Making Massively Parallel Systems Work*

R. Berg, J. Cordsen, C. Hastedt,
J. Heuer, J. Nolte, 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 speed network. Making these systems work requires a very careful operating system design. They call for a distributed operating system that takes the form of a functional 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.

1. Introduction

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

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

* This work was supported by the European Commission under the ESPRIT-2 program and carried out as part of the GENESIS project, grant no. P 2702

latency, is a must.

In order to improve the acceptance of massively parallel systems – and of distributed systems in general –, the buzzword *virtual shared memory* [Li 1986] stands for a central functionality which is to be provided by todays distributed operating systems. For an overview of state-of-the-art designs and implementations see [Ousterhout 1989]. Nevertheless, for efficiency reasons, message passing as the fundamental programming paradigm in distributed memory systems will not be replaced by virtual shared memory. Rather, both functionalities are to be provided by the operating system in the most efficient way.

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 design 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 founds the basis for providing transparency without loosing efficiency. In this sense, PEACE aims to be a general solution for todays, 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. Following that, 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 virtual 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 process execution and communication environment 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 hereby was to extend traditional operating system family concepts [Habermann et al. 1976] into the area of distributed and massively parallel systems. In the following subsection, these mechanisms, i.e. abstractions, 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*. The former one, a *team*, basically being a shell for one or more instances of the latter one, a *thread*. 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, the same team scheduling strategies and common access rights onto PEACE objects. These objects are, e.g., memory segments, teams, threads, files, devices, and so on.

With each thread and team own (private) scheduling strategies are associated. That is to say, different threads within the same team, as well as different teams, may be scheduled following different strategies. Team scheduling always applies to a couple of threads and is triggered on a time slice basis. In addition, different teams may be given different time slices.

In PEACE, the potential for thread scheduling is given only by two situations, namely as a side-effect to the receive of either messages or events [Hewitt 1977]. One extreme is that threads are preempted and/or immediately started upon the receive of a message and 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 virtual shared memory where a couple of teams are considered to have direct access to a common, global address space that is physically distributed over a couple of nodes. From the notion of teams it is obvious that threads of the same team do have virtual shared memory access, too.

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. 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 happened to occur.

As was mentioned above, threads may be associated with own (private) scheduling strategies. 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 to say, the actual per-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 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 streams 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 are required to overlap. Effectively it means 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. By the way, 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 the client or by relaying the client to (maybe) another server. Replying the client results in the delivery of a server-defined response message. The successful reply

[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.

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 and 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 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. Streams

The transfer of arbitrarily sized data is not accomplished by the basic message passing primitives. A separate mechanism is provided. This mechanism is termed *high-volume data transfer* and is applicable only during a rendezvous, namely in the case of being in the possession of the reply capability. Under control of the server team, data streams 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.

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

In the case of virtual memory support, a *virtual transfer* of streams is made feasible. Rather than shuffling complete streams 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 similar, i.e. pages are copied that are written into by the server team or referenced by the client team.

2.5. Leagues

In order to provide security in case of 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 termed 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 user 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.

2.6. Gates

The addressing of communication partners is by means of *system-wide unique identifiers*. These identifiers are represented by a plain, numerical value and contain a *logical host field* that is interpreted as a hint where the effective location of the communication partner is. They are locally managed on each node, hence each one also is referred to as a locally unique identifier, *lui*.

A *lui* designates a *gate*, which acts as a port-like communication endpoint without buffering capabilities, but rather 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 dynamical 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 streams 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 a couple of 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 where 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 stored at that 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.

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 will be 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 issues, is dynamically alterable and application-oriented. Hence, first of all a *functional hierarchy* of system components is to 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 Processes

Distributed memory architectures call for an object-oriented system design, which also is a prerequisite to construct 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, *processes* are used to represent functional units.

3.2.1. Service and Manager

In operating system terms, a functional unit implements a system service, such as process management or file handling, and a system service is executed by a dedicated 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 termed *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 founds a natural basis to build application-oriented operating systems. System services need only be present if they are required, meaning 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 being a trap. A system call is to be requested by means of message passing, also distinguishing between local and remote operation. In order to hide all these properties from both the service user (client) and 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 potentially 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 for 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 functional replicated and over a couple of nodes distributed I/O units such as disks. In all situations managers are replicated for either 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 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 straight forward business. The extreme solution is to have an 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 couple 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.

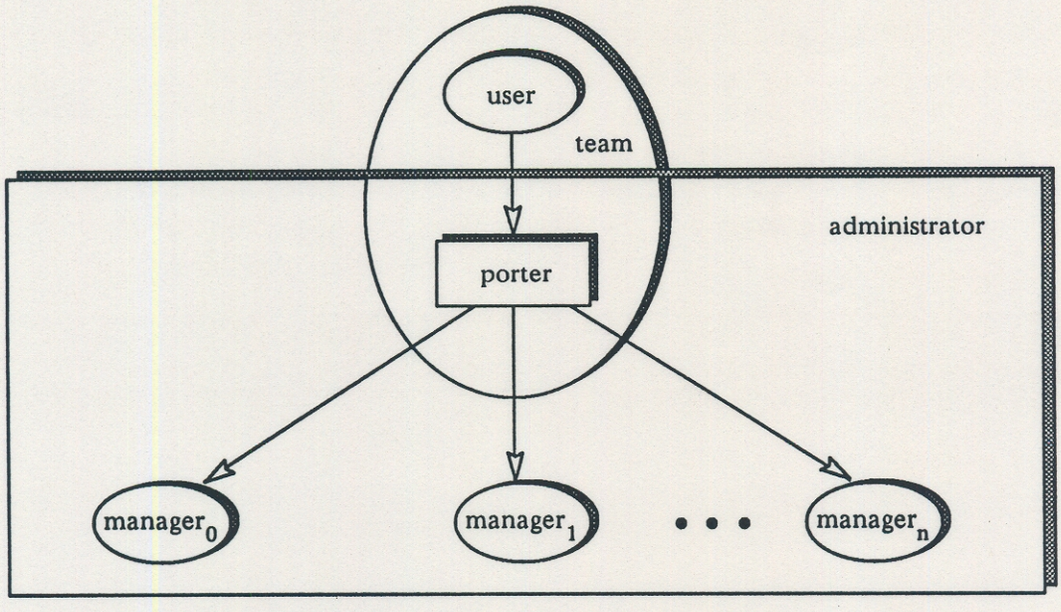

Figure 1: The administrator concept

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 capsule 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. At least 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 has to reside on a node where a disk is attached to and capsules device dependent functionalities. The file manager, which uses the disk manager, may reside elsewhere and is considered of being site-independent.

