# Towards a Scalable Kernel Architecture [*]

Jörg Cordsen
Wolfgang Schröder–Preikschat
German National Research Center for Computer Science
GMD FIRST Berlin
Rudower Chausee 5 (13.7), O–1199 Berlin–Adlershof, Germany
{jc,wosch}@first.gmd.de

July 27, 1992

”There is no tabula rasa.
We are like skippers
who have to rebuild their ship on the open sea
without ever being able to dismantle it in a dock
and reconstruct it of best parts.”
(Otto Neurath)

## Abstract

The paper starts with an examination of the notion *scalability*. Afterwards it discusses scalability issues in state–of–the–art kernel architectures, i.e., microkernels. It motivates the program family concept and object–orientation as the key to success in the design of an open microkernel architecture. To meet the special needs for massively parallel distributed memory machines, a kernel family is proposed by presenting the PEACE approach as case study.

## 1 Introduction

In the light of short–lived cycles of development it is of no surprise that nearly every hard– or software–system is decorated with the attribute of being *scalable*. Regarding kernel architectures, we hear about scalability just like multi–threading or virtual memory support. However, in difference to the latter concepts there is only a vague understanding of what scalability really means within a kernel architecture.

A lot of work has been done on scalable computing system architectures. In this context, scalability refers to an unlimited extensibility of computing systems, e.g., scaling up a 64–node system to a 512–node system without running into architectural bottlenecks. Usually, scalable computing systems are not based on shared memory, but on distributed memory, and often rely on

---

[*] This work was supported by the Ministry of Research and Technology (BMFT) of the Federal Government, grant no. ITR 9002 2.

simple processors with a limited general computation capability. In the area of parallel algorithms the notion of scalability is known as well. Despite different metrics, scalability of a parallel algorithm on a parallel machine is a measure of its capability to utilize a larger number of available processors.

But how are we supposed to understand the notion of scalability concerning kernel architectures ? Mainly, it is the task of a kernel to support the mapping of (parallel) algorithms on (parallel) computing systems. As mentioned, scalability issues are considered in the applications as well as in the underlying hardware architecture, but they are neglected by state–of–the–art kernel architectures. Up to now, kernel architectures rely on an *autonomous* design. Dependent on hardware properties a closed set of functionalities is provided, e.g., multi–tasking or virtual memory support. The kernel takes the same shape for all kinds of applications, thus, ignoring the concrete demands on an application. [1]

The autonomous design of a kernel enforces that the scalability is limited to a scale with exactly one mark. Accordingly, a scalable kernel must provide the whole set of functionality at all time. This spoils the scalability of a parallel algorithm, since the presence of unnecessary code–parts results in a reduced effectiveness. Such a drawback is unacceptable for a high–performance kernel architecture and suggests that autonomy and scalability are incompatible topics within a kernel architecture.

The paper starts with an examination of state–of–the–art operating systems, i.e., microkernel architectures. Afterwards it discusses the design of the parallel operating system PEACE [4]. It illustrates basic PEACE concepts and explains by what means support for massively parallel, distributed systems is given. Object–oriented mechanisms as well as strategies for dynamic system reconfiguration in PEACE are presented.

## 2 Microkernel Architectures

State–of–the–art operating systems are based on *microkernel architectures.* One of the most favorite systems representing this category is Mach [28]. A microkernel architecture is the attempt to decompose an operating system structure with the overall design rule to keep hold on those functions, whose processing on top of the kernel would be critical. The bulk of operating system services is accordingly executed in non–privileged user mode. Only a small set of services is subject to privileged supervisor mode execution. This organization supports a fault tolerant and application–oriented system structure. It hence seems to be the appropriate basis for all fields of application. This is true for distributed systems, but does not hold entirely for massively parallel distributed memory systems. Even the microkernel is too complex and, thus, too overhead–prone if the very hard performance requirements of massively parallel systems are taken into account. These requirements are to support a system–wide message startup time in the order of magnitude of 10 microseconds (using a 40 MIPS processor) [16].

