# The PEACE Family of Distributed Operating Systems*

*R. Berg, J. Cordsen, J. Heuer,*
*J. Nolte, B. Oestmann, M. Sander,*
*H. Schmidt, F. Schön,*
*W. Schröder-Preikschat*

GMD FIRST
Hardenbergplatz 2, 1000 Berlin 12, FRG

ABSTRACT

Massively parallel systems are based on distributed memory concepts and consist of several hundreds to thousands of nodes interconnected by a very high bandwidth network. Making these systems work requires a very careful operating system design. A distributed operating system is required that takes the form of a functionally dedicated server system. This approach reduces system overhead on the nodes and enables a problem-oriented mapping of system services onto the distributed hardware architecture. The program family concept for operating system construction, combined with a novel virtually shared memory approach, then provides a powerful basis for parallel and distributed computing.

## 1. Introduction

Massively parallel systems consist of several hundreds to thousands of nodes interconnected by a very high bandwidth network. They are multicomputer systems with distributed control, built by autonomous, cooperating nodes. At the lowest level, there is no longer the view of a global, common address space, as is the case with tightly coupled shared memory multiprocessor systems. Rather, each node has only direct access to its local, on-board memory. Global memory access and, thus, inter-node cooperation, happens exclusively by means of system-wide message passing.

In general, the notion of a distributed memory architecture comprises distributed systems capabilities, with all their advantages and problems. That is, operating system design aspects for distributed systems [Tanenbaum, van Renesse 1985] also apply to massively parallel systems [Schroeder, Gien 1989]. An important role plays *transparency*. At least, this requires to hide the way of locating and accessing system services, especially in cases where service replication is given. Making and keeping massively parallel systems work also requires some means for fault tolerance [Randell et al. 1978].

* This work was supported by the European Commission under the ESPRIT-2 program, grant no. P 2702, and by the Ministry of Research and Technology (BMFT) of the German Federal Government, grant no ITR 9002 2.

At the other end of the spectrum, *efficiency* is predominant. The startup time of a message passing operation is the most crucial factor with respect to overall communication performance [Mierendorff 1989]. Because message passing is fundamental, achieving very high performance, i.e. very short communication latency, is a must.

In order to improve the acceptance of massively parallel systems – and of distributed systems in general –, the buzzword *virtually shared memory* [Li 1986] stands for a central mechanism which must be supported by todays distributed operating systems. For an overview of state-of-the-art designs and implementations see [Ousterhout 1989].

This paper describes concepts for distributed operating systems of massively parallel systems. The elaboration of these concepts was based on experiences made with the design, development and implementation of PEACE [Schroeder 1988], the distributed operating system for SUPRENUM [Giloi 1988]. It is explained by what means PEACE copes with the tradeoff between efficiency and transparency.

The basic PEACE approach is to consequently follow the well-established software engineering rules for the development of program families, i.e. to identify a *minimal subset* and *minimal extensions* of system functions [Parnas 1979]. In order to deal with the structure and complexity of massively parallel systems, these rules are applied to construct a family of distributed operating systems. In a similar way, a message-passing kernel family lays the basis for providing transparency without loosing efficiency. In this sense, PEACE aims to be a general solution for current, as well as future, massively parallel systems based on distributed memory.

After a description of basic abstractions provided by, and the design principles applied to, PEACE, the global system organization is discussed. Subsequently, mechanisms for service invocation and object localization are considered. By what means the system is getting started and object mobility (i.e., migration, checkpointing/recovery, virtual memory and virtually shared memory) is supported will be explained afterwards. Some concluding remarks complete the paper.

## 2. Basic Abstractions

The primary purpose of PEACE was and is to provide a **p**rocess **e**xecution **a**nd **c**ommunication **e**nvironment for large scalable distributed applications. A minimal subset of system functions should be applicable for making both user and system applications work on massively parallel systems.

In this sense, it was not intended to construct another general purpose distributed operating system. Rather, the focus was on fundamental mechanisms that make the construction of a large class of distributed applications feasible. Special concern was hereby devoted to the extension of traditional operating system family concepts [Habermann et al. 1976] into the area of distributed and massively parallel systems. In the following subsection, the basic abstractions employed to build the PEACE distributed operating family are described.

### 2.1. Threads and Teams

Following state-of-the-art distributed operating systems, above all V [Cheriton 1984] and Amoeba [Mullender, Tanenbaum 1986], the PEACE process model distinguishes between *heavyweight process* and *lightweight process*. Heavyweight processes are *teams* of lightweight processes, i.e., shells for one or more instances of *threads*. The term *process* is used as a synonym for either thread or team.

Teams define a global execution domain for threads. This comprises a common address space, uniform team scheduling strategies, and common access rights onto PEACE objects. These objects are, e.g., memory segments, teams, threads, files, devices, and so on. Team scheduling always applies to a number of threads and is triggered on a time slice basis. In addition, different teams may be given different time slices and may be scheduled following different strategies. With threads being mapped onto different processors, true parallel processing of a team is supported in case of shared memory multiprocessor systems.

In PEACE, the potential for thread scheduling is given only by two situations, namely as a side-effect to the reception of either messages or events [Hewitt 1977]. One extreme is that threads are preempted and/or immediately started upon receiving a message. The other extreme, which is the default case, is that threads behave like coroutines and are scheduled in a round-robin fashion.

A thread is the unit of execution, whereas a team is the unit of distribution. Consequently, different threads of the same team are not distributed over different nodes. This is also true in cases of virtually shared memory where teams are considered to have direct access to a common, global address space that is physically distributed over a number of nodes. From the notion of teams it is obvious that threads of the same team do have virtually shared memory access, too.

Threads, in the form of lightweight processes, are more than a unit of execution. Interactions between threads are supervised by the PEACE nucleus. For dedicated, massively parallel applications, however, thread supervision can be sacrificed to speed up overall performance. The purest form as a unit of execution is wanted. Therefore, PEACE introduces the notion of *featherweight processes* [Giloi, Schroeder 1991]. This process type represents extremely lightweight processes. Their weight is largely determined by the type of processor actually used, and not by the system software. Featherweight processes are the most basic PEACE process type that can be distinguished from simple coroutines, i.e. they are subject to lightweight thread scheduling.

### 2.2. Events

The potential for more than one thread of control within a team calls for *intra-team synchronization*. Because threads implicitly share the address space of their team, only simple *event propagation* mechanisms are required.

An event is specified by an arbitrary bit pattern fitting into a single processor register. Two event propagation strategies are provided, one-to-one and one-to-many. In the former case, the bit pattern is interpreted as a thread identifier and is classified

accordingly. In the latter case, the bit pattern remains uninterpreted. In both cases, the bit pattern then is used to check for suspended threads that await the specified event.

Events are not queued and do not cross team boundaries. An *event count* is maintained, allowing a thread to unblock only if the specified number of propagation requests related to the same event has occurred.

Propagating an event to a suspended thread that awaits the event implies setting this thread ready to run again, meaning thread scheduling. In addition to the event, the propagation request also specifies the scheduling strategy which is to be applied to each properly blocked thread. That is, the actual thread scheduling strategy may be bypassed. Instead, a preemptive strategy may be applied, i.e. the propagating thread may decide to implicitly relinquish control of the processor if some other thread is synchronized on the occurrence of the indicated event.

### 2.3. Messages

Communication between different threads is one of the primary issues in making massively parallel systems work. In addition, very high performance is predominant, especially limiting the startup time for a single message transfer to an acceptable minimum.

A message startup comprises all system activities at the sending and the receiving site in order to transfer user-level data between peer address spaces. Before the end-to-end data transfer will be possible, the receiving thread is to be located and its existence is to be verified. This at least requires the transmission of a small control message (header) to the receiving site. Before checks are possible, the incoming control message must be buffered and queued. In order to reduce system overhead at the receiving site in this situation, the control message should take the form of a fixed-size packet.

In PEACE, message passing means the exchange of fixed-size packets which, in addition to the header field, carry some user data to the receiving thread. This approach guarantees a most efficient communication. As explained later, the transfer of arbitrarily sized data is accomplished by a separate mechanism.

A *synchronous* request – response model of communication is supported. Asynchronous communication implies increased buffer management and copying overhead. Therefore it is not considered of being a minimal subset of system functions, i.e. a basic mechanism. In addition, buffering and copying increases the message startup time.

The only need for asynchronous communication is in cases where computation and communication shall overlap. In means in effect to have at least two separate threads of control, namely one for communication and another one for computation. For these purposes lightweight processes, i.e. PEACE threads, are the best choice. The benefit of this approach is a general improvement in structuring concurrent programs [Gentleman 1981].

The request – response model implies that a *client* issues the request message to a *server* and implicitly awaits the response message. In terms of PEACE, the issuing of the

request message blocks the client thread[1]. A server thread is blocked in cases where its request message queue is empty. In this situation it will be unblocked by the first incoming request message.

Once having accepted the request message, a *rendezvous* between client and server thread is established. The rendezvous is finished either by replying to the client or by relaying the client to (maybe) another server. Replying to the client results in the delivery of a server-defined response message. The successful reply unblocks the client thread. By default, relaying the client leads to the retransmission of the original request message to the new server thread. The alternative is that this message is overwritten by the server, i.e. the server thread passes a request message on behalf of the client thread. Both, replying and relaying are non-blocking activities for the server. A relay keeps the client thread suspended.

In order to reply or relay the client thread, a *reply capability* is required. In PEACE, the entire server team gains this capability as soon as the rendezvous is established. As was already mentioned, all threads of a team implicitly have the same access right onto PEACE objects. A client thread is such an object. Thus, terminating the rendezvous is not restricted to the server thread, rather it is extended to the server team.

### 2.4. Transfers

The transfer of objects of arbitrary size is not accomplished by the basic message passing primitives. Rather, a separate mechanism is provided, called *high-volume data transfer*. This mechanism is applicable only during a rendezvous, namely when in possession of the reply capability. Under control of the server team, data sets can be either read from or written into the client address space. Because the rendezvous is assumed to either loosely or tightly synchronize the involved teams, high-volume data transfer is a non-blocking activity.

With respect to intermediate buffering, this approach places no demands on the lower level communication system and enables data streams to be always exchanged end-to-end between peer team address spaces. Thus, the need for intermediate buffering may only be caused by limited network interface capabilities, e.g. in cases where it is not possible to directly pass incoming data through to the target team address space.

In the case of virtual memory support, a *virtual transfer* of arbitrarily sized messages is made feasible. Rather than transferring complete data sets between the involved team address spaces (*copy semantic*), the corresponding memory segments can be marked as *copy-on-write*, at the client site, and *copy-on-reference*, at the server site. Thus, only those pages are copied to the server address space which either are written into by the client team or referenced by the server team. Transferring data into the client address space works similarly, i.e., pages are copied that are written into by the server team or referenced by the client team.

[1] Usually, the blocking of a thread does not imply the blocking of the thread's team. As long as at least one more thread is ready to run within the team, processing continues.

### 2.5. Leagues

In order to provide security in the case of a multi-user mode of operation, only related threads are allowed to communicate with each other. Threads which shall belong to the same communication domain are said to belong to the same *league*. Communication within the league is unrestricted, while the crossing of league boundaries is prohibited. In order to provide *communication security*, different users form different leagues.

Leagues may overlap and are related to threads. The former aspect is required to invoke system services by means of message passing, which is the only way in PEACE to request service execution. With leagues being related to threads, it is feasible to have system threads executing within user teams. These threads then perform system-related actions, e.g. checkpointing, in close cooperation with other server processes which, for safety reasons, are mapped into a private league. They perform these actions with permissions granted to the private league, rather than permissions granted to the league of their own team.

Controlling league membership of a thread is not assumed to be supported by hardware. It is the responsibility of the nucleus to ensure thread integrity. Therefore, leagues only apply to threads taking the form of lightweight processes. A league is not associated with featherweight processes.

### 2.6. Gates

The addressing of communication partners is by means of *system-wide unique identifiers*. These identifiers are represented by a numerical value and contain a *logical host field*, providing the information where the effective location of the communication partner is. They are locally managed on each node, thus, they are referred to as locally unique identifiers, *lui*.

A *lui* designates a *gate*, which acts as a port-like communication endpoint [Balzer 1971] without buffering capabilities but with routing capabilities. Normally, gates are associated in one-to-one correspondence with threads, i.e. the creation of a thread also results in the creation of a gate. Consequently, the *lui* also designates a thread.

A thread may have several addresses (gates) at the same time, whereby each thread address is represented by a *lui*. This way dynamic restructuring is supported. Even in the case of team migration, threads still are reachable via their old address. The gate then acts as a *forwarding identifier*, automatically routing messages and transfers to the effective destination. By means of this routing capability, communication monitoring is enabled too.

Remote gates may be cached locally and, hence, the same gate may be distributed over several nodes. In this sense, the *lui* does not necessarily specify the address where the gate is stored, rather it is considered as a hash key to locate locally cached gates. There are two reasons why the potential for gate caching is given. First, in order to keep a distributed system running in case of a node crash, the in-use gates, having been stored at a crashed node, are to be distributed to the nodes where they are used by threads.