In many cases, the same service can show for 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 dependent 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 of performance, protection or hardware availability.

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

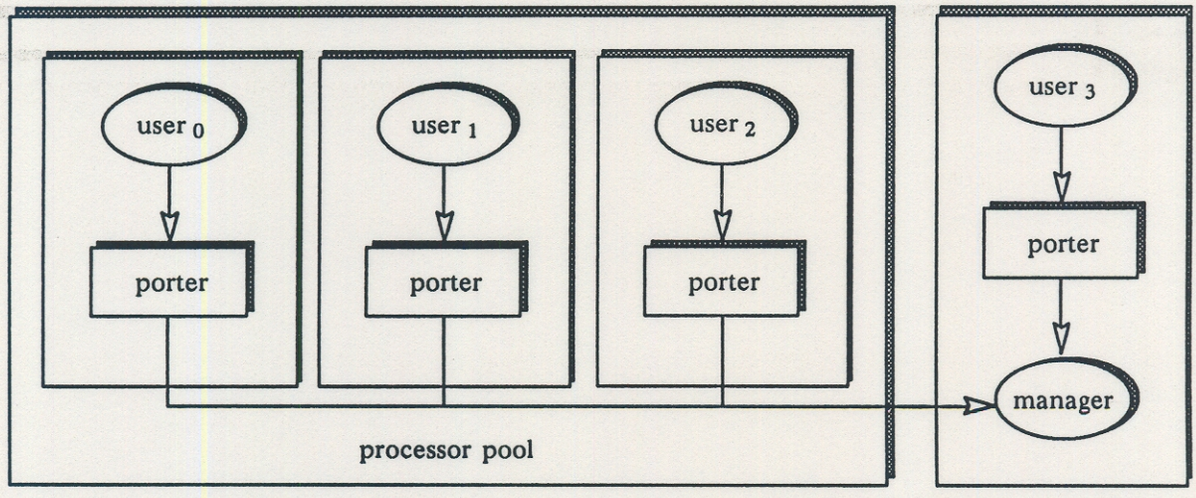

Figure 2: Functional dedicated system

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 with the GENESIS node [Giloi 1989]. The idea of this node architecture is to use both processors, which share the same node, as two dedicated functional units, a CPU for user process execution and a CP, the communication processor. In fact, distinguishing between CPU and CP, i.e. the assignment of a communication manager to the CP, then becomes a matter of configuration.

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. The creation of processes is considered, surely 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 to create the process, which image will 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 either of parent or child on the same node.

This leads to three fundamental functionalities, namely image transportation, routing of service requests and management of system data structures. For safety reasons, solely system data structure management is to 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 new created team.
4. The porter transfers its team image to the new team on the basis of 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 couple of process porter modules, each one capsuled by a team whose threads potentially request process creation. In addition, the process manager is responsible for process management on a couple of nodes, for example in cases where all the nodes together are used as a processor bank. Finally, the need for process manager replication is obvious if one is faced with the management of up to thousands 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 has to reside on each node. It provides *system-wide inter-process communication* and, thus, implements the basic PEACE abstractions. That is to

say, the nucleus implements objects related to threads, teams and leagues, events, messages and streams, and gates. However, they are not dynamically allocated by the nucleus. Whatever system component is responsible for the creation of these objects – by the way, exactly this is 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 than 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 absolute minimal subset of system functions means support for process execution and communication only.

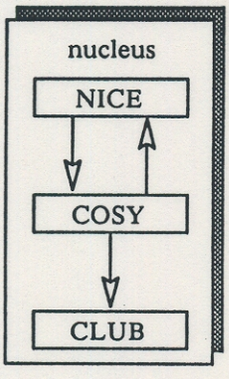

Figure 3: Nucleus organization

The top layer of the nucleus is dedicated to *NICE*, the network independent communication executive. 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 communication system. It 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 scheme of data transmission is realized. The COSY interface additional provides *upcalls* [Clark 1985] which are used to directly pass network events to the NICE layer. Usually, these events are related to incoming messages which are to be processed by NICE, accordingly.

The basic idea of COSY is to make NICE really independent from network interface capabilities, thus generally making the nucleus portable. Nevertheless, the COSY interface was tuned to meet the requirements for massively parallel systems, especially with respect to performance. This e.g. means to implement a high-volume data transfer facility that is optimal 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 data streams are tagged with different *luis*.

The bottom layer is dedicated to *CLUB*, the **cluster bus**. It capsules the network interface driver routines and, basically, is not concerned with any kind of network protocols excepted node addressing. Above CLUB, nodes are logically addressed. This 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 cases of remote communication, the *lui* is directly passed through to CLUB which, in turn, maps the logical host address onto a physical node address.

4.1.2. Nucleus Family

The nucleus, more specifically NICE, shows for several representations [Nolte et al. 1990]. The major distinction is made between single-threading, single-tasking, multi-tasking and multi-user support. In the case of single-threading and single-tasking, the nucleus may take the form of a communication library which is directly linked with the user task, i.e. the team. In the case of multi-tasking and multi-user, the nucleus is isolated from user tasks and resides within an own address space.

From the application viewpoint, the major difference between these representations is with respect to performance. For example, in contrast to the single-tasking mode of operation, the multi-tasking nucleus can only be activated by means of traditional system call invocation techniques, i.e. via traps. This property – in addition to team scheduling and address space isolation – introduces additional system overhead and, therefore, results in loss of communication performance. The advantage of this representation however is that scalability is increased, because more than one team can be executed on a node. Note, a single-tasking nucleus supports only the execution of a single team on a node. Consequently, applications only are scalable with respect to the number of available nodes.

Each nucleus representation stands for a member of a nucleus family, offering different performance characteristics and different functionalities. Based on a first PEACE nucleus performance analysis, it is safe to state that a rendezvous performed by the multi-tasking nucleus implies between 40 and 60 percent pure multi-tasking overhead which will not be present in a single-tasking nucleus. The

conclusion is to have a nucleus family that provides different support for different applications. Meaning that applications which scale well with the actual number of nodes will be supported by a single-tasking nucleus, and application which scale not well are to be supported by the multi-tasking nucleus.

4.1.3. Multiprocessor Support

Some members of the nucleus family also are subject 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 *functional dedicated* processors are used, as for example in the case of GENESIS [Giloi 1989], the nucleus is distributed over the processors. Figure 4 gives a rough idea about a functional dedicated nucleus representation for a single GENESIS node.