---

[1] Some very first and timid attempts has been done to break up the inflexibility. These efforts are related to the notion of *extensibility.*

Mach is a good example to clarify what the climax of microkernel complexity can be. The Mach 3.0 kernel is a software system of about 100.000 lines of C code (with comments excluded). Alone this large amount of source code is in contradiction to the common understanding of the notion **micro**kernel. As comparison, the ancestor of the most successful operating system known to date was based on a kernel implementation of about 10.000 lines of C code (with comments included) [14]. Microkernel–based operating systems have been developed as counterpart to monolithic operating systems such as UNIX. However, this does not imply that the microkernel is no monolith too. The monolithic Mach microkernel is an order of magnitude more complex than the original monolithic UNIX kernel.

From the functional point of view, standard microkernels as used in Mach and Chorus [22] typically encompass interprocess communication, scheduling, security, process management, (virtual) memory management and exception handling. These are functions to support a multi–tasking mode of operation. They are necessary to process microkernel–based operating systems where services are provided by tasks being executed in user mode. Thus, even if an application does not require this operation mode, it must pay for it. This introduces significant overhead for those types of parallel applications which expect that tasks are mapped in one–to–one correspondence with the nodes of a massively parallel distributed memory system [25]. Multi–tasking is not free of charge and it is only needed if the one–to–one mapping can't be done for all the tasks.

It is true that, in terms of software engineering arguments, a microkernel must not be of minimal size [9]. However, are all the functions really mandatory fundamental building blocks? It obviously depends on the application field. Only if applications always demand these functions, either explicitly or implicitly, then a microkernel of this complexity is the right choice. A further software engineering argument is to have available for user applications only those functions which are really demanded. This is the only way to keep kernel complexity as small as possible, to have the chance to understand potential performance bottlenecks and to be really application–oriented [21]. In addition, for security reasons complexity must be sacrificed as far as possible.

## 3 Approaching the Concept of Program Families

Forthcoming massively parallel systems are distributed memory architectures and will consist of several hundreds to thousands of autonomous processing nodes interconnected by a very high–speed network. A major challenge in operating system design for these parallel architectures is to elaborate a structure that reduces system bootstrap time, avoids bottlenecks in serving system calls, promotes fault tolerance, is dynamic alterable, and application–oriented. At the same time utmost highest communication performance must be provided. The solution to these problems is an approach in which an operating system is understood as a *family of program modules* [21] and not as a monolithic "saurian" of more or less related components.

A parallel operating system has to provide only a *minimal subset of system* *functions*. Driven by the application, additional system/kernel services are to be considered as *minimal system extensions* [21]. In order to optimally support applications, minimal system extensions then are loaded on demand, at the time initially requested.

This approach especially would mean that a microkernel is built by minimal extensions to a "**nano**kernel" and the minimal extensions are subject for incremental loading. Operating system scalability is generally improved. While the microkernel approach promotes a scalable system organization for distributed systems, the "nanokernel" does so for massively parallel systems too – it promotes a scalable kernel architecture. Hence, a "nanokernel" bridges the gap between massively parallel systems and distributed systems. It makes it feasible that design principles of distributed systems can be applied to massively parallel systems.

### 3.1 Minimal Basis

In the program family concept a *minimal subset of system functions* provides a common platform of fundamental abstractions. This minimal basis encapsulates solely *mechanisms* from which more enhanced system functions can be derived. It will be built by a consequent postponement of design decisions. Fundamental abstractions to make massively parallel systems work then are *processes* and *communication*, i.e., message passing.

Processes introduce scaling transparency and, thus, make the modeling of parallel applications independent from the actual number of processing nodes. Scaling transparency, however, is not only an issue in the programming of massively parallel systems [10], but improves also availability in the case of permanent nodes failures. Even if the application is tailored to the actual number of processing nodes, the crash of a single node could mean the premature end of application processing if the system does not support the migration of program activities onto still functioning nodes. For this purpose the system needs an instrument for the modeling of program activities, which is the process. Thus, a process serves as the common abstraction for both the application and the operating system.

Communication based on message passing is a must when processing nodes have direct physical access only to local memory. Access to non–local memory, i.e., to the local memories of other nodes, involves the execution of a network communication protocol. Because processes form the basic abstraction to model (user/system) activities, interprocess communication rather than internode communication is required.