Second, communication security also means to prohibit the sending of messages out of a node. This is accomplished by storing a league identifier in each gate, in addition to a thread identifier, and by replicating the corresponding gate to the (sender) nodes.

## 3. Design Principles

By applying basic PEACE abstractions the execution of processes as well as synchronization and system-wide communication between any pair of related processes is made feasible. Based on these abstractions all other system functions are realized. This includes traditional operating system functions like disk and character I/O, device and resource management, networking, fault tolerance, debug support, exception handling, and so on. The following subsections describe by what means the PEACE operating system family is structured, represented and configured.

### 3.1. Structuring by Functional Decomposition

A major challenge in operating system design for massively parallel systems is to elaborate a structure that reduces system bootstrap time, avoids bottlenecks in serving system calls, promotes fault tolerance, is dynamic alterable and application-oriented. Hence, first of all a *functional hierarchy* of system components must be found that corresponds to these needs.

According to the well known principle of *stepwise refinement* [Wirth 1971], functional units are to be identified which are either independent from each other or show a minimum of interaction. Note that this approach does not yet impose any representation restrictions on a functional unit. Rather it aims to define and clarify inter-unit relationships.

The resulting structure forms a hierarchy of functional units according to the *uses relation* [Parnas 1976]. This way minimal subsets of system functions are identified. The primary objective of this minimal subset is to form a problem-oriented abstract machine for massively parallel systems. Using the primitives of this machine, minimal system extensions then are realized in an application-oriented way. As a consequence, each such extension defines a new operating system family member.

### 3.2. Representation by means of Active Objects

Distributed memory architectures call for an object-based system design, which also is a prerequisite for constructing an operating system family. In addition, the *actual structure* [Randell et al. 1978] of system software has important impact on the reliability of the entire computer system. For all these reasons, functional units are represented by active objects, i.e. *processes*.

#### 3.2.1. Service and Manager

In operating system terms, a functional unit implements a system service, such as process management or file handling. The system service is executed by a dedicated system process. Consequently, requesting the execution of a system service requires to

send a message to some process. A typical client – server relation is established.

The functional decomposition usually results in a multi-level hierarchy of service-providing system processes. This means that these processes are in the situation of being both client and server. Because of this duality, they are called *manager*. A team is used to implement a single manager, enabling concurrent service execution within the same manager by means of multiple threads. In PEACE, the entire operating system then consists of a multitude of cooperating manager teams distributed over the nodes.

The consequent usage of teams for system service encapsulation has several benefits. It provides a natural basis for building application-oriented operating systems. System services need only be present if they are required, meaning that the corresponding teams are created and loaded on-demand. Especially in the case of massively parallel systems, it is not required that user teams share the same node with system teams. This significantly reduces global system initialization time and makes the parallel system to appear as a processor bank whose purpose is to exclusively execute user applications.

Following the team structuring approach, the notion of a system call (service invocation) is slightly different from the traditional viewpoint of a trap. A system call must be requested by means of message passing, distinguishing between local and remote operation. In order to hide all these properties from both the service user (client) and the service provider (server), a PEACE system call takes the form of a *remote procedure call* [Nelson 1982]. Based on the remote procedure call approach, only services which are used to build the minimal subset of system functions, the *abstract PEACE machine*, are subject to privileged supervisor mode execution by the underlying processor. All other services are executed in non-privileged user mode. This, for example, distinguishes PEACE from most of the state-of-the-art distributed operating systems such as Mach [Young et al. 1987] and Chorus [Rozier et al. 1988], whose system managers are subject to supervisor mode execution. In this sense, PEACE follows the pattern of object-oriented operating systems [Balter et al. 1986] by means of a process-structured design approach.

### 3.2.2. Administrator and Porter

There are several reasons for service replication in distributed systems. One aspect is to avoid the presence of bottlenecks in cases where a manager is overloaded by too many service requests. Another case is redundancy for fault tolerant purposes. Furthermore, there are replicated and over a number of nodes distributed I/O units such as disks. In all these cases, managers are replicated because of performance, availability, or architectural reasons.

This leads to the concept of distributed managers. The set of managers of the same type constitutes a PEACE *administrator*. For scalability reasons, processes should not be aware of using replicated services (managers). Rather they interact with an administrator. In this situation, the administrator has to keep track of which manager is to be selected for service execution. Explicit cooperation among the managers may be one solution, meaning that all managers are trading resources with each other. However, this implies a loss of scalability at the manager level.

The more flexible and efficient solution to the manager selection problem is to assume an object-oriented view of system service invocation. Provided that the service instance, i.e. object, is properly identified by the request, manager selection becomes a straightforward business. The extreme solution is to have a one-to-one correspondence between service instance and thread. For example, in case of a file manager, each open file object may be managed by a private thread of the file manager team, meaning that the thread's *lui* can be used as a file descriptor. Issuing a file I/O request means that the corresponding message can be directly passed to the proper manager without intermediate routing.

For these reasons, the PEACE administrator concept is not only supported by a number of managers, but also by a related *porter* that directs requests to the proper manager and, thus, serves as an administrator interface. Figure 1 illustrates this approach.

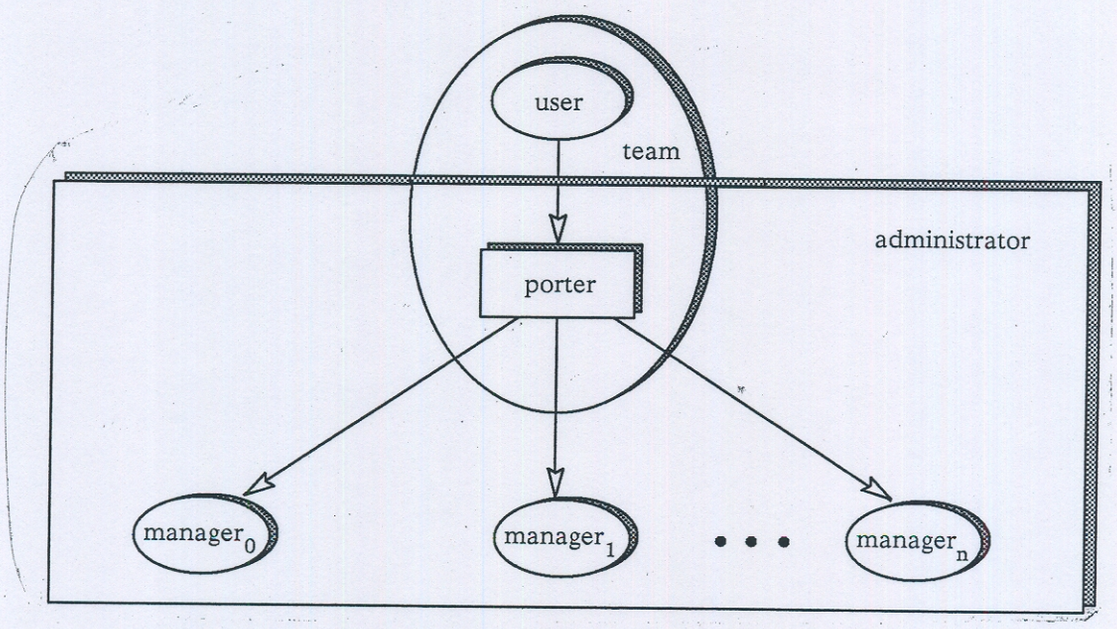


Figure 1: The administrator concept

The porter takes the form of a library and, thus, is part of the team address space of the service-requesting process. Dependent on the type of service, the porter may also encapsule private threads. For example, using porter threads enables service-related exception handling on a message-passing basis. In addition, the administrator typically forms a separate league, meaning that porter threads, on the one hand, belong to the administrator league and, on the other hand, reside within a team that belongs to a different league. These porter threads then access system services with permissions granted to the administrator, rather than permissions granted to their teams.

### 3.3. Configuration by a Third Party

In the scope of massively parallel systems it is important to reduce the amount of system software which is to be executed by each node; otherwise, system bootstrapping becomes a nightmare. For this reason, PEACE distinguishes between site-dependent and site-independent managers. A site-dependent manager typically provides low-level and hardware-related services. For example, the disk manager must reside in a node that has a disk attached and encapsules device dependent functions. The file manager, which uses the disk manager, may reside elsewhere and is considered site-independent.

In many cases, the same service may have several representations. A memory management service, e.g., can be realized with or without dedicated hardware support such as a memory management unit. This does not necessarily require to provide a different service interface. Rather, the service can be viewed as an *abstract data type* [Liskov, Zilles 1974] that has several implementations which all inherit the same external interface. Dependent on the user requirements and on the availability of dedicated hardware support, the proper memory manager can be used. The configuration decision then will be made with respect to either performance, protection, or hardware availability.

The property of being configurable is absolutely necessary to meet the needs for massively parallel systems. Except in the case of site-dependent managers, a *third party* is able to establish configurations based on the individual needs of distributed applications. That way the parallel machine can be considered as a *processor pool* that is exclusively used for the execution of user application processes. Figure 2 illustrates this approach.

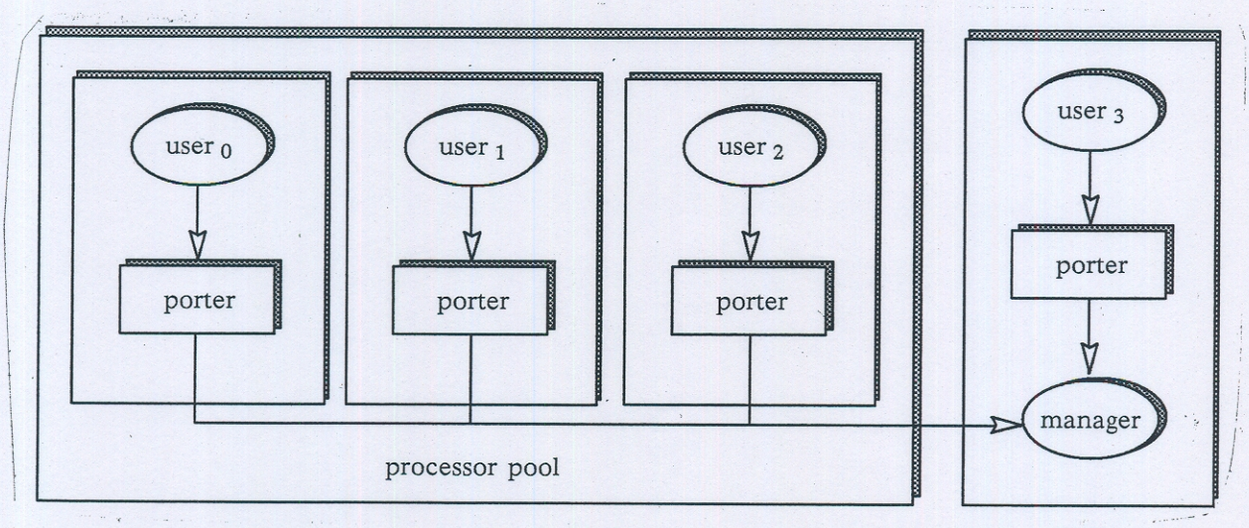


Figure 2: Functional dedicated system

Nevertheless, managers and user processes might be forced to share the same node if a temporary deficiency of node resources exists. As soon as nodes become available again, user processes are preempted to continue execution on their own nodes. The

buzzword for this functionality is *team migration* in PEACE.

Another example that calls for configuration by means of a third party is given in the case of the MANNA node [Giloi 1991]. The idea of this node architecture is to use both processors, which share the same node, as two dedicated functional units: an AP, the **a**pplication **p**rocessor, for user process execution and a CP, the **c**ommunication **p**rocessor, for system-wide message passing and high-volume data transfers. In fact, distinguishing between AP and CP, i.e. assigning a communication manager to the CP, then becomes a matter of configuration, rather than implementation.

### 3.4. A Case Study – Process Creation

A short example from the area of resource management shall help to clarify how the PEACE administrator concept works. Surely, the creation of processes may be considered, one of the most significant services. For this purpose, the PEACE *process administrator* is investigated.

The creation of a new process in a distributed environment is characterized by the questions where the process is to be created, what image shall be loaded and how will it be loaded. For reasons of simplicity, a *fork*-like system call is discussed.

The *fork* semantic requires to replicate the address space of the creating process, the *parent*, which then builds the address space for the created process, the *child*. Thus, the image to be loaded is a copy of the actual image of the parent team. For efficiency reasons, loading the image in a distributed environment should be done directly from parent to child, especially if one keeps in mind that a *process manager* is not required to reside with the parent or child in the same node.