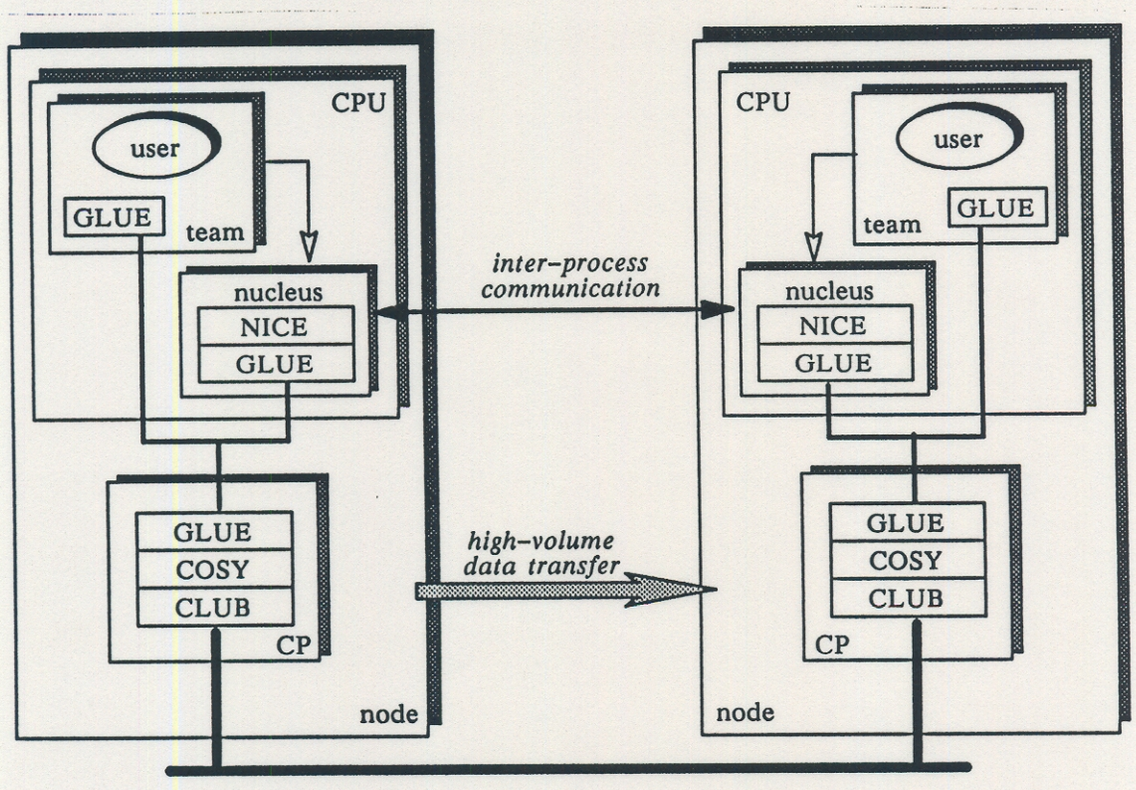

Figure 4: Functional dedicated nucleus organization

As illustrated, COSY and CLUB are mapped onto the communication processor, whereas NICE remains present on the processor that is subject for user program execution. Hence, the major part related to communication is offloaded.

This approach however requires an additional module, GLUE, for interfacing to the communication processor. Both, NICE and user teams have to use GLUE in order to request data transmission by the communication processor. This way, processes are able to interact directly with the communication processor. They use NICE only in cases where synchronization is required. Because the COSY interface is non-buffered asynchronous, *no-wait send* semantics of communication

[Liskov 1979] can be easily provided at team library level.

4.2. Kernel

Basically, the kernel serves to be a *hardware abstraction*. It offers services to dynamically create and destroy process objects and to associate these objects with address spaces. In addition, it capsules device drivers and enables higher-level system processes to attach 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 on a remote procedure call basis. 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 5 the kernel organization in terms of threads is exemplified.

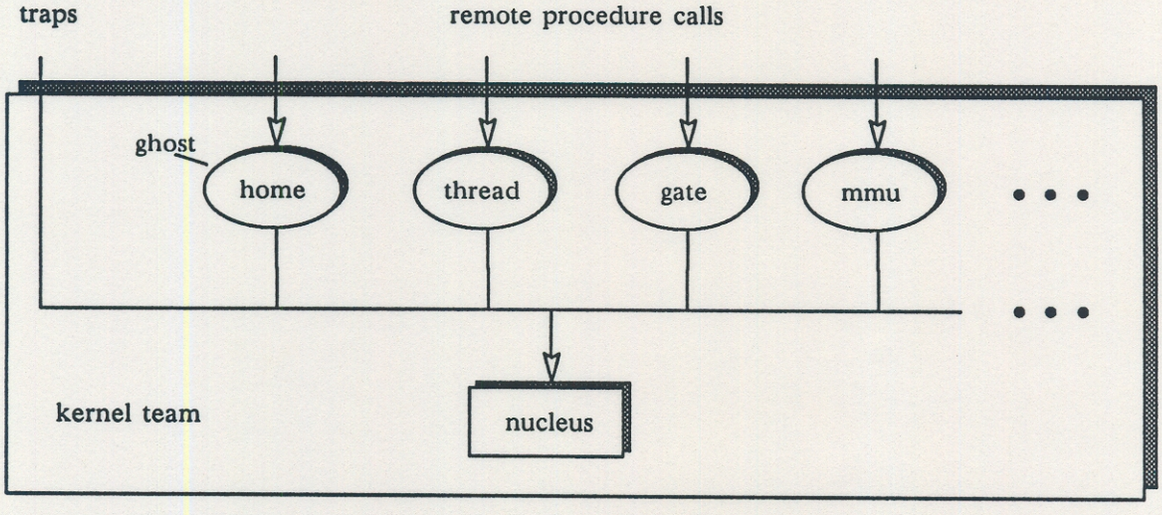

Figure 5: Kernel organization

Excepted the *ghost* (i.e. *home*), the presence of a kernel service depends on both the application and the hardware. Dynamic process creation and destruction is to 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. The extreme situation arises if single-tasking mode of operation is supported, only. 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 functionality it is obvious that, for some configurations, the kernel is to be executed in privileged supervisor mode of the processor. At least this will be the case if devices are to be managed, e.g. a memory management unit or a disk. On the other hand, the kernel, i.e. each kernel

thread, depends on the nucleus and, following the uses relation, is layered above the nucleus. Consequently, if the kernel is subject for supervisor mode execution, the nucleus is subject too.