Whether synchronous or asynchronous communication should be supported strongly depends on the process model and on its implementation [3]. Synchronous interprocess communication is the best choice in order to achieve the maximal utilization of network bandwidth. In contrast to asynchronous communication, intermediate buffering and, hence, additional overhead of message copying is not implied by the communication model; at most, it will be implied by the network hardware interface.

The decision for synchronous interprocess communication implies a potential loss of parallelism. This must be compensated by a process model which allows concurrent programming even on a single node, i.e., which supports a *multi–threaded address space*. Obviously, the implementation of this process model must lead to a process switch time which is significantly smaller than the copying and buffering overhead involved in asynchronous communication. Such a model is mechanized by *lightweight processes* [13] and being implemented as *featherweight processes* [10] to meet the performance requirements for parallel operating systems. The minimal basis then strongly promotes a processing model in which concurrency is not a side effect of communication, but is expressed explicitly by means of threads building a *team*. It implements a **p**rocess **e**xecution **a**nd **c**ommunication **e**nvironment (PEACE) for parallel/distributed applications.

### 3.2 Minimal Extensions

A minimal basis which supports threading and communication already suffices to execute parallel programs. Moreover, it could be considered as the only operating system support residing on a processing node and being required by the application. In these dedicated applications the minimal basis is already the optimum. Additional functions are not used on the nodes and, hence, would only withhold system resources (such as memory space and processor time) from the given application.

It is the second important feature of the program family concept, that, dependent on the individual application, a stepwise functional enrichment of the minimal basis is performed by means of *minimal system extensions* only. These extensions encapsulate *mechanisms* and/or *strategies*. However, it might be the case that no system extensions are necessary at all. The application itself is always the best extension one can think of – it is the final extension anyway.

By adding minimal extensions, an operating system family is constructed bottom–up, whereby construction is controlled top–down: lower–level components are introduced only when required by higher–level components. This way, system functions for scheduling, security, process management, (virtual) memory management, exception handling, file handling, checkpointing and recovery are introduced. An open, application–oriented and evolutionary system organization is the consequence.

Understanding functional enrichment as an add–to in terms of components is only one aspect. It also includes *component replacement*. During the design phase a commitment on the minimal subset of system functions must be made. This includes the risk of stating wrong design decisions. One of the most important decisions is concerned with the identification of the proper operation mode, i.e., whether single–tasking or multi–tasking is to be supported.

The processing of parallel applications by a massively parallel machine always implies communication, hence the need for communication functions. The application might also call for a single or multi–threaded address space (i.e., task) on a node. Another application demands multi–tasking, which then is a functional enrichment of multi–threading. Should the minimal basis therefore support multi–tasking? If the design decision advocates multi–tasked nodes and tasks are mapped in one–to–one correspondence with the nodes, then a significant degradation of the message startup time will be the result [25]. Multi–tasking is not free of charge, even if not utilized by the application. A design decision to support solely multi–tasked nodes will handicap single–tasking applications and, thus, will not be conform with the idea of program families.

To overcome this problem, all applications must see the same external (abstract) interface of the minimal basis. What differs is the internal behavior, i.e., the concrete implementation. The external interface is mainly concerned with communication, while the internal behavior mainly dictates the process model and the operation mode of the node. With the minimal basis being an *abstract data type* [15] a number of implementations of the same interface can coexist. This makes the minimal basis exchangeable at least from the design point of view. Flexibility is maintained although the minimal subset of system functions must have been fixed early in the software design process.

## 4 The Role of Object Orientation

Applying the family concept in the software design process leads to a highly modular structure. New *system features* are added to a given subset of system functions. One instrument to implement a program family is to apply the abstract data type mechanism. An instance of an abstract data type is implemented by a module. System functions then are represented by the operations which are defined by the module interface specification. The entire system ends up with a multi–level hierarchy of a multitude of program modules, with a well–defined *uses relation* [21] between the modules to associate them to levels in the hierarchical system.