This leads to three fundamental functions, namely image transportation, routing of service requests, and management of system data structures. For safety reasons, only the system data structure management must be performed by the process manager. The other two tasks are performed by the *process porter* on behalf of the parent process. The complete *fork* operation then comprises the following activities:

1. The porter requests from its process manager a team map, giving information about the threads and memory segments allocated by its team. It then requests from a *node manager* the allocation of a new node.
2. The porter forwards (i.e. routes) a process creation request that carries the team map to the process manager related to the selected node.
3. According to the team map, the selected process manager creates a new team, associating it with an address space cover. In addition, the team of the porter is given reply access right onto the newly created team.
4. The porter transfers its team image to the new team on the basis of the high-volume data transfer. Note, the porter team was given proper access right for this purpose.
5. The porter terminates the rendezvous to the new team, thus starting the child process. Again, note that the porter was given a reply access right capability.

This approach leads to a decentralized control of the process creation activities. Usually, there is a number of process porter modules, each one encapsuled by a team whose threads potentially request process creation. In addition, the process manager is responsible for process management in a number of nodes, for example in cases where all the nodes are jointly used as a processor bank. Finally, the need for process manager replication is obvious if one is faced with the management of thousands of nodes.

## 4. System Organization

In addition to the application, PEACE distinguishes three major system building blocks. These building blocks realize system-wide inter-process communication, hardware abstractions and application-oriented system services. They are explained in the following subsections.

### 4.1. Nucleus

The nucleus is the most basic PEACE system component accessible by processes and must reside in each node. The nucleus provides *system-wide inter-process communication* and, thus, implements the basic PEACE abstractions. That is, the nucleus implements objects related to threads, teams and leagues, events, messages and transfers, and gates. However, these objects are not dynamically allocated by the nucleus. Whatever system component is responsible for their creation – this, by the way, is exactly the domain of the PEACE kernel –, the nucleus simply assumes that they are present. Indeed, resource management is not its job. Rather, the nucleus performs nothing else but the queuing of existing objects, i.e. interprets some more or less dynamic data structures that take the form of single-linked or double-linked lists.

#### 4.1.1. Basic Organization

Two major goals stand behind the nucleus development, portability and performance. The first goal is achieved by means of a problem-oriented structure, as illustrated in Figure 3. The second goal is achieved by focusing on the substantial facts, namely that the absolutely minimal subset of system functions support process execution and communication only.

The top layer of the nucleus is dedicated to *NICE*, the **n**etwork **i**ndependent **c**ommunication **e**xecutive. It serves as the nucleus interface, providing primitives for system-wide message-passing and high-volume data transfer. There is no other way in PEACE to communicate excepted by means of NICE. All other system services, even higher-level communication functions, are implemented by using NICE primitives. Besides communication, low-level process scheduling is supported as well as intra-team synchronization by means of events.

The next lower layer is dedicated to *COSY*, the **co**mmunication **sy**stem. COSY provides inter-node communication and is used by NICE to implement system-wide inter-process communication. Data transfer primitives provided by COSY work asynchronous, without copying messages but queuing data transfer requests. Threads actually issuing a remote communication request are not blocked within COSY, unless a busy waiting

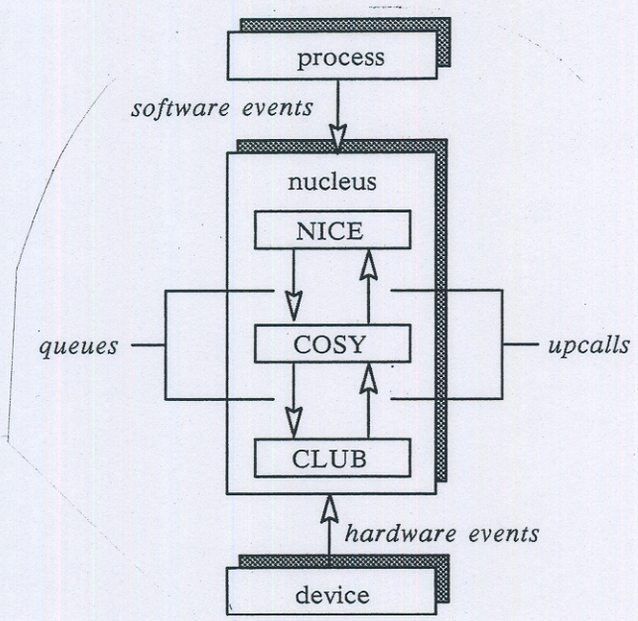


Figure 3: Nucleus organization

scheme of data transmission is realized.

The basic idea of COSY is to make NICE really independent from network interface capabilities, thus making the nucleus generally portable. Nevertheless, the COSY interface is tuned to meet the requirements for massively parallel systems, especially with respect to performance. For example, this means to implement a high-volume data transfer facility that is optimally balanced to the underlying network properties. Either a blast protocol can be used in cases of synchronous networks or segmenting can be performed to reduce the potential of blocking.

Segmenting will always be applied by COSY in cases where the network is not capable to transfer arbitrary sized data streams, as for example Ethernet [Metcalfe, Boggs 1976]. In addition, *virtual channels* are realized in this case, meaning also to provide multiplex and demultiplex capabilities. In order to control multiplexing/demultiplexing, segments belonging to different transfers are tagged with different *luis*.

The bottom layer is dedicated to *CLUB*, the **clu**ster **b**us[2)]. It encapsules the network interface driver routines and, basically, is not concerned with any kind of network protocols except for node addressing. Above CLUB, nodes are logically addressed. The logical node address is represented by the host field which is contained in each *lui*. As was outlined previously, a *lui* is used to address communicating threads, indirectly via gates. This addressing scheme is applied by NICE. In case of remote communication, the *lui* is directly passed-through to CLUB which, in turn, maps the logical host address onto a physical node address.

[2)] A heritage from SUPRENUM, where nodes inside a cluster are interconnected via a bus. In general PEACE terms, CLUB stands for a club of network hardware interface drivers.

In downward direction, request queues are used to interface adjacent layers. In upward direction, *upcalls* [Clark 1985] are applied to enable the propagation of network events to the COSY and NICE layers. Usually, these events are related to incoming messages which, finally, are to be processed by NICE.

### 4.1.2. Nucleus Family

In order to meet the various communication performance requirements of parallel applications, a nucleus family is identified in PEACE. The purpose of this family is to offer different modes of operation for a single node, e.g. to provide optimal support for either single-tasking or multi-tasking. However, the distinction between single-tasking and multi-tasking support is too rigid [Schroeder 1991]. The PEACE development for SUPRENUM demonstrates that a more discriminating approach must be taken. One reason is that a large class of applications must be optimally supported. A second reason is that it is much less risky to develop an operating system incrementally rather than trying to implement the entire system in one shot.

In this sense, a multi-tasking nucleus will begin with supporting static scheduling, to be extended by preemptive scheduling and address space isolation, by memory protection and, finally, by multi-user security. The key aspect is that the nucleus design is required to be complete before implementation starts. All the intermediate steps towards the final implementation are considered as representing an autonomous member of the nucleus family in its own right, exhibiting different functionality and performance characteristics. Figure 4 illustrates this approach by means of a nucleus family tree. As will be discussed, nucleus functionality increases from left to right and from top to bottom. By the same amount by which the functionality is increased, the communication performance drops, i.e. the message startup time increases.

At the root of the family tree, there are two major user requirements, namely single-tasking and multi-tasking. In case of single-tasking, a distinction is made into single-threading and multi-threading. Note, a task is a PEACE team and hence may be multi-threaded. In case that only a single thread of control must be supported on a node, the nucleus purely implements *communication*. This is the situation where a one-to-one mapping between a node and a user task is feasible.

In PEACE, synchronous inter-process communication is considered as the fundamental communication paradigm, offering the highest possible end-to-end communication performance. *Threading* introduces concurrency relative to a common address space, Multiple threads of control constitute a team, specifying a one-level scheduling domain. This nucleus representation implements *featherweight processes*, i.e. extremely lightweight processes. No mechanisms for security are available at this stage in the family tree. Thus, featherweight processes can be compared to coroutines which are subject to lightweight thread scheduling.

Multi-tasking increases only the scalability and does not imply protected address spaces. Concurrency relative to several teams belonging to the same user application is supported. Thus, a non-preemptive *scheduling* strategy is introduced into the nucleus. This constitutes a switch from one-level to two-level scheduling, i.e. it extends thread

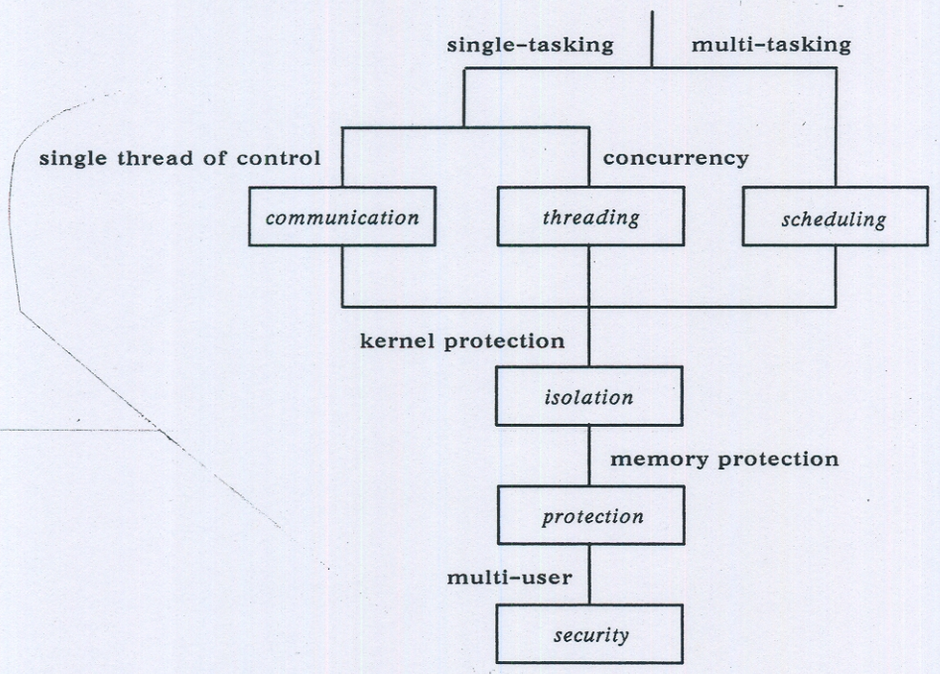


Figure 4: Nucleus family tree

scheduling by team scheduling.

In all these cases, the nucleus takes the form of a non-protected shared library that is directly linked to the application task. In addition, a node controlled by these nucleus instances is used exclusively for user task execution. Except for the nucleus, no other PEACE system components are located on these nodes. Application tasks (teams) are loaded as part of the node bootstrap procedure. Following the pattern of coroutines, merely dynamic creation/destruction of threads is supported. Since system services are invoked by means of remote procedure calls, PEACE server teams may reside elsewhere and are not required to share a single node with the user teams (i.e. tasks).

The requirement for preemptive scheduling implies a complete nucleus *isolation*. Now, interrupts must be handled, and the teams are given a limited time quantum for program execution. Dynamic creation/destruction of teams is introduced. However, team address spaces are not yet isolated, i.e., protected. In addition, the nucleus must be invoked through traps, thus introducing more overhead. Consequently, the nucleus as part of the kernel is executed in privileged supervisor mode and protected against user mode tasks. Nucleus (kernel) protection may also be a case for single-threading and multi-threading instances.

Applying address space isolation to user teams completes *protection* in its traditional sense. The kernel is extended by services to associate process objects with address spaces. Except for having explicitly requested segment sharing, teams are prohibited to directly access and manipulate peer address spaces. This makes the nucleus a true multi-tasking system, able to support the execution of multiple teams belonging to

different applications. In this nucleus instance, high-volume data transfer, for example, implies explicit segment validation before accessing a team address space.

If a multi-user mode of execution is to be supported, memory protection alone is not sufficient for massively parallel systems. Rather, communication firewalls in terms of leagues must be established such that threads belonging to different user teams are prohibited to communicate with each other. Thus, the principle of memory protection is extended into a distributed environment. Precautions for communication *security* are necessary. The nucleus is required to enforce integrity of the different distributed user applications running concurrently on the parallel machine.

### 4.1.3. Problem Orientation

With respect to different user requirements, different NICE modules are used for optimal support. Each NICE instance implements a member of the nucleus family. Different network characteristics are covered by dedicated COSY modules, improving the portability of NICE by means of isolation from network details. Network hardware transparency is achieved by CLUB, i.e., there are different CLUB implementations for different network hardware interfaces. This generally improves portability by means of isolating nucleus instances from the details of network devices.

The order in which we discussed the members of the nucleus family also defines the order of increasing system functionality. User-level communication performance decreases in the same order. Applications which scale well with the given number of nodes are supported by either of the first two nucleus instances (communication and threading), depending on whether synchronous or asynchronous inter-process communication is required. A first step towards multi-tasking is to introduce the scheduling nucleus. This nucleus represents a compromise between scalability and protection. It is used for dedicated and mature applications.

A similar compromise can be made concerning nucleus isolation. However, there are two additional major issues to be addressed by the nucleus. First, dynamic nucleus reconfiguration must be feasible. Second, multi-tasking must be supported regardless of the availability of a MMU. This generally enlarges the applicability of the system.