4.3. System

The third PEACE building block to be discussed is that of 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 of being site-independent. Both properties make the operating system appear as an arbitrarily configurable building block. Figure 6 illustrates the complete PEACE system organization, including a building block of distributed applications.

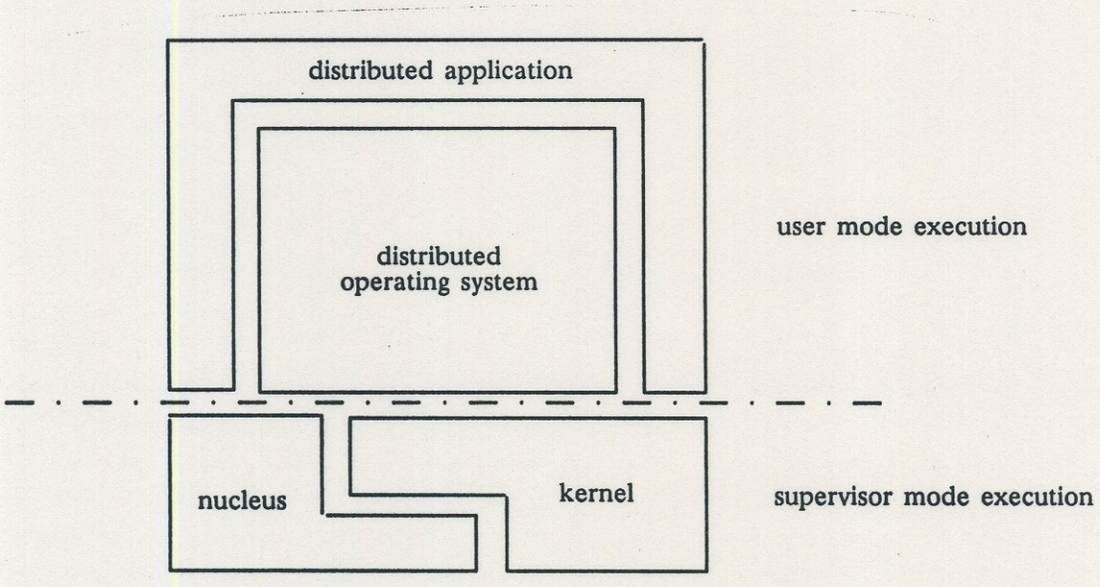

Figure 6: System organization

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

name	Provides services to locate objects in a distributed environment.
entity	Makes PEACE a dynamic 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 virtual 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.

load Performs the loading of teams.

There are other administrators dealing with *synchronization*, *migration*, *checkpointing* and *recovery*.

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. Together providing a powerful combination for 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 dynamical 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 1989], 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 turns out 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 semantic is considered of being the next minimal extension, thus introducing another family member. For reasons of

simplicity and efficiency, *at-most-once* semantic 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 independent from 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 manager 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 functions, 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 similar transparent scheme is that of using entity names, whereby in this case a single name stands for a couple of service functions. Effectively, both addressing styles are meaningful if a simple load splitting distribution scheme is used, but it does not hold for a distributed system which shows for 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 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 then is going to 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 going to be located.

Addressing by means of manager gates introduces some kind of object orientation. In this case the function going to be invoked deals with an explicitly defined object that helps to address the proper manager. Both, object and operations on that object are capsuled 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 to be managed, only. But it is efficient when such an object is e.g. a file and a couple of these 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 having been 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 capsule all the mystics of service invocation in a distributed environment.

In addition to these basic functionalities, 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 virtual 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.

The passing of complex, dynamical data structures is not supported by PRECISE, this is one of the domains of PEACE virtual shared memory. Moreover, problems that may be caused by different representation and alignment of data in the case of an heterogeneous environment are no concern in PEACE because massively parallel systems, for performance reasons, usually form a homogeneous environment.

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 a numerical value 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 then is used to map relative addresses onto effective addresses. Figure 7 illustrates the PEACE naming scheme.

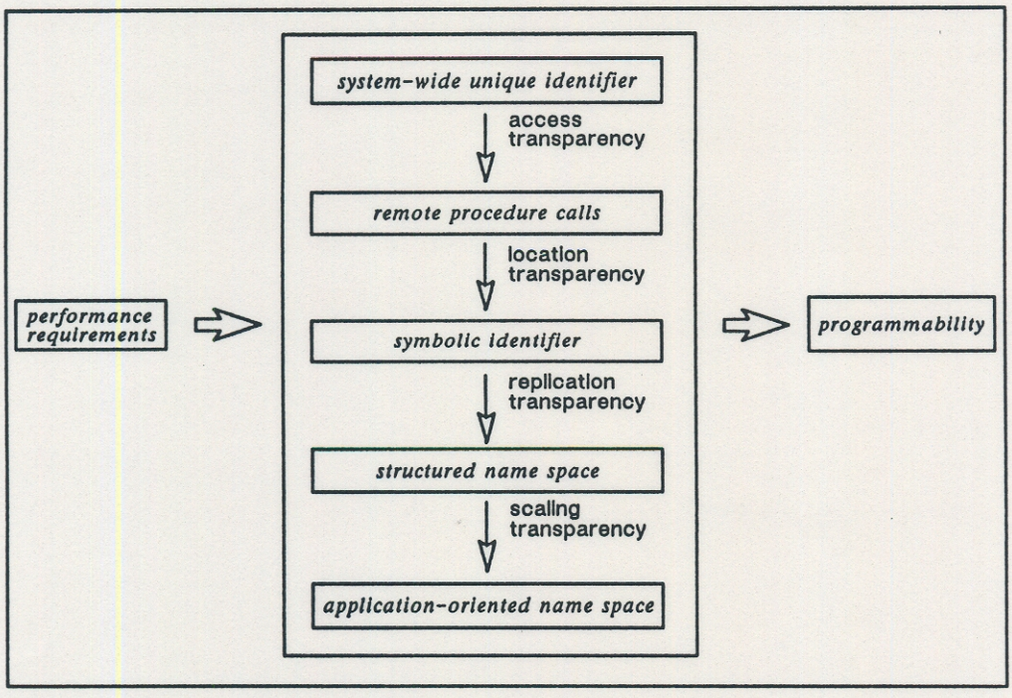

Figure 7: The PEACE Naming Scheme

System services are invoked on a remote procedure call basis, which 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 to

say, 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* is to 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 followed with 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 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 manager*. The entire name space then 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 on a remote procedure call basis. 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 giving the hint on 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 neither interprets the name nor knows its semantic. Each symbolic name is associated with a user-definable and variable sized data set. Resolving a name means the delivery of the data set. The contents 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 again. 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 run time, 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 again.