A problem with the module–oriented approach is the potential for a large number of redundant code and data portions in those cases where different implementations of the same module interface coexist [6]. That the redundant portions are not encapsulated by an abstract data type on its own, i.e., extracted and implemented by a separate "service module" and then being properly used by the instances is due to at least two facts: *genericity* and *efficiency*. One often examines that the new service module must be capable to deal with objects of different type, whereby the type is defined by those instances which will use the new module: the new module is generic. Having strongly followed the pattern of abstract data types, the additional module boundary often implies an increase in runtime overhead due to additional procedure calls for operation invocation: the new module introduces a potential performance bottleneck.

The feasibility of this kind of abstract data typing depends on the power of the programming language to implement generic module interfaces and on the function inlining capabilities of the compiler. If a parallel operating system is required to guarantee a message startup time in the order of magnitude of 10 microseconds (assuming a 40 MIPS processor), any increase of runtime overhead caused by either of programming paradigm, programming language or compiler is not acceptable. [2] The much more promising approach in the

---

[2] The first PEACE kernel prototype for a distributed memory parallel computer was im-

design and development of operating system families, therefore, is to apply *object orientation* [27]. In other words, object orientation is the natural choice to build program families [7]. The buzzword is *inheritance* [12] to avoid large portions of different module versions to be identical. Functional enrichment defines new family members, which always inherit properties of existing family members. The new family member is built by at least one new specialized class by derivation from one or more base classes (*single/multiple inheritance*). This implies the re–usage of existing implementations on a sharing basis, meaning that code/data redundancy will never appear in a clean object–oriented design.

In *class–based object orientation* [5], the class definition includes the implementation of the methods defined on objects of that class. This makes function inlining straightforward and, hence, reduces the procedure call overhead to an absolute minimum. An example is C++ [26], which also supports *abstract data type based object orientation.* Note, the major problem with identical portions of different module versions primarily is not wasted memory space, which function inlining implies too. Above all, it is a software maintenance problem, which (class–based) object orientation with or without function inlining helps to avoid.

There is another feature of object orientation which is of importance for the implementation of a family of operating systems. This feature is known as *polymorphism.* A base class specifies the operations which are defined on objects of that class. In the course of inheritance, a derived class may specify either the same operations again or a subset only. These redefined operations usually show for a different, more specialized implementation. The external class interface is still described by the same base class, while different implementations of the same interface can coexist by means of inheritance and dynamical binding. Polymorphism strongly supports the design and implementation of replaceable components. Featuring the proper derived class is dynamic and works transparently to the instance applying the base class only.

# 5 A Parallel Operating System

The PEACE family concept distinguishes between a macroscopical view to identify the overall system architecture and a microscopical view to define the minimal subset of system functions that must be present on each node. The former aspect deals with distribution and the latter aspect deals with performance.

## 5.1 Macroscopical View

A member of the PEACE parallel operating system family is constituted by three major building blocks: *nucleus*, *kernel*, and POSE (Figure 1). The nucleus implements system–wide interprocess communication and provides a runtime executive for the processing of threads. The PEACE nucleus is part of the kernel

---

plemented in Modula–2. Performance was not acceptable. A transformation into C and non–optimized compilation let to a negligible performance improvement. Applying the keyword "*register*" at meaningful places and with optimized compilation, a 40 percentage performance increase was obtained [24]. Register optimization let to fewer memory traffic, which is significant if the processor executes 3 (4) wait states on each read (write) memory access.

Figure 1: Building Blocks

The dividing line between user and supervisor mode is a logical boundary only. It depends on the concrete representation of the interactions specified by the functional hierarchy (and of the processor architecture) whether this boundary is physically present. The functional hierarchy of these components (Figure 2) defines the way decentralization works with Peace. All components are encapsulated by (active/passive) objects. An object invocation scheme must therefore be used to ask for service execution.

Nucleus services are made available to the application via *nearby object invocation* (Noi). The logical design assumes a separation of the nucleus from the application (and Pose), which calls for the potential of address space isolation and of traps to invoke the nucleus. This is the place where *cross domain calls* may happen. The kernel shares with the nucleus the same address space and, hence, performs *local object invocation* (Loi) to request nucleus services. Kernel services are made available via *remote object invocation* (Roi) [19], an object–bound mechanism similar to the *remote procedure call* paradigm [18]. Services of Pose are requested via Loi and Roi. Here, Loi is used to interact with the Pose runtime system library and Roi is used to interact with the Pose active objects.