Dynamic nucleus reconfiguration is needed in cases where system availability is the dominating issue. A single-tasking nucleus is used in all cases where tasks are mapped one-to-one to nodes. Node crashes, however, may require task redistribution and, thus, could cause the switching to a multi-user multi-tasking nucleus at some other node. This also may imply that some nodes are now to be shared between a number of tasks belonging to different user applications. In this case, the single-tasking nucleus need be replaced by the multi-tasking nucleus at runtime.

### 4.1.4. Multiprocessor Support

Some members of the nucleus family are also suitable for *shared-memory* multiprocessor execution. In the traditional approach of multiple processors that have equal rights with respect to shared-memory access, the nucleus is shared accordingly. In

an approach where *functionally dedicated* processors are used, as for example in the case of MANNA, the nucleus is distributed over the processors. Figure 5 gives a rough idea about a functionally dedicated nucleus representation for a single MANNA node.

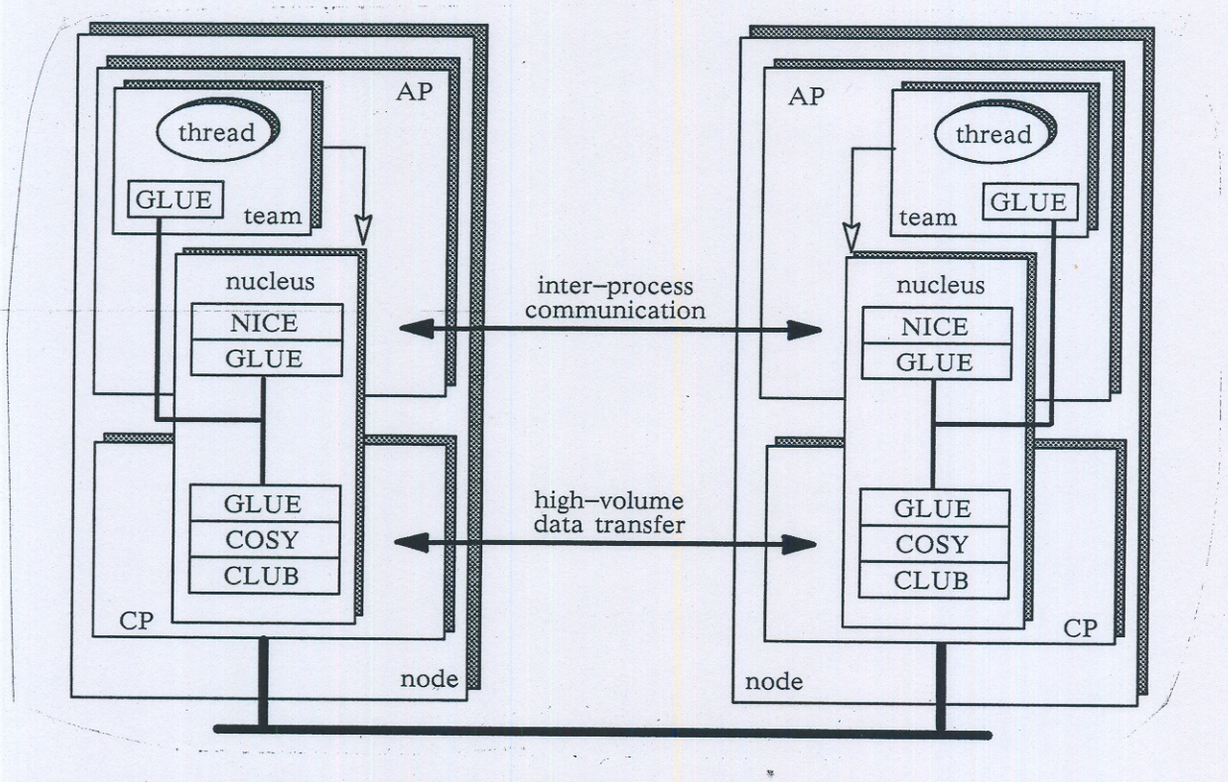


Figure 5: Functional dedicated nucleus organization

As illustrated, COSY and CLUB are mapped onto the communication processor (CP), whereas NICE remains on the processor that performs the user program execution, the application processor (AP). Hence, the major part related to communication is off-loaded.

This approach however requires an additional module, GLUE, to interface to the CP. Both, NICE and user teams must use GLUE in order to request data transmission by the CP. Thus, processes are able to interact directly with the CP. They use NICE only in cases where synchronization is required. Because the COSY interface is non-buffered and asynchronous, *no-wait send* semantics of communication [Liskov 1979] can be easily provided at team library level.

### 4.2. Kernel

Basically, the kernel serves as a *hardware abstraction*. It offers services to dynamically create and destroy process objects and to associate these objects with address spaces. In addition, it encapsules device drivers and enables higher-level system processes to be attached to the trap/interrupt vector of the underlying processor. This way, the kernel converts traps and interrupts into messages and forwards these messages

to some higher-level system process.

Each of these services is provided by a dedicated *kernel thread*, meaning that the kernel is multi-threaded and invoked by remote procedure calls. There is one distinguished thread which is called the *ghost*. This thread always is bound to a well-defined *lui* that only consists of a non-zero host field. Thus, in terms of PEACE, the *ghost* is the logical representative of a node.

Besides some basic services to support the PEACE naming system, the *ghost* initially creates threads and teams having been booted onto the node. In Figure 6 the kernel organization in terms of threads is exemplified.

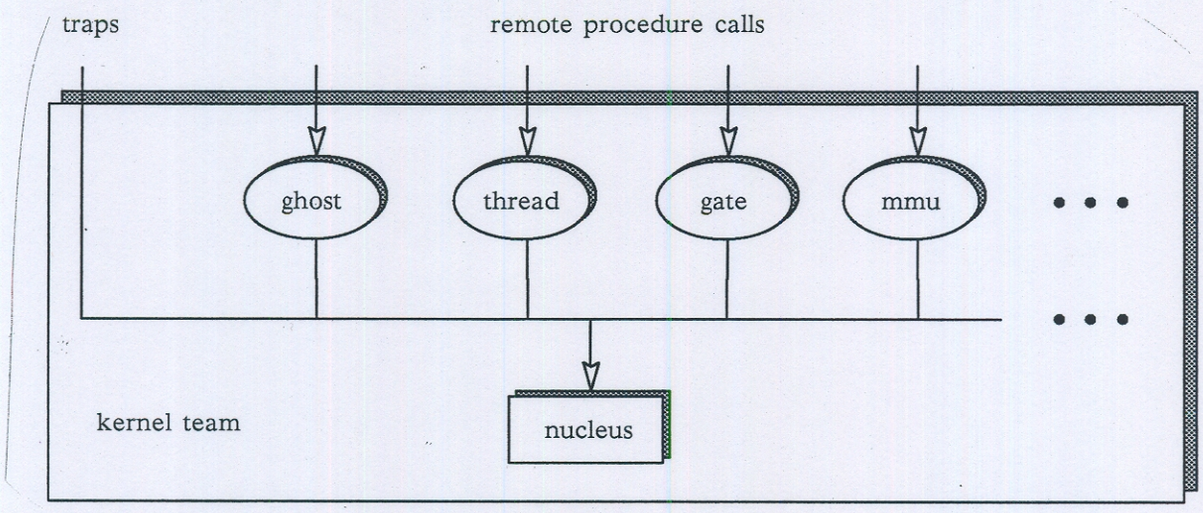


Figure 6: Kernel organization

Except for the *ghost*, the presence of a kernel service depends on both the application and the hardware. Dynamic process creation and destruction must be supported if the application requires to have multiple teams running on the same node, whereas a disk driver thread obviously is not required on diskless nodes. An extreme situation arises if only the single-tasking mode of operation is supported. In this case the *ghost* effectively represents the user task, meaning that no other kernel service is provided on that node.

Because of its basic functions it is obvious that, for some configurations, the kernel is to be executed in privileged supervisor mode of the processor. Dependent on the given hardware structure, this will be the case if devices are to be managed, e.g. a memory management unit or a disk. Note, it is not PEACE that demands that device drivers must be executed in supervisor mode. The kernel, i.e., each kernel thread, depends on the nucleus and, following the uses relation, is thus layered above the nucleus. Consequently, if the kernel is subject to supervisor mode execution, the nucleus must execute in supervisor mode too.

### 4.3. System

The third PEACE building block is the distributed operating system whose major purpose is to provide *application-oriented system services*. Without exception, the services provided by this building block always are assumed to be executable in non-privileged user mode of the underlying processor. In addition, all these services are considered site-independent. Both properties let the operating system appear as an arbitrarily configurable building block. Figure 7 illustrates the complete PEACE system organization, including a building block of distributed applications.

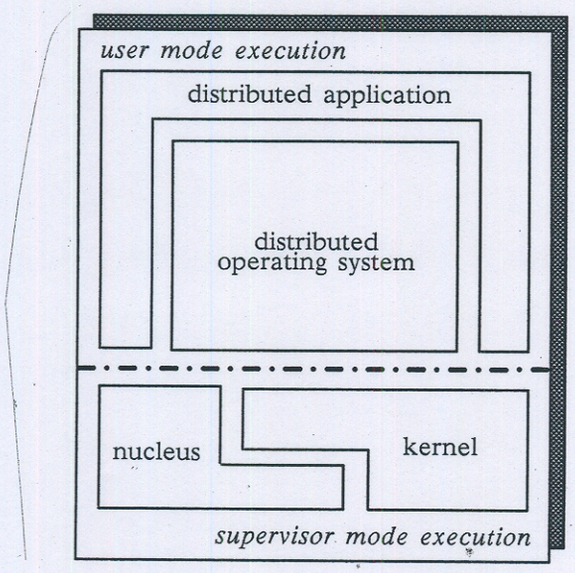


Figure 7: System organization

The PEACE operating system is constituted by a multitude of active objects, i.e., managers that are combined into administrators, which then are distributed accordingly. The following basic administrators are defined:

*name* Provides services to locate objects in a distributed environment.

*entity* Makes PEACE a dynamically alterable system.

*process* Performs application oriented process management, i.e. creates/destroys threads and teams.

*memory* Allocates and deallocates memory segments and supports virtual memory as well as virtually shared memory.

*signal* Implements exception handling by means of message passing and threads residing within user teams.

*fault* Catches hardware exceptions that are not properly handled.

*clock* Introduces some means of time.

*file* Manages files that are distributed over a couple of disks, whereby the disks usually are attached to dedicated nodes.

*uio* Provides a **u**niform **i**nput and **o**utput interface.

*load* Performs the loading of teams.

There are further administrators dealing with *synchronization*, *migration*, *checkpointing*, *recovery*, *debugging* and *networking*.

As is discussed later, the operating system is incrementally loaded, namely when a process requests a system service the first time. This feature leads to an application oriented system structure. The application determines at runtime what system services are assumed to be present.

## 5. Service Invocation

A prerequisite for service invocation is *location transparency* and *access transparency*. Location transparency is achieved by means of naming, whereas access transparency is supported by remote procedure calls. Naming, as well as configuration, addresses a scheme which is often referred to as "programming in the large" and the remote procedure call paradigm addresses "programming in the small". Combined, this provides a powerful approach to the development of distributed/parallel systems.

### 5.1. Remote Procedure Calls

A major requirement from the efficiency point of view is to provide fast invocation protocols, as for example described in [Birrell, Nelson 1984]. This is best accomplished by a *protocol family* that meets the individual needs of different service classes. A flexible service binding scheme is required, to transparently support dynamic alterable system structures. Another important aspect is that of an interface definition language, enabling the system designer to specify service interfaces. By means of stub generators, e.g. [Gibbons 1987] and [van Rossum 1990], system call libraries (i.e. stubs) are automatically produced.

#### 5.1.1. Protocol Family

Dependent on the type of service, different invocation semantics [Nelson 1982] can be identified. The two basic semantics being employed in PEACE are:

*at-least-once* Request and response messages are not checked for duplicates. This strategy is used to invoke *idempotent* services and happens to be the most efficient one.

*exactly-once* Request and response messages are checked for duplicates. This implies long-term buffering of request message header information and of complete response messages to execute the *duplicate suppression* protocol.

The minimal subset of the protocol family provides at-least-once invocation semantics. Based on this subset, exactly-once semantics is considered the next minimal extension,

thus introducing another family member. For reasons of simplicity and efficiency, *at-most-once* semantics as proposed by [Liskov et al. 1983] is not included in PEACE.

In addition to the invocation semantics covered by different members of the protocol family, *concurrent invocation* is supported by applying the team concept. Two approaches are distinguished in PEACE. On the one hand, each service request will be processed by a separate manager thread called *clone*. This works independently of the kind of service, i.e. system call, and is based on a *clone pool* that is maintained by the stub of the manager team. On the other hand, for different system call classes different manager threads are used. Thus, the team concept is applied in conjunction with the abstract data type concept. In both cases PEACE managers are assumed to be multi-threaded. Both forms of *threading* significantly improve overall system performance because of team-internal concurrency.