5.2.3. Name Space Organization

The global PEACE name space is implemented by a multitude of name manager, each one controlling a single name plane. A couple 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 creation of a name space itself is a dynamic activity and mainly depends on manager processes. In short, name manager are created on demand, namely when a new manager is integrated into the system and assumes the presence of a specific name plane. This also includes that name space editing is to be performed. A name server 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 to say, loading of system services 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 couple 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 a disk.

After having been reseted 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. As long as no team 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. It also enables that, dependent on the master, only a minimal subset of nodes are bootstraped. Furthermore, the slaves are not required to know any master, meaning that becoming mastership is a matter of configuration.

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 bootstraped. In addition, it contains information about the number of threads which reside within each team and where these threads reside. A thread is represented by a function, or procedure, of the program being bootstraped as a team. Thus, it is a program address, in conjunction with stack size and some thread/team attributes, which is to be stored as additional control information.

The first team being bootstraped 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 establishes teams and threads according to the boot image. 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 approach is to say, system service 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 handler", such that incremental loading of additional system services is possible [Schmidt 1990]. 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 for teams that belong to either of these application classes is *entity*. Note, teams are the units of distribution in PEACE. 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 a stub routine, once having determined that the entity is not yet available, 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 that loading will 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 single-tasking or multi-tasking mode of operation for the entity is also made.

6.2.2. Name Server 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 which 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 server 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 by the remote procedure call level. Unfortunately, this will not work because these services are used for entity localization. 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 server fault is handled by a dedicated system manager, the *name usher*. This manager, however, again 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 on 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 bootstraped when the distributed/parallel machine is switched on – it serves as the origin of "life".

6.2.3.1. Minimal Subset of Services

Despite the nucleus, the minimal subset of system functions required for incremental loading are shown in figure 8. 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 not only required 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.

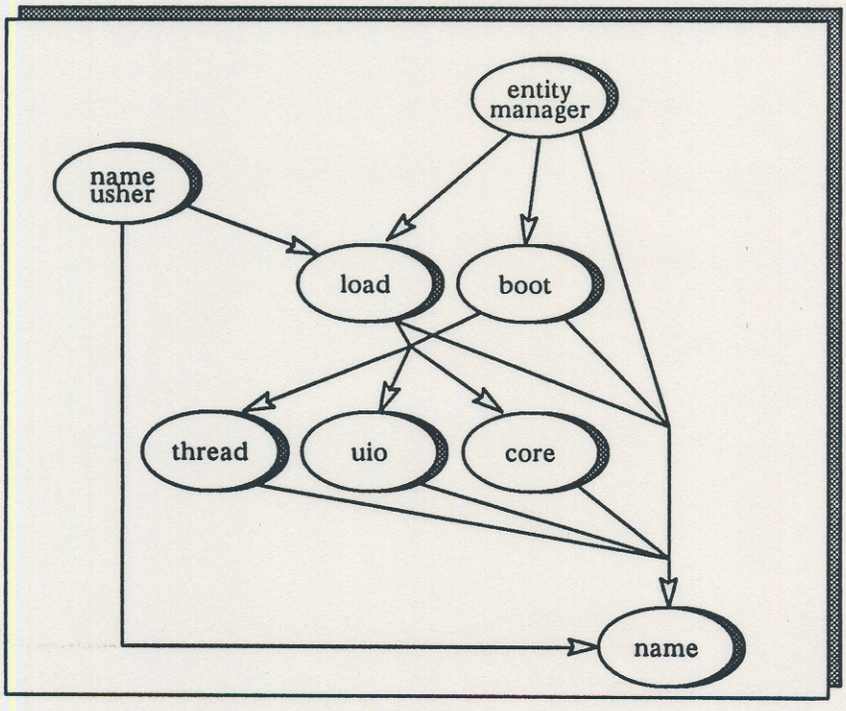

Figure 8: The radical node

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 multitasking 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 bootstraped. In the latter case, the boot manager first performs the bootstrapping task to get the kernel running and then the entity manager loads the entity by means of the load manager onto that node. Bootstrapping only the kernel on a node is performed in configurations where the bootstraped node is required to run in 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, i.e. operational, state.

6.2.3.2. Configuration Descriptions

Basically, the entity manager interprets a *configuration specification* [Kramer, Magee 1985] which 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 the compilation of a configuration description that was written in the PEACE system configuration language, AIDA. This language provides high level constructs to specify the distributed application in terms of entities and to give hints about the distribution of entities over a couple of nodes. By applying AIDA, a *logical description* of distributed applications is produced. For any types of entities, the following basic specifications are made feasible:

- a *file name* for each entity which is going to be loaded as part of this 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 single-tasking and multi-tasking 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 can be specified by AIDA constructs. Based on this physical description, 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 of a couple of threads, which is also due to recovery from a node crash after having made a team checkpoint. The latter case means virtual memory and virtual shared memory, and addresses 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 subject to 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 approaches of object (team) consistency and *synchronization* are 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 to that, checkpointing/recovery is related to a couple of teams, together constituting a distributed application and, 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 freezed team. Because PEACE supported synchronous message passing only, stopping these threads works implicit. Non-blocking stream 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 the team is freezed. This also includes in PEACE 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 it takes charge of team reconstruction. Similar to the process manager, a *migration manager* only is 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 consequently applying the PEACE administrator concept.

In order to enforce *synchronization* a *sift manager* (software implemented fault tolerance) preempts the teams belonging to the application being subject for 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 total 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, with 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 utmost 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. Any way, 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* that 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 (refer also to [Denning 1970], [Lister 1979], [Peterson, Silberschatz 1985], [Bach 1986], [Tanenbaum 1987]). To realize these objectives the PEACE concept includes several virtual memory mechanisms.

The first well-known mechanism is *paged segmentation.* The segmentation provides the logical organization of the address space and the paging creates the physical division of memory to implement a one-level-store [Lister 1979]. The PEACE concept includes the *demand fetch policy* for paging.

Paging as described is one realization of a one-level store. 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 filesystem 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][2)].

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 excepted 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*).

The inter-relationship between virtual address spaces exists not only during a copy (move) or copy-mapping (move-mapping). It is also possible to share lets say a segment. 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 accessible main memory the *virtual shared memory* mechanism to logically share memory got included too. 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[3)] and *shared libraries* become a part of the system. This shows how neat the different virtual memory mechanisms influence each other and improve the system efficiency.