From the design point of view neither the kernel nor Pose need to be present

Figure 2: Functional Hierarchy

on each node, but the nucleus. In a concrete configuration, the majority of the nodes of a massively parallel machine is equipped with the nucleus only. Some nodes are supported by the kernel and a few nodes are allocated for Pose. All nodes can be used for application processing, but they are not all obliged to be shared between user tasks and system tasks.

It is important to understand that the functional hierarchy of the three building blocks expresses the logical design of Peace only, and not necessarily the physical representation. The building blocks are designed with respect to the various schemes of object invocation as shown in Figure 2. However, it depends on the actual operating system family member whether these schemes become effective as specified by the design or can be replaced by a more simple alternative. For example, although the functional hierarchy assumes Noi for the interaction between application (Pose) and nucleus, the Loi scheme is used for those members of the nucleus family which place their focus on performance and support single–tasking mode of operation only.

### 5.2 Microscopical View

A process execution and communication environment forms the minimal subset of system functions required by massively parallel systems. This minimal basis of Peace is a compromise between *transparency* and *efficiency*. For different applications there are different implementations of the same interface of the minimal basis, hiding all the internals. This transparency is to the convenience of the application programmer.

The minimal basis is defined as a *family of functional dedicated units* with a single external interface – all family members inherit the same base class that

Figure 3: Nucleus Family Tree

The entire family tree shows different nucleus versions, with the root (top) being the most simple and the leaf (bottom) being the most complex instance. As complexity increases, performance drops.

The nucleus family defines a pool of functional units of more or less complexity, likewise offering lower or higher performance. Dependent on application requirements and on the actual utilization of the parallel machine, the proper nucleus version comes into play. Whether a nucleus instance is being integrated statically or dynamically is not of primary importance from the design point of view. First the complete family structure must be known and then the decision can be made to implement the family as a dynamically alterable system.

---

[3] In reality there are several base classes which represent the external view of the minimal basis. These classes are *ticket* (delivery of system–wide unique communication endpoint identifiers), *notice* (intra–team thread synchronization with empty messages), *parcel* (packet–based synchronous, system–wide inter–process communication), and *region* (segment–based asynchronous, system–wide inter–team communication). They stand for horizontal independent functional units of the nucleus.

### 5.2.1 Single–User/Single–Tasking

There are three different nucleus instances supporting single–user/single–tasking mode of operation. The two most efficient instances provide *network communication* and *thread scheduling* on a library basis. Thus, these nucleus instances are part of the address space of the user/system process. This implies that no overhead–prone address space boundaries must be crossed to invoke the nucleus.

Peace only implements synchronous interprocess communication. Concurrency then is to be modeled explicitly by the application using multiple threads of control. The threading instance (i.e., thread scheduling) is the corresponding mechanization. Because of the non–existent address space boundary, this nucleus is extremely lightweight and, thus, supports the notion of *featherweight processes*. Featherweight processes are a specialized implementation of lightweight processes. They are the purest form in Peace to represent units of execution, without consideration of any protection and security measure.

The threading instance combined with the need for kernel code separation makes nucleus calls more heavyweight. Now, traps are used to invoke the nucleus. This implies very small stub routines to marshal and unmarshal nucleus service requests similar to the remote procedure call paradigm. However, instead of passing a message over a narrow channel, a local trap is to be performed. A *featherweight remote procedure call* (i.e., Noi) is executed to activate the nucleus. Solely the gap implied by the trap interface is bridged. Kernel code separation is supported, but not memory protection. As a consequence, the passing of complex data structures between the nucleus and higher–level entities is straightforward and involves no programming of address space protection hardware.

The functional enrichment introduced by *nucleus separation* enables dynamic component replacement by a third party. Higher–level entities are physically uncoupled from nucleus code. Because each nucleus instance is an abstract data type, these entities are also logically uncoupled from nucleus data. The basic mechanism to switch between different nucleus instances on the fly is to exchange trap vector entries.

