This facility is applied to different protocol functions, depending on the transfer direction. The following two sub-sections describe the scenario for *movefrom* and *moveto*, separately.

4.2.4.1. Reading from a Remote Address Space

With *movefrom*, the address space of a client is read. This level 2 protocol function is executed as an upcall using specific *data transfer protocol* services in the following fashion:

1. request the transfer of *movefrom* to the peer *dispatching protocol* entity, using a specific upcall service function provided by the *data transfer protocol*;
2. issue *grant*, qualified with *upcall* and carrying the *movefrom* protocol data unit within its user data field, to the peer *data transfer protocol* entity;
3. once *grant* has been accepted by the peer *data transfer protocol* entity, pass the upcall request *movefrom* to the *dispatching protocol* level, in order to verify the addressed client process for an existing rendezvous with the requesting server process;
4. in case that no rendezvous exists, immediately terminate the transfer request by *finish*, qualified with *upcall* and carrying a failure indication. In case of an existing rendezvous, immediately enter the blast transfer phase;
5. terminate the blast transfer phase with *finish*.

Considering this mapping scenario, the entire upcall and channel management overhead is defined by

$$\beta + \sigma + \text{sizeof}\ (\ movefrom\ ).$$

A total of 5 *cluster bus* transfer units are used to handle the *movefrom* upcall. The overhead σ produced during the blast transfer phase is dependent on the actual segment size, which determines the number of *data* transfer activities. The minimal protocol overhead will be 1 *cluster bus* transfer unit, if the entire message is transfered without any segmentation.

4.2.4.2. Writing into a Remote Address Space

With *moveto*, into the address space of a client may be written. As with *movefrom*, this *dispatching protocol* function is executed as an upcall using specific *data transfer protocol* services. However, the mapping sequence differs from the corresponding one given for *movefrom*. The following steps are performed to execute a remote *moveto*:

1. request the transfer of *moveto* to the peer *dispatching protocol* entity, using a specific upcall service function provided by the *data transfer protocol*;
2. issue *prepare*, qualified with *upcall* and carrying the *moveto* protocol data unit within its user data field, to the peer *data transfer protocol* entity;
3. once *prepare* has been accepted by the peer *data transfer protocol* entity, pass the upcall request *moveto* to the *dispatching protocol* level, in order to verify the addressed client process for an existing rendezvous with the requesting server process;
4. in case that no rendezvous exists, return a corresponding failure indication using *refuse*, qualified with *upcall*. In case of an existing rendezvous, return *grant* and await the blast transfer of data streams;
5. in case of the acceptance of *refuse*, return the failure indication to the *dispatching protocol* level. In case of the acceptance of *grant*, enter the blast transfer phase;
6. terminate the blast transfer phase with *finish*.

When compared to the mapping scenario of *movefrom*, this mapping scenario shows that one more *data transfer protocol* activity, namely *prepare*, is performed. The overhead produced by the upcall facility actually is

$$\alpha + \beta + \sigma + \text{sizeof}\ (\ \textit{moveto}\ ).$$

The consequence is a total of 6 *cluster bus* transfer units in order to handle the *moveto* upcall. As already outlined in conjunction with *movefrom*, the overhead σ produced during the blast transfer phase is dependent on the actual segment size, which determines the number of *data* transfer activities. The minimal protocol overhead will be 1 *cluster bus* transfer unit, if the entire message is transfered without any segmentation.

4.3. Flow-Control

As long as no inter-networking activities are necessary with inter-process communication in PEACE, the end-to-end significance of the message exchange has as its consequence, that no buffering inside the PEACE communication system is required. However, with inter-networking, which takes place with each inter-cluster communication in SUPRENUM, buffer management activities are required on the corresponding gateway nodes. Because buffer space is limited on the nodes, flow-control mechanisms are required, in order to implement a virtually unlimited buffer space.

We consider flow-control as being a management activity of the *data transfer protocol*. Using the standard protocol functions for maintaining transfer channels, flow-control is implementable. Once a transfer channel is active, the transfer of data streams is terminated by *finish* on behalf of the receiving site. If delivered because a temporary run out of buffer space, *finish* is qualified accordingly. The next *grant* delivered to the sending site may re-enable the transfer of further data streams over the flow-controlled transfer channel. In order to support channel-oriented flow-control strategies, *grant* as well as *finish* are qualified with a specific transfer channel identification.

4.4. Store and Forward Principle

Besides the flow-control aspect associated with SUPRENUM gateways, there is another important aspect in order to implement problem-oriented communication semantics. If inter-networking over at least two gateways is performed, buffer space may be reclaimed on predecessor gateways applying the store and forward principle.