### 5.1.2. Addressing

As was explained with the administrator concept, one task of the porter is to select a manager that is responsible for executing a given service function. Service invocation is by means of remote procedure calls, and service addressing (manager selection) is capable by either of:

*function name* The name of the service function, i.e. system call, is used to address the corresponding manager. The function name is required to be unique within a global name space.

*entity name* The name of the module and/or manager that exports a given service function is used for addressing. The entity name is required to be unique within a global name space.

*name domain* A separate name space is used for manager addressing. This name space partition, i.e. name domain, is related to the service requesting process. Either function name or entity name is required to be unique within a name domain.

*manager gate* The system-wide unique manager identifier is used to address the manager. That is to say, a *lui* serves as the manager address.

Addressing by a function name is the most transparent approach, because it reflects the same addressing scheme as a local procedure call. A similarly transparent scheme is that of using entity names, whereby in this case a single name stands for a number of service functions. In effect, both addressing styles are meaningful if a simple load splitting distribution scheme is used. However, this does not hold for a distributed system with replicated managers. In the case of identical, replicated managers the potential for *name clashes* is given, because each manager provides the same set of service functions and, therefore, will be addressable by the same set of either function or entity names.

The name domain addressing scheme prevents name clashes, assuming that the global name space is structured into several name domains. Each system call then is to be supplied with a domain identifier in order to resolve the manager address issue relative to a given domain. For example, in case of the PEACE process management service the

domain identifier is derived from a *lui* argument that specifies a thread. Within the domain of this thread the corresponding process manager can then be located.

With the manager gate addressing scheme, service addressing is made feasible without any additional mapping overhead. As was outlined previously, a gate is designated by a *lui*, which is also a thread identifier. Hence, the manager gate directly refers to the manager thread responsible for service execution. As in the case of the name domain addressing scheme, a system call function is to be supplied with the gate identifier of a manager thread. This addressing scheme is used in PEACE to select name managers, e.g., during the phase a service is located.

Addressing by means of manager gates introduces some kind of object orientation. In this case the function invoked deals with an explicitly defined object that helps to address the proper manager. Both, object and operations on that object are encapsuled by a single manager thread.

In fact, this scenario is not efficient with respect to process and memory resources if small and simple objects are only to be managed. But it is efficient when such an object is, e.g., a file, and several objects are managed by a single file manager team.

Common to all addressing schemes is the same stub functionality in terms of marshaling and unmarshaling of system call arguments. In addition, the actual invocation protocol is independent from the addressing schemes discussed.

### 5.1.3. Interface Definition Language

In order to hide most of the details discussed so far, an interface definition language and compiler, i.e. stub generator, is required. Given an interface definition, the stub generator automatically produces *stub modules*, for client and server site, that encapsule all the mystics of service invocation in a distributed environment.

In addition to these basic functions, the PEACE interface definition language, PRECISE [Nolte 1990], reflects the addressing schemes discussed above. Especially object orientation is supported in a flexible way, because this approach is a convenient solution for a great variety of problems in the area of distributed and/or parallel computing.

In PRECISE, parameter passing semantics comprise *call-by-value*, *call-by-result* and *call-by-value-result*. In the absence of virtually shared memory, *call-by-reference* semantic is emulated by call-by-value-result and applied to simple data structures such as strings, arrays, records, etc.

## 5.2. Service Localization

The previous subsection discussed the way service invocation works in PEACE. Service localization, however, was not investigated. This function is related to the *naming* and *addressing* of distributed entities [Shoch 1978]. In the following, a short overview of the PEACE naming approach [Sander et al. 1989] is given.

### 5.2.1. On Transparency in PEACE

As was pointed out, a PEACE process is associated with at least one system-wide unique identifier, a *lui*. The *lui* is represented by an integer and is considered to be the *effective address* of a thread. At a higher level a process is associated with at least one *relative address*, represented by a symbolic name. The PEACE naming system is then used to map relative addresses onto effective addresses. Figure 8 illustrates the PEACE naming scheme.

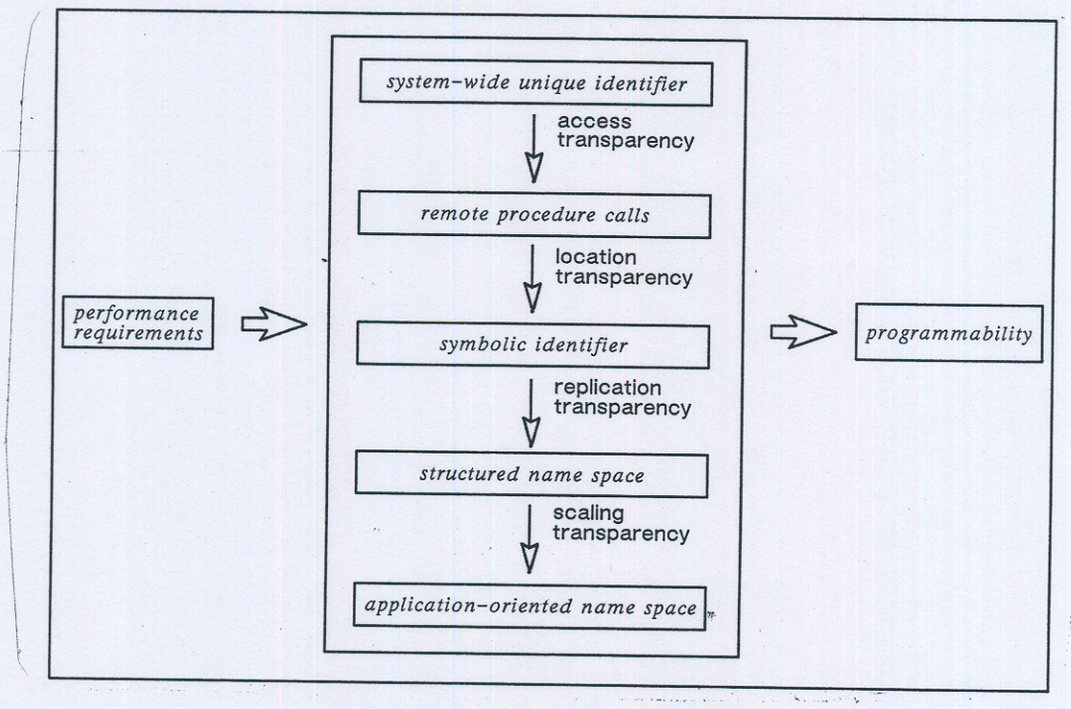


Figure 8: The PEACE naming scheme

System services are invoked by a remote procedure call that introduces *access transparency*, i.e., hides the difference between local and remote operation. A procedure name stands for a particular service function which, in turn, is implemented by some PEACE manager. Usually, a complete service interface encompasses a set of service functions, thus representing a set of relative addresses for the same manager. Dependent on the type of service, relative addresses are also related to argument (object) names. A typical example is the file service, where each file name (file object) is given a relative address in PEACE.

A *symbolic identifier* is used to designate a relative address. All symbolic identifiers together constitute a global name space. They will be used to generally identify distributed objects independent of the nature of the object. That is, symbolic names are used in PEACE to identify devices, nodes, processes, functions, data, and so on. This introduces *location transparency*. The mapping function defined between effective process address and relative process address is dynamic. It makes dynamic reconfiguration feasible and leads to a flexible system organization.

In case of replicated services the same service name is to be used to address different manager processes. Within a flat name space, this service name potentially is ambiguous. Hence, in the absence of object-oriented system interfaces, *replication transparency* must be provided by a *structured name space*. Names of replicated services constitute a specific name space partition. In general, such a partition guarantees the uniqueness of names relative to a specific context and is called a *name plane*. The sharing of name planes then gives different processes access to the same set of system services.

Scalability is one of the most important characteristics of distributed systems, related both to hardware and software architecture. From the application program viewpoint, *scaling transparency* is desirable such that program execution works independently of the actual underlying hardware and software organization. Above all, the operating system family concept as applied in PEACE requires a scalable operating system architecture. The idea is that dedicated PEACE system processes provide application-oriented operating system services and that these processes are loaded at the time the distributed application is installed. This leads to an *application-oriented name space*, used to isolate distributed applications and to model the set of system services available for the given application. It also requires a hierarchically structured organization of name planes, building a unique *name domain* for a specific set of processes.

### 5.2.2. Symbolic Naming

In order to keep the kernel complexity small and achieve high-speed inter-process communication, location transparency is not exclusively supported by the kernel, but rather in cooperation with symbolic naming functions provided by dedicated system processes, so-called *name managers*. The entire name space consists of a multitude of name managers, together constituting the *name administrator*. Access to the name space is by means of dedicated *name porter* libraries.

As with any other PEACE system service, the naming service is invoked by a remote procedure call. In PEACE, each team, and, thus, each thread of the team is bound to a *domain identifier* which is a system-wide unique name manager identifier, i.e. the *lui* of a name manager. This approach either enables the sharing of a name domain, in case of identical domain identifiers for different teams, or leads to a complete isolation of name domains, in case of different domain identifiers for different teams.

The domain identifier of a given team is determined by executing a *ghost* request in form of a remote procedure call. Obviously, the naming service cannot be applied to locate this kernel service, rather a fixed, effective process address is needed. In PEACE, the *ghost* is always the first active process on a node and, therefore, is referred to by a system-wide unique identifier which only consists of a non-zero host field, providing the information in which node the *ghost* resides. Usually, at the name porter level, this identifier will be derived from the *lui* of the requesting thread.

Upon request, name mapping will be performed by a name manager. The manager does not know the semantics of a symbolic name. Each symbolic name is associated with an user-definable and variable size data set. Resolving a name means the delivery of the data set. The content of the data set is transparent to the name manager. Its

interpretation is completely up to the user. At the remote procedure call level, for example, the data set is assumed to contain the following entries:

*server* The *lui* of the manager responsible for service execution.

*object* A manager-relative object and/or function identifier.

Usually, at manager startup time, the manager stub generates a data set for each exported system function. It then requests from the name manager the creation of a name and the association of the specified data set with that name. Upon name resolution, the stub at the client site retrieves the data set. This way, various strategies of access transparency can be supported by the remote procedure call system. A manager can be addressed either by a global entity name or by a service/object name.

Creation and destruction of names is dynamic. Typically, at initialization time, a manager exports its services by creating corresponding names. Name mapping is done at runtime, too. For example, the first time a system service is invoked, the corresponding stub routine (on behalf of the requesting thread) requests name resolution from the name porter that caches the data set associated with the name. Cache updates then are performed by the team itself, as part of handling naming exceptions. For this purpose, the name porter is supported by a private thread, which acts as a per-team exception handler at the naming library level and requests name resolution.

### 5.2.3. Name Space Organization

The PEACE name space is implemented by a multitude of name managers, each one controlling a single name plane. A number of name planes constitute a unique name domain for the distributed application. The structure of name domains is influenced by several aspects, namely:

- the organization of a distributed application;
- the mapping of the application onto the underlying hardware system;
- the isolation of name spaces for different applications;
- the operating system services exclusively related to the application;
- the global operating system services.

It was one of the major requirements in the design and development of the PEACE naming system to cover all these aspects with a single mechanism. Basically, the name administrator defines a *structured name space*, whereby name managers, i.e. name planes, are arranged in a tree-like fashion. The interconnection between different name planes is by means of names. Upon name resolution, the name porter then observes the name space by traversing name plane links. Thus, the way the name space is observed is defined by the linkage structure of the various name planes.

The construction of a name space itself is a dynamic activity and depends mainly on the system services encapsulated by manager processes. Name managers are created on demand, namely when a new service (e.g., file manager) that assumes the presence of a specific name plane is integrated into the system. This also implies that name space editing must be performed. A name manager is placed into the name space by updating

the name plane linkage structure accordingly.

## 6. Getting Started

Getting the complete operating system over hundreds or thousands of nodes instantaneously distributed is not only utopia, but will also significantly slow down the entire system startup time. It will neither be desirable from the system point of view nor will it be necessary from the application point of view. What is to be guaranteed is that system services are available at the time they are used by some process. That is, loading system services at least should be postponed until processes that depend on these services are loaded.

Nevertheless a bootstrap facility is indispensable. However, the only purpose of this facility is to load those system services which implement an incremental load procedure. This way a minimal subset of system functions is initially distributed over a massively parallel system. As will be discussed, it turns out that the majority of nodes are not bootstraped anyway – incremental loading encompasses incremental bootstrapping.

### 6.1. Instantaneous Loading

In PEACE, instantaneous loading is closely related to the bootstrapping of nodes with a number of teams. In its simplest form node bootstrapping means the transfer of program images from disk into main memory of the node – in terms of PEACE, a program image is represented by a *team image*. However, in a massively parallel system the majority of nodes is diskless. These nodes must be bootstraped via the network. Therefore they are required to provide low-level boot interfaces that are attached to the network, rather than to a disk.