### 5.2.2 Multi–User/Single–Tasking

In a distributed memory parallel machine, multi–user mode of operation is feasible even if only a single task is mapped onto each node. The entire multi–node machine can be allocated to different users at the same time. Obviously, this does not require local ("on–board") security measures to protect the tasks from each other, but it requires to protect the network interface from unauthorized access. By direct network access the user task could be able to intrude the network and, thus, tasks of different user applications.

In order to provide a multi–user function, the nucleus must be completely isolated. Memory protection is to be introduced, leading to a new instance: *kernel isolation*. Because the nucleus is part of the kernel domain, applying memory protection to the nucleus also implies the isolation of parts of the kernel address space. Concerning nucleus separation, no additional overhead is introduced. However, the isolated nucleus address space makes the passing of complex data structures heavyweight. It mainly depends on the address space protection hardware how crucial the additional overhead really is. Anyway, the increase of nucleus functionality is encompassed by the potential of communication performance loss.

On each node, *network integrity* must be guaranteed, but not necessarily the integrity of user task address spaces. This leads to the introduction of communication firewalls between different user applications. Each user application builds a unique *communication domain*. The same holds for the set of system processes constituting the operating system. Within the same domain communication is unlimited. In order to invoke system services, application processes must communicate with system processes. Consequently, different communication domains must overlap to let communication succeed. Thus, communication security does not mean complete isolation, solely, but also controlled access.

A capability–based approach is used in PEACE for this purpose. This approach grants object access only if a thread (i.e., subject) is in the possession of that object or one of its proxies. An object must be created before it can be used. It is then the autonomous decision of the object creator to make the object globally accessible. The access domain of an object may be extended by the object creator by exporting a *proxy object* [19]. Via the proxy global (i.e., network–wide) object access then is feasible.

### 5.2.3 Single–User/Multi–Tasking

The first step towards multi–tasking support is to introduce *task scheduling*. In PEACE, a task maybe multi–threaded, which implies only lightweight scheduling. In order to schedule tasks, a second scheduling level is implemented. This level knows the *bundle* as scheduling unit, which consists of one or more threads. A single threads bundle always is executed by one processor, with non–preemptive scheduling of the threads of the same bundle. Preemptive scheduling is between bundles only, and so is shared–memory multiprocessor scheduling with the different bundles being executed by different processors. A task then may consist of several bundles to take advantage of preemption and of the shared–memory processor architecture. The result is a slightly more expensive scheduler.

At this stage, multi–tasking can be supported even if *task isolation* by means of private address spaces is not provided. A private address space serves for two basic purposes. On the one hand it implements memory protection, isolating programs from each other. On the other hand it defines a logical address space for program execution, enabling code/data relocation at runtime. Being relocatable is also a property of *position independent* code, which then needs to be generated by a compiler. In addition, the use of secure programming languages supports program isolation without the necessity of address space protection hardware. Therefore, the minimal basis to support multi–tasking is task scheduling. Task isolation is the minimal extension of task scheduling. It is used to generally improve system availability and in those cases where neither the programming language nor the compiler supports the nucleus.

### 5.2.4 Multi–User/Multi–Tasking

The fourth operation mode being supported by the nucleus family is the natural consequence of the two modes discussed before. There is little more of functionality to add. Global multi–user mode of operation is made feasible by enforcing network integrity, whilst local multi–user function is directly supported by task isolation. The nucleus then provides general *security* measures, with completely isolating different (user/system) domains from each other.

# 6 Adaptive Operating System Architecture

The operating system building block of PEACE is mainly represented by POSE, which implements a family of parallel operating systems. POSE services are application–oriented extensions of the PEACE minimal basis, i.e., of the nucleus and the kernel. These services are provided by teams of lightweight processes and, usually, are executed in non–privileged user mode. Since the representation of the functional hierarchy of PEACE enables an almost arbitrarily decentralization of the building blocks, this does not enforce a microkernel approach and, thus, the need for multi–tasking on a single node.