The units considered of being stored and forwarded are represented by a *data transfer protocol* segment group. This, logically, is the consequence of the fact, that segment groups are the retransmission units of the *data transfer protocol*. Additionally, because segment groups are dynamically alterable in its size, the actual gateway load may be balanced, using segment groups of different sizes.

Once a store unit, i.e. a segment group, has been forwarded, the allocated buffer space may be reclaimed by the forwarding gateway node. Thus, the corresponding segment group actually is discarded. The original segment group, however, still is stored at its source node. This save copy is released only is case of the complete acceptance of the entire message by the destination node. Thus, end-to-end significance of a message transfer still remains present, independently of any special purpose inter-networking activity.

Because of this very special protocol functionality, a dedicated *gateway protocol* is required. This protocol is very close to the *data transfer protocol*, more specifically, it enriches the original *data transfer protocol* level by a very specific set of management activities.

Chapter 5

Concluding Remarks

This paper described the structure of the basic PEACE communication system. We have shown the necessity of the different protocol layers in order to realize a decentralized/distributed PEACE operating system on top of SUPRENUM. The following sub-sections will review the most important aspects of the PEACE communication systems design.

5.1. Faster Nodes are Needed

The design of the PEACE communication system has been mainly influenced by the results of [Lantz et al. 1984][13]. The central statement of this work is, that *faster hosts are needed*. In the sense of the PEACE communication system, hosts are represented by SUPRENUM nodes. One might say, that more functionality should be associated with lower-level network hardware interfaces in order to relieve the nodes from cumbersome protocol and/or management activities. However, this strategy actually would not be as promising as it seems.

Independently of the network hardware functionality, there are always node-related activities necessary in order to drive this interface. The activities might be interrupt driven and, usually, will effect the execution state of certain processes. Thus, special synchronization as well as dispatching and/or scheduling activities are required from the operating system. In conjunction with interrupt processing, time consuming buffer management strategies usually are necessary. These aspects are not only close to the PEACE operating system, moreover, they are valid for all (non-) decentralized/distributed operating systems. The overhead produced, significantly reduces the overall network communication throughput from/to the corresponding nodes. As a consequence, synchronization and dispatching strategies, as well as protocol functionalities, are forced of being very basic in nature. Without this requirement, the best possible utilization of the underlying communication system is not made feasible, at all. The PEACE message-passing kernel consequently is designed with this respect, however at the expense of less security and functionality.

[13] A more detailed performance analysis of distributed applications can be found in [Lantz et al. 1985].

5.2. Postponing Design Decisions

From the software design point of view, the only promising way to achieve a high-performance communication system is to find a structure and system organization, with which little operating system activities are necessary in order to drive the network protocols. Besides the very basic PEACE message-passing kernel, another consequence would be, for instance, to completely avoid any buffering and memory management activities during a message transfer. These activities always slow down the overall communication throughput.

Instead of buffering inside the communication system – excepted within gateway nodes – we promote end-to-end data transfer techniques. Actually, we burden higher-level software components with the necessity of buffering. The extrem situation arises in case that application processes directly manage their buffer pools in an application-dependent manner. The lower-level system components merely verify the existence of these memory resources. Thus, the design decision for complex and cumbersome buffering strategies, from the communication systems point of view, has been postponed and is considered of being a higher-level system and/or application functionality. This principle consequently follows the pattern of [Parnas 1975] and [Habermann et al. 1976], and it promotes the construction of application-oriented operating systems.

5.3. Separation of Concerns

In PEACE, high-volume data transfer activities are separated into two main functionalities. Firstly, the rendezvous concept is used in order to inter-connect different address spaces, thus establishing a permanent and low-overhead data flow channel between client and server team. And secondly, specific and non-blocking primitives are used for blast data transfer, without the necessity of additional resource allocation. That processes are used in PEACE in order to establish such connections, does not mean, that more communication overhead is produced. This connection management does not require any other special operating system functionalities except synchronous inter-process communication based on the rendezvous concept. Moreover, high-volume data transfer is performed without any intermediate process switch.

It is a natural consequence to choose the most basic construction principle in order to build up more enhanced communication functionalities. We consider synchronous message-passing based on the rendezvous concept as the most basic communication principle, because with each communication activity end-to-end arguments always are present. In the last analysis, different user processes want to communicate with each other, which obviously means the information transfer between the user address spaces. Asynchronous message-passing in each case would slow down the information transfer, because of intermediate buffering. The only advantage of asynchronous message-passing would be the increased parallelism in process execution. This aspect, however, is completely handled using the team concept of PEACE.