After having been reset or switched on, a node initializes its boot interface and subsequently is ready to accept one or more team images. In the following these nodes, each one called *slave*, play the *passive role* in the bootstrapping procedure. The *active role* is dedicated to nodes having direct disk access. Each node playing the active role is termed *master*.

The master reads team images from disk and transfers these images over the network to a slave. Team images in addition with some control information together constitute a *boot image*. A boot image is interpreted by a slave to determine the number of teams being bootstrapped on its node.

As long as no boot image is received from the master, slaves remain passive, i.e. inactive with respect to process execution. This approach avoids that the network is flooded by bootstrap requests issued by the slaves. Dependent on the master, it also enables that only a minimal subset of nodes are bootstrapped. Furthermore, the slaves are not required to know any master, meaning that becoming a master is a matter of configuration.

The first team being bootstrapped on a node is the kernel, which at least consists of the nucleus. Once having finished the bootstrap procedure, a slave bootstrap module passes control over to the kernel, i.e. the *ghost*, which in turn creates a *root thread object*

for each loaded team. Additional threads then are created by the teams itself. Note, this way a multi-tasking mode of operation can be supported although it might be the case that scaled down kernels do not offer any process creation services. The tasks (i.e. teams) simply are created at bootstrapping time.

### 6.2. Incremental Loading

The basic idea in PEACE is to perform *on-demand loading* of system services. That is, system services are only loaded at the time when they are really needed. This is comparable to the *trap-on-use* property of Multics [Organick 1972]. Obviously, there is the need for some low level services, i.e., "trap handlers", such that incremental loading of additional system services is possible [Schmidt 1991]. The following subsections describe these low-level services and their inter-relationships.

#### 6.2.1. Entity Faults

On-demand loading of services at runtime can be accomplished either explicitly, by using dedicated system calls, or implicitly, during service invocation if the corresponding manager does not yet exist. The latter approach requires close cooperation with the remote procedure call level. If service addressing fails, a *server fault* is raised, similar to a page fault in virtual memory systems. Handling a server fault results in the loading of the requested service, i.e. the proper manager team is created and given a program for execution.

Any kind of service that can be loaded on demand is in no way distinguished from an application process. Thus, incremental loading works for both user and system applications. The general term *entity* is used for teams that belong to either of these application classes. In this sense, the server fault actually means an *entity fault*.

Entity faults are propagated to a system process called *entity manager*. Basically, this means that, once having determined that the entity is not yet available, a stub routine requests entity loading by instructing the *entity administrator* accordingly. The stub passes the load request to the *entity porter* which then takes charge of all activities related to the loading of the specified entity. Note, the entity porter takes the form of a system library and belongs to the team of the thread that caused the entity fault. As long as fault handling is in progress, the thread is blocked on the entity manager, waiting for loading to be completed.

The entity manager maps *entity names* onto file names, i.e. associates with entities a file that describes the team image to be loaded. With each entity name several attributes are stored. For example, the file may describe either a plain team image or a complete boot image and, hence, is to be classified accordingly. In case of site-dependent managers, the node addresses are stored with the entity name. In addition, a name scope may be explicitly associated with the new entity. A distinction between the single-tasking or multi-tasking mode of operation for the entity is also made.

### 6.2.2. Name Manager Faults

For being addressable, managers are required to export symbolic names into a structured name space. These names are imported from the name space when they are used for the first time. For example, the remote procedure call level executes this procedures in order to achieve dynamic manager binding. In PEACE, a structured name space is formed by a multitude of name managers, interconnected by well-defined symbolic names.

In case of a tree-structured representation, different tree nodes represent different *system scopes* and are, likewise, implemented by a name manager. Hence, a manager that is related to a specific scope is forced to export its names into that name manager that implements this scope. It is not guaranteed that the scope is already defined, i.e. that the corresponding name manager is present. Note, a scope is only required if related processes are already present. Hence, the potential for a *name manager fault* during the name export sequence executed by a manager is given.

Because of the nature of naming in PEACE, a name manager cannot be loaded as a consequence of an entity fault. This would assume that name manager services are dynamically resolved at the remote procedure call level. However, this will not work because exactly these services are used for entity localization during the binding phase of a remote procedure call. As a rule of thumb, name manager services are never resolved dynamically, rather they are addressed by means of name manager gates, e.g. the per-team domain identifier or some other identifier (i.e. *lui*) that is directly obtained from the name space.

A name manager fault is handled by a dedicated system manager, the *name usher*. This manager may be loaded dynamically by means of an entity fault. Note, in order to determine its scope a manager initially queries the entity manager. Upon success, the delivered name of a name manager is trying to be resolved. In case of a miss, the creation and installation of a new name manager is requested from the name usher. The name usher then performs name space editing to define a new scope.

### 6.2.3. Basic Requirements

Basically, incremental loading is controlled by two system processes, the entity manager and the name usher. These PEACE processes are required to reside in some node. For simplicity reasons, both fundamental managers share a single node called *radical node*. In fact, the radical node is the only one which is to be bootstrapped when the distributed/parallel machine is switched on – it serves as the origin of "life".

#### 6.2.3.1. Minimal Subset of Services

The minimal subset of system functions required for incremental loading are shown in Figure 9. Obviously, at least one name manager, *name*, is required that implements a single, flat name space. This name space contains names of system services which constitute the minimal subset of system functions. From this subset, the following services are required not only to support incremental loading, but also to provide

dynamic process management on a single node:

*thread* A kernel manager thread to create processes, i.e. thread and team objects.

*uio* A generic manager that provides basic and uniform **i/o** services. In this case the manager performs *disk i/o*. As was outlined, entity images are held in files.

*core* A generic manager thread that provides memory allocation and deallocation services.

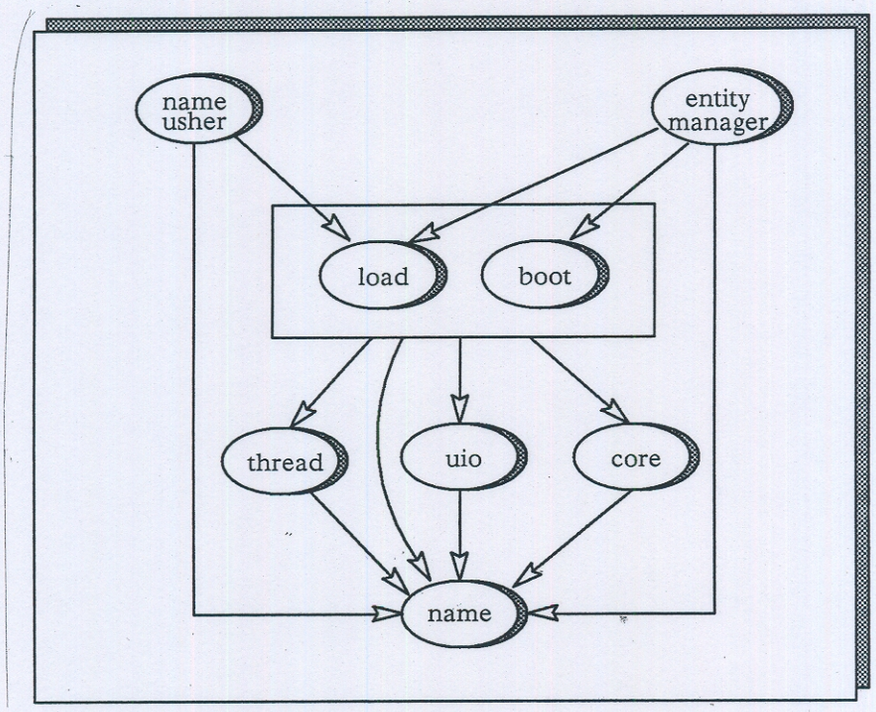


Figure 9: The radical node

These services are used by the load manager, *load*, to get entities running. The load manager is capable to dynamically relocate entity images, which is required if multi-tasking is to be supported without being based on a memory management unit that performs address translation.

In addition to the load manager, a boot manager, *boot*, is added to the minimal subset. This manager controls the bootstrapping procedure for a node. Thus, it acts as the master which directs slave nodes to initially load boot images. In situations where entities are to be loaded on passive nodes, either instantaneous or incremental loading is performed. In the former case, the entity and the kernel for that node are bootstrapped. In the latter case, the boot manager first performs the bootstrapping task to get the kernel running. Subsequently, the entity manager loads the entity by means of the load manager into that node. Bootstrapping only the kernel into a node is performed in configurations where the bootstrapped node is required to run a multi-tasking mode of operation. Note, the boot manager can be dynamically loaded at the time the boot services are required to switch nodes from passive to active.

#### 6.2.3.2. Configuration Descriptions

Basically, the entity manager interprets a *configuration specification* [Kramer, Magee 1985] that is contained in a data base. The data base is stored in a file. Upon startup, the entity manager inputs this file via *uio*.

The data base is automatically generated by compiling a configuration description written in the PEACE system configuration language. This language provides high level constructs to specify the distributed application in terms of entities and defines the distribution of the entities over the nodes. A *logical description* of the distributed applications is produced. For any type of entities, the following basic specifications are given:

- a *file name* for each entity, which is going to be loaded as part of the configuration;
- an *export configuration*, to direct the name export procedure;
- a *target node definition*, for site-dependent entities;
- any *environment data*, which is passed through to the entity;
- special *attributes*, to distinguish for example between the single-task and multi-task mode of operation for a given node.

Besides this logical description of a distributed application, a *physical description* of the actual architecture of the distributed/parallel machine may be specified as well. Based on this physical description, the initial mapping of distributed applications onto the machine is performed.

## 7. Object Mobility

PEACE supports mobility of active and passive objects. The former case means migration of teams consisting of a number of threads, which also may be caused by recovery from a node crash after having made a team checkpoint. The latter case involves virtual memory and virtually shared memory, and concerns the mobility of memory pages. The following subsections describe the basic PEACE approaches.

### 7.1. Migration

Migration is a method to transfer processes between nodes in a distributed environment, as for example realized by DEMOS/MP [Powell, Miller 1983] and Sprite [Douglis 1987]. In PEACE, migration is based on teams and is a basic mechanism for *load balancing*. It is also closely related to fault tolerant issues based on *checkpointing* and *recovery*. In fact, team migration can be viewed as an atomic sequence of team checkpointing, immediately followed by team recovery, whereby the team is recovered at a different node.

In this sense the basic approach of object (team) consistency and *synchronization* is the same for migration and checkpointing/recovery in PEACE. The major distinction between both types of object mobility is the granularity of synchronization. Usually, migration only requires synchronization related to a single team. In contrast,

checkpointing/recovery is related to a number of teams, jointly constituting a distributed application. Hence, synchronization is concerned with a team group.

Besides synchronization, the second major issue of migration is the *transfer* of the complete team context to another node. This context includes the address space, the threads, and all dependencies to external objects such as files, devices, teams and threads. Transparency for both the migrated team and threads/teams that interact with the migrated team is maintained.

Context transfer is to be performed in a consistent state, which requires to freeze the current team activities by means of synchronization. This state means that threads of other teams are stopped if they tend to communicate with the frozen team. Because PEACE supports only synchronous message passing, stopping these threads works implicitly. Non-blocking high-volume data transfers are temporarily inhibited.

Following the pattern of [Zayas 1987] in order to improve migration performance and throughput, the freezing state is shortened by shifting activities out of the period during which the team is frozen. The PEACE approach is to basically let the team migrate itself. By applying the administrator concept, a *migration porter* is capsuled by the team being migrated. The porter then controls the migration of its team, i.e., performs team preparation, issues a team migration request, and takes charge of team reconstruction. Similar to the process manager, a *migration manager* is only responsible for system data structure manipulation – team migration is realized as a *fork* crossing node boundaries with implicit termination of the parent.

Team reconstruction means to re-establish the original team context once transfer has been completed. For example, the migration porter flushes name caches, thus canceling any existing service connection. As soon as the threads of the migrated team call stub routines, service rebinding is performed relative to the new team location. In case of shared services, the same connection is re-established. However, in case of local services as, e.g., provided by the kernel, new connections are installed.

### 7.2. Checkpointing and Recovery

In contrast to migration, checkpointing and recovery in PEACE is related to a distributed application consisting of a multitude of teams. This requires a different synchronization scheme [Koo et al. 1987] which forces a group of teams into a consistent state. Nevertheless, the same approach as with team migration is used, namely to consequently apply the PEACE administrator concept.

In order to enforce *synchronization*, a *sift manager* (software implemented fault tolerance) preempts the teams belonging to the application that is subject to checkpointing/recovery. For this purpose, the *sift porter* of each team is assisted by a private thread that receives synchronization requests from its sift manager. Being synchronized, the sift porter either checkpoints or recovers its own team context.

Because of the absence of true synchronism – not only in the scope of massively parallel systems – a *virtual snapshot* of a distributed application is made. After having synchronized all the teams, this snapshot encompasses the address space (memory

segments) and activity space (threads) of each team involved in checkpointing. It is a distributed snapshot, because each team capsules its own runtime context. The team checkpoint includes a complete description of the team context at snapshot time. This checkpoint then is written by the sift porter to secondary storage in a special loadable format. Migration means to transfer the checkpoint to another node.