## 6.1 Active Objects

Distributed memory architectures at least call for an object–based system design. In POSE, system services are represented by active objects, i.e., teams of lightweight processes implement system functions such as process management or file handling. Consequently, requesting the execution of a system service requires to send a message to some process. A typical client–server relation is established. POSE then consists of a multitude of cooperating teams distributed over the nodes. These teams are called *manager*.

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

Following the team structuring approach, the notion of a system call (service invocation) is slightly different from the traditional viewpoint of a trap. A system call must be requested by means of message passing, distinguishing between local and remote operation. In order to hide all these properties from both the service user (client) and the service provider (server), a PEACE system call in general takes the form of *remote object invocation* [19].

Figure 4: System Access

The porter takes the form of a library; it is part of the team address space of the service–requesting process. Dependent on the type of service, the porter may also encapsule private threads. For example, using porter threads enables service–related exception handling on a message–passing basis.

### 6.3 Third Party Configuration

Above all, a parallel operating system must be designed such that the amount of system software which is to be executed by each node can be reduced to an absolute minimum; otherwise, system bootstrapping becomes a nightmare. For this reason, POSE 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 encapsules device dependent functions and, thus, must reside in a node that has a disk attached. It is site–dependent, whereas the file manager, which uses the disk manager, may reside elsewhere and is considered as site–independent. Another example of a site–dependent manager is the kernel team. If dynamic process management is required on a node, a kernel must be present on that node to construct/destruct process objects. A process manager, however, is site–independent. It may reside on any other node and may also be responsible for the management of several nodes.

The property of being configurable is absolutely necessary to meet the needs for massively parallel systems. Except in the case of site–dependent managers, a *third party* is able to establish PEACE (i.e., POSE) configurations based on the individual needs of parallel/distributed applications. The configuration decision then will be made with respect to either performance, protection, or hardware availability.

## 6.4 Incremental Loading

The basic idea in PEACE is to perform *on–demand loading* of system services [23]. That is, system services are only loaded at the time when they are really needed. 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 ROI layer.

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. [4] Entity (or server) faults are propagated to a system team called *plumber*. Basically, this means that, once having determined that the entity is not yet available, a stub routine requests entity loading by instructing the plumber accordingly (Figure 5). The stub passes the load request to the plumber which then takes charge of all activities related to the loading of the specified entity. Note, the 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, on behalf of the porter the thread is blocked on the plumber, waiting for loading to be completed.

The plumber 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. In case of site–dependent managers, the node

---

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

Figure 5: Entity Fault Handling

addresses are stored with the entity name. A distinction between the single–tasking or multi–tasking mode of operation for the entity is also made.

In Peace, the minimal basis for dynamic restructuring requires no complex memory management functions. A maxim was that even with a single–tasking nucleus instance, which is not based on address space protection and, therefore, encompasses no memory management functions, dynamic restructuring of the node of that nucleus must be feasible yet. This node, e.g., must be given multi–tasking capability by exchanging the kernel and then allocating tasks. If the Peace kernel comes up, and so the nucleus, it always assumes non–protected address spaces. The capability to protect address spaces is the kernel taught by the *memory manager*, a side–independent system team which is loaded on demand.

## 7 Related Works

The Peace approach goes beyond that what is presently intended by state–of–the–art microkernel designs, it defines a microkernel family. In systems such as Mach [28] and Chorus [22], the microkernel is a fairly complex component, used to support the implementation of operating system services and the processing of distributed applications. As in Peace, a Chorus operating system is considered as a member of a family of functional units, with a unit being represented by a (multi–threaded) system server process, i.e., an active object. Peace also applies the family concept to structure the kernel and not only an operating system. This results in a (multi–threaded) kernel implementation with a distinguished component, the nucleus, providing a common process execution and communication environment. The Chorus microkernel (also termed nucleus) is the only choice applications have. In PEACE, the nucleus family presents an assortment of up to eight different members.

*Ra* [2] is a minimal kernel for the Clouds distributed operating system [8]. The *Ra* kernel is designed to support the implementation of large scalable object–based systems. *Ra* is a fairly complex minimal kernel too, implementing segment–based virtual memory management and short term scheduling. At best, *Ra* can be compared to the PEACE nucleus instance that provides task isolation, which is one of the most complexest nucleus family members at all.