Considering asynchronous message-passing in more detail, separation of concerns means the application of two different principles in order to achieve more parallelism in process execution. Thus, we generally promote a design decision with which parallelism is expressed by processes, instead of using specific and overloaded mechanisms for message-buffering. This, too, is the most basic design principle, because processes always represent the thread of control within any operating and/or application system.

5.4. Present State

This paper actually is a design paper. Experiences from designing other decentralized/distributed operating systems, especially V [Cheriton 1984] and AMOEBA [Mullender, Tanenbaum 1986], had been considered in order to design the SUPRENUM operating and communication system. First experiences with the non-decentralized PEACE message-passing kernel underpin the main design concepts. Actually, a local rendezvous on top of a SUPRENUM node performs in approximately 400 μsec when applying the *send - receive - reply* model.

The decentralized version of the PEACE message-passing kernel presently is in development. A first prototype, running on upto 9 SUPRENUM nodes, has been completed. With this version, the principle working of the *dispatching protocol* was shown. Presently, the *word transfer protocol* of the *cluster bus* is implemented in software, i.e. it is completely driven by the central processing unit instead of running a dedicated microprogram and using state of the art DMA interfaces. Based on this software emulation package, a remote rendezvous performs in approximately 1500 μsec. A significant speed-up is expected if this emulation package is replaced by firmware/hardware components.

We are confident, that the design of the entire PEACE communication system will adequately drive SUPRENUM, especially with respect to high-performance communication. Future work will show if we are right with the PEACE concept. The state of the art in distributed operating system design, as can be taken from [Mullender 1986], presently shows that we are on the promising way.

Bibliography

[Abramson 1973]
N. Abramson: **Packet Switching with Satellites**, AFIPS Conference Proceedings 42 (June), 695-702, 1973

[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

[CCITT 1984]
CCITT: **Interface Between Data Terminal Equipment (DTE) and Data Circuit Terminating Equipment (DCE) for Terminals Operating in the Packet Mode and Connected to Public Data Networks by Dedicated Circuit**, Recommendation X.25, 1984

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

[Cheriton 1986]
D. R. Cheriton: **VMTP: A Transport Protocol for the Next Generation of Communication Systems**, Proceedings of the SIGCOMM '86 Symposium, Stowe, Vermont, August 5 - 7, 1986

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

[Franke 1986]
B. Franke: **Pipeline Bus Anschluβ**, SUPRENUM Technischer Bericht, Institut für Datenverarbeitung, Technische Universität Braunschweig, 1986

[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

[Hahn, Herrtwich 1983]
J. Hahn, R. G. Herrtwich: **Formal Specification with RSPL-Z – A Tutorial Introduction**, Technical Report 83-14, TU Berlin, Fachbereich 20 (Informatik),

KBS, 1983

[ISO 1985]
ISO: **Information Processing Systems – Open Systems Interconnection – Basic Reference Model**, International Standard 7498, 1985

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

[Lantz et al. 1984]
K. A. Lantz, W. I. Nowicki, M. M. Theimer: **Factors Affecting the Performance of Distributed Applications**, Proceedings of the SIGCOMM '84 Tutorials and Symposium, Montreal, Quebec, Canada, June 6 - 8, 1984

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

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

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

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

[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

[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

[Saltzer et al. 1984]
J.H. Saltzer, D.P. Reed, D.D. Clark: **End-To-End Arguments in System Design**,

ACM Transactions on Computer Systems, Vol. 2, No. 4 (November), 277-288, 1984

[Schoen 1987]
F. Schön: **Hochvolumendatentransfer in PEACE**, SUPRENUM Memo, GMD FIRST an der TU Berlin, 1987

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

[Schroeder 1987]
W. Schröder: **Basic Inter-Process Communication in a Decentralized Operating System**, Technical Report, GMD FIRST an der TU Berlin, to be provided, 1987

[Schroeder 1987b]
W. Schröder: **Ligth-Weigthed Processes and the Concept of Teams**, Technical Report, GMD FIRST an der TU Berlin, to be provided, 1987

[Schroeder 1987c]
W. Schröder: **Naming and Identification of Services in a Distributed Operating System**, Technical Report, GMD FIRST an der TU Berlin, to be provided, 1987

[Terry 1984]
D. B. Terry: **An Analysis of Naming Conventions for Distributed Computer Systems**, Proceedings of the SIGCOMM '84 Tutorials and Symposium, Montreal, Quebec, Canada, June 6 - 8, 1984

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