As in the case of migration, by having teams to freeze and write their own checkpoint, performance limiting bottlenecks are avoided. In addition, because the teams may be related to different disk I/O scopes, writing the distributed checkpoint in parallel is made feasible. The same is true for recovery, i.e. reading a previously written checkpoint. Thus, by means of the PEACE *sift administrator* a true distributed approach for checkpointing and recovery is realized.

Checkpoint writing is by means of a *two-phase commit* protocol [Kohler 1981] and thus represents an atomic transaction. A *log structured file system*, e.g. [Finlayson, Cheriton 1987], is used to store checkpoints. This guarantees the highest performance in writing and reading a team checkpoint. In conjunction with multiple disks and a team distribution scheme that enables parallel disk I/O, a powerful checkpointing and recovery facility is supported by PEACE.

A request for checkpointing is either issued by the application itself or by the system. The latter case means a timer-based approach. In any case, the same system function is invoked – the sift administrator simply receives a proper request from some other entity. In a similar way the request for recovery is triggered. Based on the *diagnosis administrator*, which controls the functioning of the massively parallel system, the recovery procedure for crashed nodes will be started automatically. This might also include a complete reboot of nodes.

### 7.3. Virtual Memory

The objectives of memory management in general are relocation, protection, logical organization and physical organization [Peterson, Silberschatz 1985]. To realize these objectives the PEACE concept uses *paged segmentation*, including the *demand fetch policy* for paging.

Paging as described below is a realization of a one-level store [Lister 1979]. It creates the extension of main memory with secondary memory, typically exclusively using a specific device or file for paging. The files on the other devices or just the other files have to be used via a separate file system interface. To create a "real" one-level-store, the PEACE operating system allows the whole secondary memory to be mapped into main memory, leading to *memory mapped files* as introduced by Multics [Organick 1972]. The main memory then can be viewed as the cache for secondary memory objects [Tevanian 1987][3)].

[3)] Explicit file buffering is no longer needed since the segments the files are mapped to function as "buffers".

Following the pattern of [Nelson, Ousterhout 1988], *copy – on – write* and *copy – on – reference* mechanisms are included into the PEACE virtual memory system. These mechanisms are used to support high-volume data transfer, team migration, and the creation of processes by means of *fork*. Both mechanisms work system-wide, crossing node boundaries. Basically, migration and *fork* are very similar in PEACE except for the difference that in case of migration pages are moved (*move-mapping*), whereas a *fork* leads to the copying of pages on-demand (*copy-mapping*).

It is also possible to share segments. Physically sharing memory is well-known but can be done only in memory accessible by all sharing processes. Since PEACE is designed for a distributed architecture with local memory, the *virtually shared memory* [Giloi et al. 1991] mechanism to logically share memory was included. The share mechanism even across node boundaries opens a range of uses. For example, sharing of a memory mapped file now means the sharing of the segment the file is mapped to[4] and *shared libraries* become a part of the system. This shows how the different virtual memory mechanisms influence each other in improving the system efficiency.

### 7.3.1. Page Faults

The PEACE virtual memory system is part of the *memory administrator* and, hence, realized in a distributed fashion. Page faults first are caught by the *page porter*, i.e. a trap handler residing at each node where virtual memory is supported. The page porter is part of the kernel and tries to handle the fault locally. In cases where additional support is required, the page porter forwards the page fault to its *page manager* by applying nucleus primitives. Thus, a page fault trap is converted into a message and sent to some other system process for further processing. This works system-wide in PEACE and usually addresses a page manager that is subject to user mode execution. Among other things, the page manager encapsules global page replacement strategies.

The page porter is executed on behalf of the faulting process, meaning that sending the trap message blocks this process. A rendezvous is established, and the page manager receives a page fault request. Having completed page fault handling, the page manager simply replies to the faulting process, resulting in the reactivation of the page porter at that site. The reply message contains porter control information. Before process restart is carried out, the page porter performs some local housekeeping, e.g., updating the per-process page table.

## 7.4. Virtually Shared Memory

Sharing of memory is a widely used scheme. The appearance of massively parallel systems with only local memory introduces requirements far beyond the simple technique of physically sharing global memory. In contrast to limited scalable memory systems like Memnet [Delp 1988], PEACE provides a way for logically sharing distributed memory. Thus, *virtually shared memory* is established as part of the memory system offering a

[4] The *secondary memory consistency* gets guaranteed by the mutual consistency protocol of the virtually shared memory.

location transparent application view of virtual memory in the distributed system.

### 7.4.1. Synchronization and Mutual Consistency

With regard to the distributed memory parts, replication and distribution of shared pages over the nodes is an important aspect for a virtually shared memory system in order to ensure efficient memory access time. This raises the problem of ensuring the consistency for all copies of a memory item. The *mutual consistency protocol* maintains this task through data access synchronization. While logically sharing within a single node with respect to *data access synchronization* is satisfied by mutual exclusion, e.g. semaphores, general sharing requires a different way to ensure the correctness of a (parallel) program.

Unlike other attempts, e.g. [Li 1986], the PEACE approach favors weakening consistency in addition to *strong consistency*. Strong consistency has the significance that a read of some location of shared memory returns the same value as had the most recent write to the same location. This causes a large amount of interactions between the participating nodes.

In order to minimize the overhead of keeping the distributed units of main memory consistent, an application specific definition of consistency is supported. Beyond the default mechanism of strong consistency, a weakening of the consistency condition is the essence of the mutual consistency protocol. This ensures more independence between the parallel operating processes, the key aspect [Hutto, Ahamad 1990] for achieving an efficient virtually shared memory system.

### 7.4.2. Strong Consistency

The strongly consistent virtually shared memory management in PEACE is built on top of standard paging mechanisms as used for virtual memory with demand paging. The basis mechanism is *paged segmentation*. Segmentation provides the logical organization of the address space, and paging creates the physical division of memory, representing local accessible data or references to remote data. Page fault handling works as in the usual virtual memory case, except that the page manager additionally encapsules virtually shared memory consistency strategies.

The main design issues for strong memory consistency protocols are the strategies used by the respective page manager. These strategies consist of a *page synchronization method* and a *page ownership strategy*. There are two basic page synchronization methods: *invalidation* and *writeback*. Both work on page granularity to keep the memory consistent. Evaluation results [Li, Hudak 1989] show that only the use of page invalidation yield acceptable performance. Furthermore, in the light of massively parallel systems consisting of several hundreds to thousands of nodes, only the (fixed/dynamic) distributed manager approach for page ownership is feasible.

In the PEACE virtually shared memory, the *invalidation approach* is based on a single page owner. The owner may be granted read or write access rights on the page [Giloi et al. 1991]. In the case of a replicated page, each sharing process, even the owner, merely

is granted the read access right on that page. Integrity is then ensured by properly defined page descriptors.

Page invalidation will take place if a *write fault* occurs, meaning that a process intends to manipulate a shared page. Basically, all page descriptors referring to the replicated page are invalidated and the read access right of corresponding processes are revoked. Furthermore, the faulting process becomes owner of the page and, if not yet available, the page will be copied into local memory. In case of a *read fault*, the process acting as page owner has revoked its write access right, while read access is still granted. The faulting process gains read access right and the corresponding page is copied into local memory. Figure 10 illustrates this protocol, which follows the *dynamic page ownership* approach.

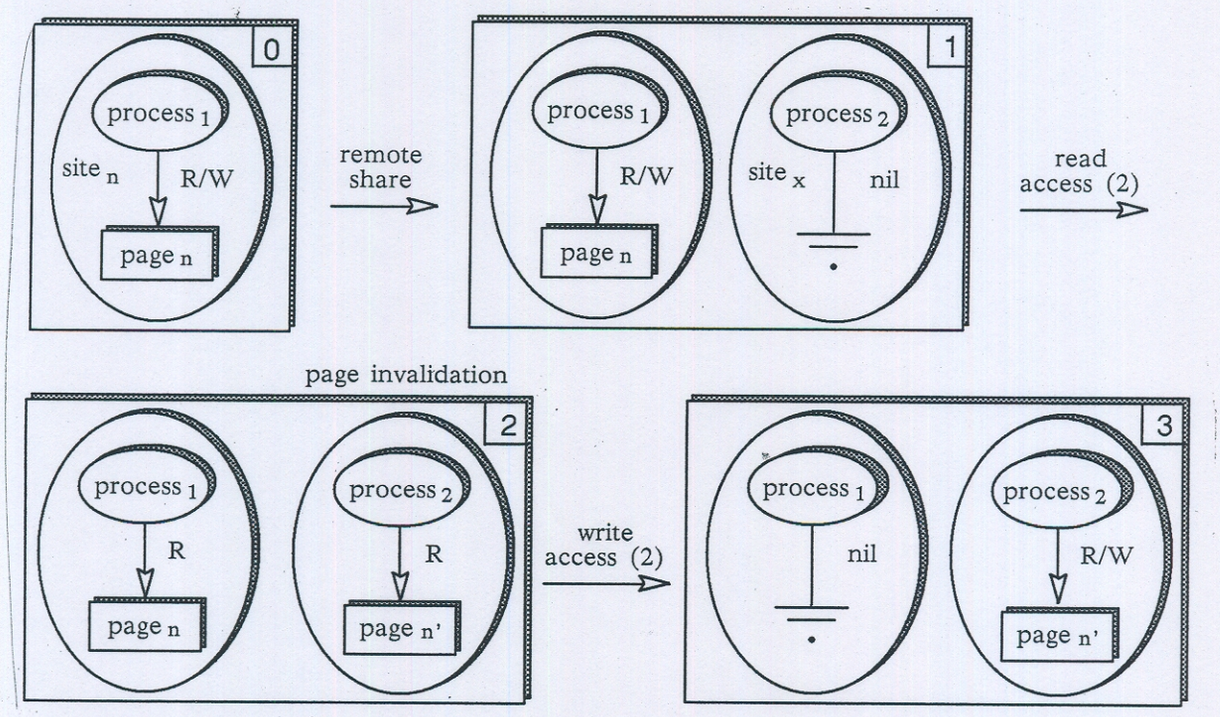


Figure 10: Page invalidation synchronization

As was shown, with this strategy page ownership changes as the page is written to. This can be managed either by a centralized manager or by a distributed manager. The centralized version has the advantage that management synchronization is easy since there is only one manager handling the pages. On the other hand, the overall communication overhead is large because of long network distances. Furthermore, the manager process (and the processing node it resides on) constitute a bottleneck when page fault rates are high. The decentralized/distributed version offers two variations: the fixed and the dynamic. Fixed distributed managers work like the centralized manager except that the pages to be managed are devided between several managers instead of one. This allows for more parallelism than in the centralized version but creates a

similar communication overhead. In the dynamically distributed manager approach, the manager is the owner site of the page. Thus, it changes too when the page gets write-accessed by another site. This policy adopts itself well to application programs which show unpredictable locality.

All the strategies described above show better or lesser performance depending on the application running on the virtually shared memory system. Therefore, the PEACE approach allows for any of these policies. The memory management is designed to support page manager which implement any of the existing (and we hope also most future) strategies. By means of configuration of system servers, the memory management system may be adopted to fit a particular application.

### 7.4.3. Weak Consistency

Strong consistency protocols include the problem to deal with frequent updates of a shared page used by multiple sites. These frequent write-accesses might easily cause *thrashing* [Li 1986]. In the Mirage system a strong consistency protocol is realized including a clock mechanism to keep the ownership of a page at a site for a certain amount of time [Fleisch, Popek 1989]. The efficiency of this scheme strongly depends upon the write-access rate, the number of processes write-accessing that page, and the time the page ownership is forced to stay at a site.

Another idea to avoid thrashing is *weak consistency*. It allows multiple writes into different locations of the same page [Bisiani et al. 1989]. Weak consistency clearly needs some compiler support to ensure that the processes use disjoint parts of a page. The advantage of weak consistency is its significantly lower page traffic, paid for by the disadvantage of loosing the transparency of the strong consistency protocols.

The PEACE design includes a novel approach. A weak consistency scheme is build on top of the strong consistency protocol [Giloi et al. 1991]. It is also possible that strong consistency and weak consistency coexist for a replicated page.

By default, the strong consistency protocol is used for each shared page. As shown in Figure 11, a process is able to leave strong consistency by indicating its entrance into a *weak block*. On demand, the involved pages are replicated into local memory and marked as copy-on-write. In case of a write access, a reference copy of the respective page is retained in local memory. Inside the weak block the process can modify the shared pages in local memory without transferring them upon write-accesses by other processes. The end of a weak block is again indicated by the process and triggers a procedure by which a difference set of the reference page and the modified page is built. This difference set then gets merged into the strongly consistent shared page by virtue of the strong consistency mechanism. As described in [Giloi et al. 1991], merging a number of difference set pages can be accomplished in a pipeline mode of operation.