The PEACE virtual shared 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

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

[3)] The *secondary memory coherence* gets guaranteed by the main memory coherence mechanism of the virtual shared memory.

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 for user mode execution. Among other things, the page manager capsules 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 going to be established and the page manager receives a page fault request. Having completed page fault handling, the page manager simply replies 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. Virtual Shared Memory

The technique of physically sharing memory is quite known for a while but can be done only in memory accessible by all sharing processes. Thus with the introduction of multiprocessors with local accessible main memory a way to logically share memory was needed. Logically shared memory is called virtual shared memory and it is the task of the operating system to realize the virtual shared memory as part of the virtual memory system on top of the distributed (physical) system [Li 1986].

7.4.1. Synchronization and Coherence

The key idea behind virtual shared memory is to replicate and distribute shared pages over a couple of nodes. First of all this requires *data access synchronization* by e.g. semaphores to guarantee mutual exclusion while manipulating shared data. There is no difference whether the application uses physically shared memory or virtual shared memory. In either cases data access synchronization is needed to ensure the correctness of the program.

In order to realize virtual shared memory there has to be a way to ensure *memory coherence.* Memory coherence means that a read of some location of shared memory must return the same value as the most recent write to the same location. A memory is *strong coherent* if it is coherent at all times.

With the memory being distributed and only locally accessible the memory coherence protocols will always cause some transfer of units of main memory from one site to another site to deal with updates. Since the processes which are sharing the memory are running in parallel, the most feasible *granularity* at the system level is a page allowing to use standard paging facilities. A large granularity of e.g. a segment would cause memory contention [Li 1986], excepted in cases of paged segmentation. The operating system then is required to implement some kind of protocol to ensure memory coherence.

7.4.2. Strong Coherence

The main design issues for strong memory coherence protocols are its strategies. 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 coherent. Evaluation results, [Li 1986] and [Li, Hudak 1986], show that only the fixed distributed manager and the dynamic distributed manager for page ownership using page invalidation for page synchronization yield acceptable performance.

In PEACE virtual shared memory the *invalidation approach* is based on one page owner only. The owner may be granted both read and write access right on the page. In the case of a replicated page, each sharing process, even the owner, merely is granted read access right on that page. Integrity then is to be ensured by properly defined page descriptors.

Page invalidation will take place if a *write fault* occurs, meaning that a process tends to manipulate a virtual shared page. Basically, all page descriptors referring to the replicated page are invalidated and the corresponding processes are withdrawn their read access right. Furthermore, the faulting process becomes owner of the page and, if not yet available, the page gets to be copied into local memory. In case of a *read fault*, the process acting as page owner is withdrawn its write access right, read access is still granted. The faulting process gains read access right and the corresponding page is copied into local memory. Figure 9 illustrates this protocol, which follows the *dynamic page ownership* approach.

As was shown, page ownership changes as the page gets written to. It 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 it is just one manager handling the pages. On the other hand, the overall communication overhead is big. The decentralized version offers two approaches: the fixed and the dynamic. Fixed distributed managers work like the centralized manager except that the pages to be managed are split between several managers instead of one. This allows more parallelism than the centralized version but creates a similar communication overhead. In the dynamic 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.

7.4.3. Weak Coherence

Strong coherence protocols include the problem to deal with frequent updates of a virtual shared page used by multiple sites. These frequent write-accesses might easily cause *thrashing*. In the Mirage system a strong coherence 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 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 coherence*. It allows multiple writes into different locations of the same page [Bisiani et al. 1989]. Weak coherence

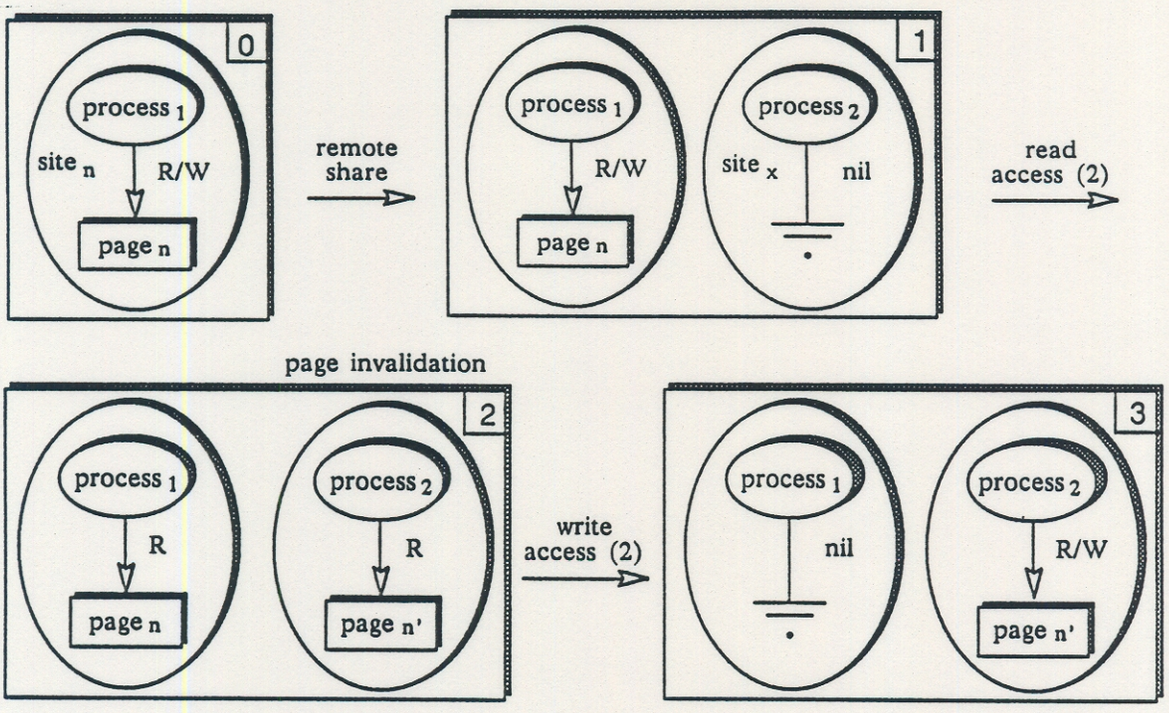

Figure 9: Page invalidation synchronization

clearly needs some compiler support for achieving full transparency to e.g. ensure that each involved process uses disjoint parts of a page. The advantage of weak coherence then is its significantly lower page traffic with the disadvantage of loosing the transparency of the strong coherence protocols.

The PEACE design includes a novel. Here, a weak coherence scheme is build on top of the strong coherence protocol [Giloi et al. 1990]. It is also possible that strong coherence and weak coherence coexist for a replicated page.