In this decentralized (i.e., distributed) scheme, each process involved determines the changes and uses the strong consistency protocol to merge the different copies back into one page. This avoids the bottleneck of a centralized weak consistency protocol, where the merge is done by the page owner that was known at the time the weak block was

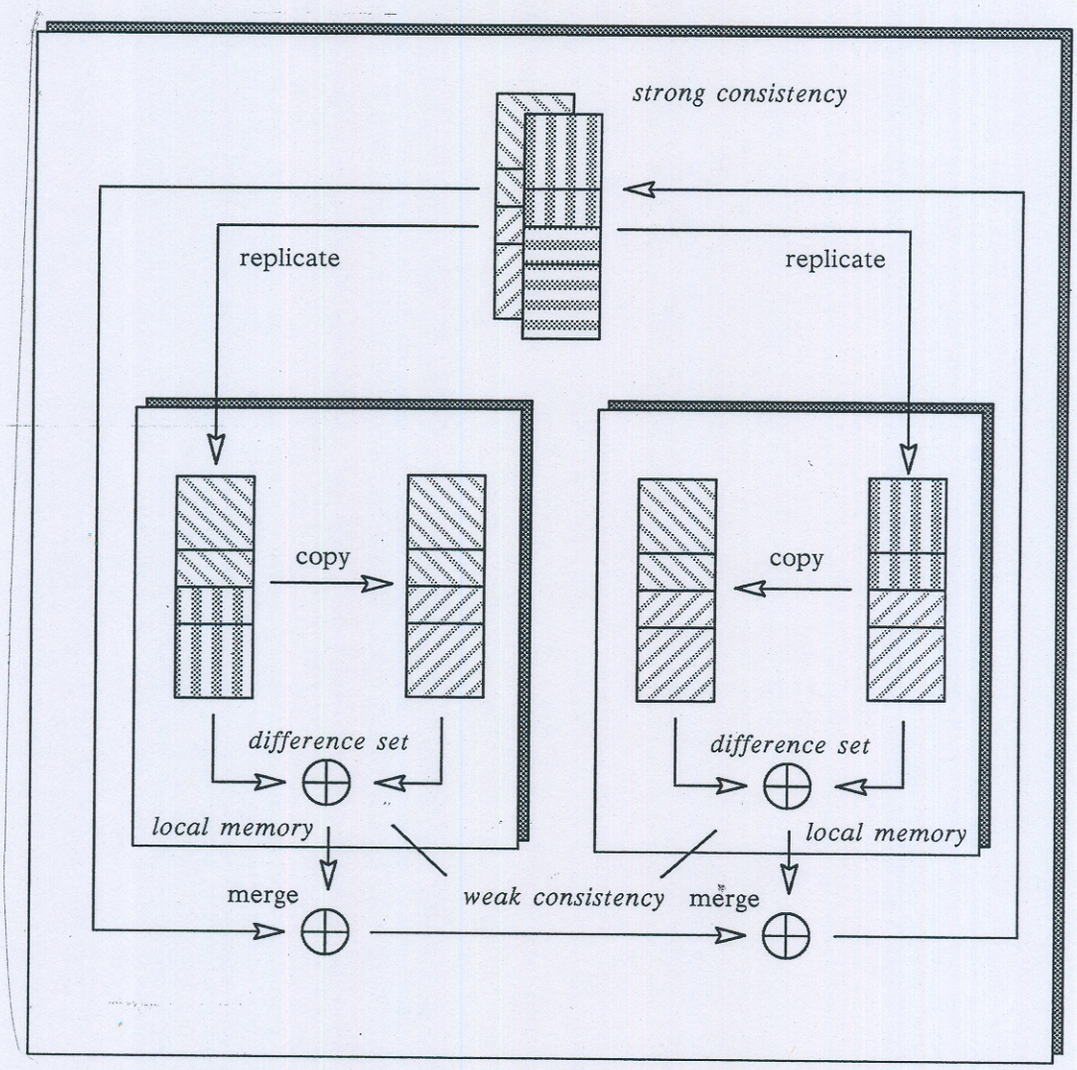


Figure 11: Weak consistency coexisting with strong consistency

entered. The decentralized solution furthers scalability, meaning that at runtime any number of processes can enter into weak consistency – they may even be dynamically created. That way the autonomy of a process also supports the use of specific synchronization mechanisms between processes that can be generated by tools or compilers, instead of global synchronization mechanisms as used by human programmers.

## 8. Concluding Remarks

The buzzword "*microkernel*" pertains to state-of-the-art operating system design. One of the most favorite systems falling into this category is Mach, i.e., the version known as the OSF/1 operating system [OSF 1990]. In the microkernel approach, the bulk of operating system services is executed in non-privileged user mode. Only a small set of services is subject to privileged supervisor mode execution. This organization

supports a fault tolerant, scalable and application-oriented system structure. It hence seems to be the appropriate basis for all fields of application. This is true for distributed systems, but does not hold entirely for massively parallel systems being based on a distributed memory architecture.

Even the microkernel is too complex and too overhead-prone if the very hard performance requirements of massively parallel systems are taken into account. The recently released Mach 3.0 microkernel is an implement of about 100.000 lines of C code (with comments excluded). This is in contradiction to what is really meant by a microkernel. The new PEACE kernel for MANNA supports the execution of parallel/distributed programs, being an implement of about 8.000 lines of C++ code (with comments included). Of course, even the PEACE kernel will become much more complex if, e.g., virtual memory gets to be included. However, this kernel then represents a different member of the PEACE operating system family. The PEACE user has the choice between several kernel representations and, as it is the case with Mach, is not confronted with a single solution only. Note, in order to support massively parallel systems, it clearly does not suffice to simply provide some sort of server system. The key question is not what should be included into a microkernel, rather one has to answer the question of how the entire operating system structure shall look like, so that a true microkernel is feasible.

The PEACE nucleus family approach overcomes the microkernel problem. What is really needed for massively parallel systems is a *"nanokernel"*, which provides only a minimal subset of services actually required by a given application. Additional system/kernel services then are loaded on demand, at the time when initially requested by the application. This also means that a microkernel is built by minimal extensions to the nanokernel, the minimal extensions being incrementally loaded. Operating system scalability is generally improved.

While the microkernel approach only promotes a scalable system organization for distributed systems, the nanokernel does so for massively parallel systems too – it promotes a scalable kernel architecture. Hence, a nanokernel bridges the gap between massively parallel systems and distributed systems. It makes it feasible that design principles of distributed systems can be applied to massively parallel systems.

The paper describes PEACE concepts for making massively parallel systems work. Although the elaboration of these concepts is focussed on massively parallel systems, they are general enough to make distributed systems work as well. The distinction between both types of system architectures basically is made at the nucleus level, especially by means of a family of NICE modules. As often required by the user community, support for parallel systems in general means to make the naked machine available to the parallel application program. Exactly this is realized in PEACE by having nucleus implementations that take simply the form of a communication library. The key aspect with the PEACE approach is, that even in this extreme situation there is no loss of scalability – due to the family concept, which is not only applied to the nucleus but also to the entire operating system.

Most of the issues addressed in the paper are based on experiences made with the first PEACE implementation for SUPRENUM. Since then, PEACE has undergone several redesign phases. For example, the nucleus family and especially on-demand loading of entities are brand-new features of the forthcoming PEACE system for MANNA. Another aspect is virtually shared memory, which still is to be integrated in PEACE. Moreover, a redesign of the current PEACE stub generator is necessary to meet the needs for a more flexible and powerful interface definition language. The same holds for the configuration language so far used. On-demand loading requires some more sophisticated constructs.

PEACE aims to be a high-performance process execution and communication environment especially for massively parallel systems. Thus, the primary goal is not (yet) to provide a complete timesharing and program development environment for any kind of user program. A host operating system is still required. As illustrated in [Schroeder, Gien 1989], the cooperation between PEACE and a state-of-the-art distributed host operating system like Chorus can be accomplished in a most efficient way. One important reason is that nearly all operating systems designed to date follow the same fundamental design concepts. That is, they are in some means process-structured and are based on a message-passing kernel. In the case of Chorus, for example, the PEACE nucleus would be a dedicated network manager implementation. In a similar way, cooperation with Mach could be achieved.

Last but not least, it should be mentioned that, for reasons of acceptance, the PEACE programming interface is related to that of UNIX®. All UNIX system functions are provided by means of a *compatibility library*. To be more precise, they are realized by the *UNIX administrator*. Nevertheless, achieving full UNIX compatibility was not the primary goal of PEACE.

Finally, the basic PEACE design decisions proved to be successful in as much as that they made a most efficient implementation of a multi-tasking message-passing kernel feasible [Schroeder 1991]. PEACE outperforms Amoeba, which claims to be the world's fastest distributed operating system [van Renesse et al. 1988]. Again, the family concept as realized in the PEACE approach was the key to success [Lennon 1969].

**Acknowledgments**

We wish to express our deepest gratitude to Dr. Peter Behr. Without his advice, support and patience during the SUPRENUM development, the PEACE project wouldn't have succeeded the way is has. We are also very grateful to Prof. Giloi for his helpful and constructive remarks on the final draft of the paper.

® UNIX is a registered trademark of AT & T Bell Laboratories.

**References**


[Balter et al. 1986]

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

[Balzer 1971]

R. M. Balzer: **PORTS – A Method for Dynamic Interprogram Communication and Job Control**, Spring Joint Computer Conference Proceedings, 485-489, 1971

[Birrell, Nelson 1984]

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

[Bisiani et al. 1989]

R. Bisiani, A. Nowatzyk, M. Ravishankar: **Coherent Shared Memory on a Distributed Memory Machine**, Proceedings of the 1989 International Conference on Parallel Processing, 1989

[Cheriton 1984]

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

[Clark 1985]

D. D. Clark: **The Structuring of Systems Using Upcalls**, ACM Operating Systems Review, 19, 5, Proceedings of the Tenth ACM Symposium on Operating Systems Principles, Orcas Island, Washington, 1985

[Delp 1988]

G. Delp: **The Architecture and Implementation of MemNet: A High Speed-Shared Memory Computer Communication Network**, Ph.D Thesis, University of Delaware, Department of Computer Science, 1988

[Douglis 1987]

F. Douglis: **Process Migration in the Sprite Operating System**, UCB/CSD 87/343, Computer Science Division, University of California, Berkley, February, 1987

[Finlayson, Cheriton 1987]

R. S. Finlayson and D. R. Cheriton: **Log Files: An Extended File Service Write-Once Storage**, ACM Operating Systems Review, 21, 5, pp. 139-148, Proceedings of the 11th ACM Symposium on Operating Systems Principles, Austin, Texas, 1987

[Fleisch, Popek 1989]

B. Fleisch, G. Popek: **Mirage: A Coherent Distributed Shared Memory Design**, ACM Operating Systems Review, 23, 5, Proceedings of the Twelfth ACM Symposium on Operating Systems Principles, Arizona, December 3–6, 1989

[Gentleman 1981]

W. M. Gentleman: **Message Passing between Sequential Processes: the Reply Primitive and the Administrator Concept**, Software Practice and Experience, 11,

435-466, 1981

[Gibbons 1987]
P.B. Gibbons: **A Stub Generator for Multilanguage RPC in Heterogeneous Environments**, IEEE Transactions on Software Engineering, Vol. SE-13, No.1, 77-87, 1987

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

[Giloi 1991]
W. K. Giloi: **GENESIS and MANNA – Goals, Concepts and Achievements**, in this issue, 1991

[Giloi et al. 1991]
W. K. Giloi, C. Hastedt, F. Schön, W. Schröder-Preikschat: **A Distributed Implementation of Shared Virtual Memory with Strong and Weak Coherence**, Proceedings of the 2nd European Distributed Memory Computing Conference, Munich, Germany, April 22 – 24, 1991

[Giloi, Schroeder 1991]
W. K. Giloi, W. Schröder-Preikschat: **Programming Models for Massively Parallel Systems**, International Symposium on New Information Processing Technologies '91, Tokyo, Japan, 1991

[Habermann et al. 1976]
A. N. Habermann, L. Flon, L. Cooprider: **Modularization and Hierarchy in a Family of Operating Systems**, Comm. ACM, 19, 5, 266-272, 1976

[Hewitt 1977]
C. Hewitt: **Viewing Control Structures as Patterns of Passing Messages**, Artificial Intelligence 8, 323-364, 1977

[Hutto, Ahamad 1990]
P. W. Hutto, M. Ahamad: **Slow Memory: Weakening Consistency to Enchance Concurrency in Distributed Shared Memories**, Proceedings of the 10th International Conference on Distributed Computing Systems, pp. 302-309, 1990

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

[Koo et al. 1987]
R. Koo, S. Toueg: **Checkpointing and Rollback-Recovery for Distributed Systems**, IEEE Transactions on Software Engineering, Vol. SE-13(1), 1987

[Kramer, Magee 1985]
J. Kramer, J. Magee: **Dynamic Configuration for Distributed Systems**, IEEE Transactions on Software Engineering, Vol. SE-11, No. 4, April 1985, 1985

[Lennon 1969]
J. Lennon: **Give PEACE a Chance**, The Plastic Ono Band – Live PEACE in