By default, the strong coherence protocol is used for each virtual shared page. As shown in figure 10, a process is able to leave strong coherence by indicating its entrance into a *weak block*. Inside of the weak block the process can modify the virtual shared memory pages without transfering them upon write-accesses by other processes. The end of a weak block is again indicated by the process and triggers that its copy gets merged into the strong coherence virtual shared page using the strong coherence mechanism. The following three phases are distinguished in PEACE:

replication Upon begin of a weak block a process does:

1. remove page from strong coherent virtual shared memory
2. get a copy of global reference page – either instantaneously or by copy-on-reference/write
3. duplicate it – one is working page and one is local reference page

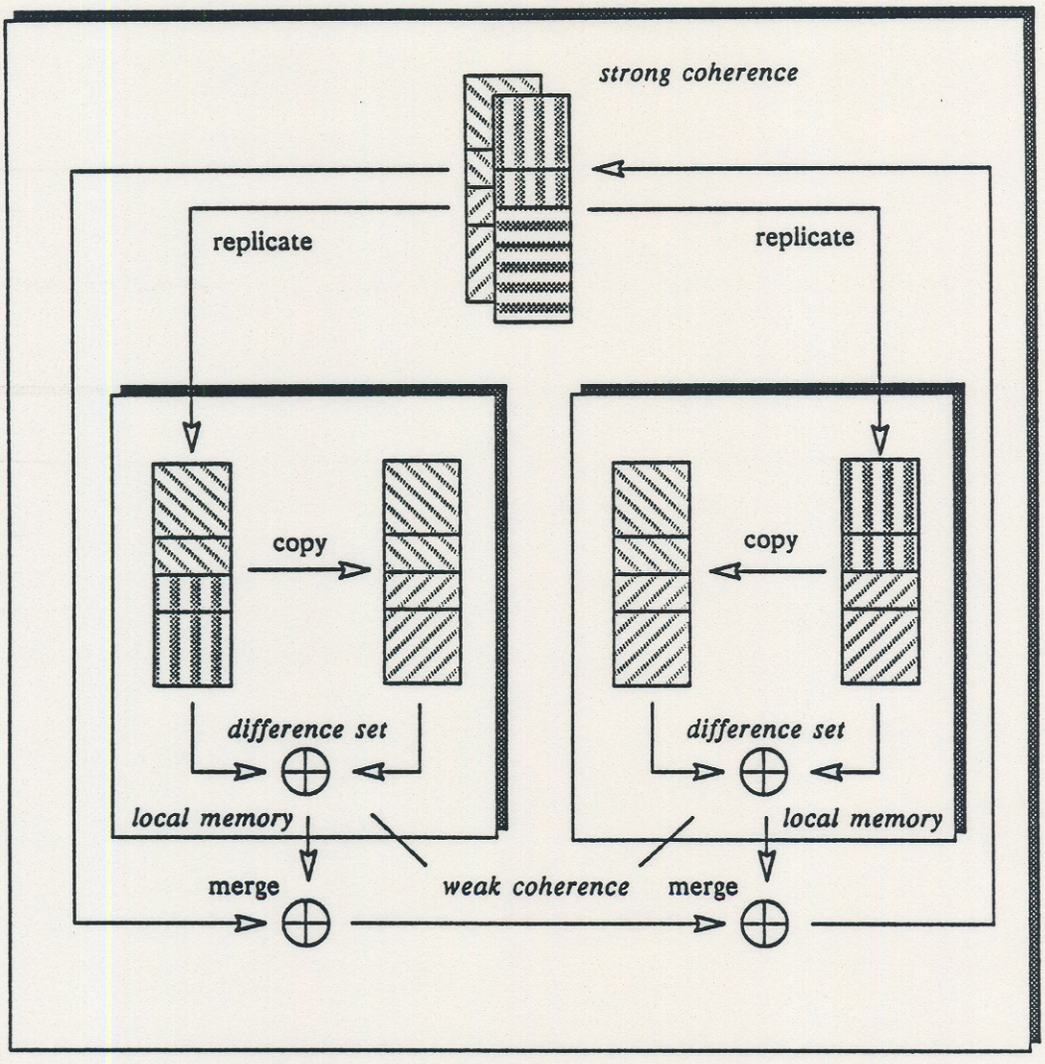

Figure 10: Weak coherence coexisting with strong coherence

4. allow write capability independent to other pages/processes

manipulation Work upon working copy

unification Upon end of a weak block a process does:

1. determine which bits have changed comparing working page and local reference page – a *difference set* is generated
2. place page into strong coherent virtual shared memory
3. merge the changes back into the strong coherence page

This description clearly shows that it is a decentralized scheme with each process involved determining the differences and using the strong coherence protocol to merge them back together. This avoids the bottleneck of a centralized weak coherence protocol, where the merge is done by the page owner that was known at the time the weak block was entered. This decentralized form includes also a high scalability meaning that at runtime any number of processes can enter the weak

coherence – they may even be dynamically created.

There is another interesting effect: Using the strong coherence protocol and locking of the page ownership and the write access right during the merge of a site causes a pipelining of the merging of pages. Let's say n processes share l pages using the weak coherence protocol and the pages are being merged by each process in the order $1..l$. Then after the setup of the pipeline and with a k $(1 < k < l)$ there exists a process i performing that operation on page k and a process j doing the same on page $k-1$. When process i is able to merge page $k+1$ process j is able to do so on page k and so forth. This shows how neat these operations can be done in parallel.

8. Concluding Remarks

The paper describes PEACE concepts for making massively parallel systems work. Although the elaboration of these concepts 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 loss of scalability is not really given – 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, a parallel machine consisting of up to 256 user nodes and 64 system nodes. In this sense, several redesign aspects have been considered. For example, the nucleus family and especially on-demand loading of entities are brand new features the forthcoming PEACE system will provide. Another aspect is that of virtual 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 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 to say, 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 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. This is another distinction between PEACE and Chorus, for example.

Finally, the basic PEACE design decisions proved to be successful for at least the fact that they made a most efficient implementation of a multi-tasking message-passing kernel feasible. Without loosing ourselves in presentation and analysis of detailed performance figures, it is safe to state that PEACE outperforms Amoeba, which claims to be the world's fastest distributed operating system [van Renesse et al. 1988]. These figures are subject of a paper to come [Nolte et al. 1990]. Thus, what still remains is to quote the following:

"all we are singing ... is give PEACE a chance".

Acknowledgments

We wish to express deepest gratitude to our *"little chief"*, Peter Behr. Without his support and patience, the PEACE project wouldn't survive. Thanks for all.

References

[Bach 1986]
M. J. Bach: **The Design of the UNIX Operating System**, Prentice-Hall, Englewood Cliffs (NJ.), ISBN 0-13-201757-1 025, 1986

[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

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

[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

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

[Denning 1970]
P. J. Denning: **Virtual Memory**, ACM Computing Surveys, Vol. 2, No. 3, pp. 153-189, September, 1970

[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 1989]
W. K. Giloi: **GENESIS: The Architecture and its Rationale**, ESPRIT Project No. 2447, Report of the GENESIS General Architecture Working Group, W. K. Giloi (Ed.), GMD FIRST, Berlin, FRG, 1989

[Giloi et al. 1990]
W. K. Giloi, C. Hastedt, F. Schön, W. Schröder-Preikschat: **Capability-Based, Distributed Implementation of Shared Virtual Memory**, submitted for publication, 1990

[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

[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

[Li 1986]
K. Li: **Shared Virtual Memory on Loosely Coupled Multiprocessors**, Ph.D. thesis, Yale University, 1986

[Li, Hudak 1986]
K. Li, P. Hudak: **Memory Coherence in Shared Virtual Memory Systems**, Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing, pp. 229-239, August, 1986

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

[Liskov et al. 1983]
B. H. Liskov: **Guardians and Actions: Linguistic Support for Robust, Distributed Systems**, ACM Transactions on Programming Languages and Systems, Vol. 5, No. 3, pp. 381-404, 1983

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

[Lister 1979]
A. M. Lister: **Fundamentals of Operating Systems**, (Second Edition), Springer-Verlag, New York, ISBN 0-387-91170-7, 1979

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

[Mierendorff 1989]
H. Mierendorff: **Bounds on the Startup Time for the GENESIS Node**, ESPRIT Project No. 2447, Internal Paper, GMD-F2.G1, 1989

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

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

[Nelson, Ousterhout 1988]
M. Nelson, J. Ousterhout: **Copy-on-Write For Sprite**, Computing Systems USENIX, pp. 187-201, Summer, 1988

[Nolte 1990]
J. Nolte: **PRECISE – The PEACE Interface Definition Language**, to be provided, 1990

[Nolte et al. 1990]
J. Nolte, J. Heuer, M. Sander, F. Schön, W. Schröder-Preikschat: **Making Massively Parallel Systems Fast**, to be provided, 1990

[Organick 1972]
E. Organick: **The Multics System: An Examination of its Structure**, MIT Press, 1972

[Ousterhout 1989]
J. Ousterhout: **Proceedings of the Twelfth ACM Symposium on Operating Systems Principles**, ACM Operating Systems Review, 23, 5, Arizona, December 3–6, 1989

[Parnas 1976]
D. L. Parnas: **Some Hypotheses about the 'uses' Hierarchy for Operating Systems**, Report, TH Darmstadt, 1976

[Parnas 1979]
D. L. Parnas: **Designing Software for Ease of Extension and Contraction**, IEEE Transaction on Software Engineering, Vol. SE-5, No 2, 1979

[Peterson, Silberschatz 1985]
J. L. Peterson, A. Silberschatz: **Operating System Concepts**, (Second Edition), Addison-Wesley Publishing Company, ISBN 0-201-06079-5, 1985

[Powell, Miller 1983]
M. L. Powell, B. P. Miller: **Process Migration in DEMOS/MP**, ACM Operating Systems Review, 17, 5, Proceedings of the Ninth ACM Symposium on Operating Systems Principles, Bretton Woods, New Hampshire, 1983

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

[Rozier et al. 1988]
M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Leonard, W. Neuhauser: **CHORUS Distributed Operating Systems**, Computing Systems Journal, Vol. 1, No. 4, University of California Press & Usenix Association, also as Technical Report CS/TR-88-7.9, Chorus systemes, Paris, 1988

[Sander et al. 1989]
M. Sander, H. Schmidt, W. Schröder-Preikschat: **Naming in the PEACE Distributed Operating System**, Proceedings of the International Workshop on "Communication Networks and Distributed Operating Systems within the Space Environment", Noordwijk, The Netherlands, October 24 – 26, 1989

[Schmidt 1990]
H. Schmidt: **Making PEACE a Dynamic Alterable System**, to be provided, 1990

[Schroeder 1988]
W. Schröder: **The Distributed PEACE Operating System and its Suitability for MIMD Message-Passing Architectures**, CONPAR 88, Manchester, UK., 12th-16th September, 1988

[Schroeder, Gien 1989]
W. Schröder-Preikschat, M. Gien: **Architecture and Rationale of the GENESIS Family of Distributed Operating System**, ESPRIT Project No. 2447, Technical Report, GMD FIRST, Berlin, FRG, 1989

[Shoch 1978]
J.F. Shoch: **Inter-Network Naming, Addressing and Routing**, Proceedings of the 17th IEEE Computer Society Conference, Compcon 78, Washington D.C., September 5 - 8, 1978

[Tanenbaum 1987]
A. S. Tanenbaum: **Operating Systems**, Prentice-Hall, Englewood Cliffs (NJ.), 1987

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

[Tevanian 1987]
A. Tevanian Jr.: **Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach**, Technical Report CMU-CS-88-106 (Ph. D. thesis), Carnegie-Mellon University, December, 1987

[Wirth 1971]
N. Wirth: **Program Development by Stepwise Refinement**, Comm. ACM, 14, 4, 221-227, 1971

[Young et al. 1987]
M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, R. Baron: **The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System**, ACM Operating Systems Review, 21, 5, Proceedings of the Eleventh ACM Symposium on Operating Systems Principles, Austin, Texas, 1987

[Zayas 1987]
E. Zayas: **Attacking the Process Migration Bottleneck**, ACM Operating Systems Review, 21, 5, Proceedings of the Eleventh ACM Symposium on Operating Systems Principles, Austin, Texas, 1987

[van Renesse et al. 1988]
R. van Renesse, H. van Staveren, A. S. Tanenbaum: **Performance of the World's Fastest Distributed Operating System**, ACM Operating Systems Review, 22, 4, 1988

[van Rossum 1989]
G. van Rossum: **AIL: A Class-Oriented Stub Generator for Amoeba,** Proceedings of the 1989 International Workshop on "Progress in Distributed Operating Systems and Distributed Systems Management", Berlin, West Germany, April 18-19, to be published in Lecture Notes in Computer Science, Springer-Verlag, 1989