Clouds distinguishes between objects and threads, i.e., it is structured by passive objects. The rationale for this approach is to avoid performance penalties caused by the virtually more complex code of multi–threaded server implementations. That multi–threaded server are more complex is only true for completely hand–coded implementations, but not for implementations that are supported by a class–based stub generator as in PEACE [19]. Any way, reducing server code complexity by downward migration of functions into the minimal kernel as followed with *Ra* is not the ultima ratio. It makes the minimal basis more complex and, thus, more overhead–prone.

The system which comes very close to PEACE is *Choices* [6]. Many ideas found in *Choices* are present in PEACE, and vice versa. This is because both systems share the same fundamental, classic idea of a family of operating systems [11]. They extend this idea into object–oriented, distributed/parallel environments. As *Choices*, PEACE is a class–hierarchical system. By means of the nucleus family, PEACE further distinguishes between a number of operation modes a node of a massively parallel system is exposed to. It is exactly this feature which becomes more and more important for forthcoming parallel operating systems.

Dynamic restructuring in PEACE is related to active and passive objects. Introducing active objects is straightforward and based on services to create and destroy teams of lightweight processes. Exchanging passive objects is limited to the nucleus. This is in contrast to Clouds, e.g., where arbitrary passive objects may be dynamically introduced. For this purpose Clouds relies on the segment–based virtual memory management service of the *Ra* kernel. These constraints are not given with PEACE in general. There are some PEACE family members implementing segment–based virtual memory management; there are others not being dependent on the presence of address space protection hardware and supporting dynamic restructuring yet.

## 8 Conclusion

The paper described rationale and concepts for the design of scalable operating systems for massively parallel systems. The program family concept combines a number of solutions to different application requirements. This concept promotes not only customized operating systems from the application point of view (top–down customization), but also from the hardware architecture point of view (bottom–up customization).

A distinction between operating system family and a nucleus family must be made to meet the performance requirements of forthcoming massively parallel systems. In the former case, the family is built by a number of site–independent functional units representing typical operating system services. In the latter case, a platform for both kernel construction and application processing is provided. A member of the nucleus family must be an abstract data type to allow a number of different implementations to coexist. The nucleus family takes the form of an assembly camp, but not the single nucleus implementation. From this assembly camp the proper solution is selected to optimally support a given application. This way, the PEACE approach provides a scalable, i.e., WYNIWYG-architecture (*What You Need Is What You Get*) for both the kernel and the operating system. A single solution always is a bad compromise if utmost highest communication performance must be guaranteed and a large spectrum of applications must be supported.

Approaching the family concept as exemplified with PEACE makes microkernels appear as extensions to a minimal basis. That is, PEACE provides a framework not only to build upward scalable but also downward scalable kernel architectures, an important property of parallel operating systems. The microkernel as being understood to date is merely a member of the PEACE family. To keep things right in mind: the functionality of state–of–the–art microkernel architectures facilitate scalability but at the same time forms an essential scalability handicap in case of unnecessary functionality is provided. Thus, the PEACE family design bridges the gap between distributed systems and massively parallel systems which are based on distributed memory architectures.

The family is designed, constructed and implemented following the paradigm of object orientation [7]. Classes implement system features and inheritance (i.e., subclassing) is used to derive new features or specializations of existing ones. First experiences with objective PEACE show that object orientation is superior to non–object oriented approaches. This is true for aspects such as maintainability, extensibility and performance of the resulting operating system. It is indeed a myth that object orientation makes the implementation of very high–performance operating systems impossible. Rather, it is true that object orientation is the only chance to build high–performance systems while maintaining a clean and evolutionary system structure.

The object–oriented paradigm in design and implementation of a distributed/parallel operating systems is widely accepted but, with the exception of a few operating systems, e.g., Choices and PEACE, not applied in correspondence to Wegners definition [27]. A general problem for commercial systems like Mach or Chorus is how to organize a complete redesign of their operating system. There are plans going into this direction and which shows that the system's investigators are encouraged of the object–oriented paradigm. Unfortunately, as pointed out by *Neurath*, they can't enjoy all the opportunities object–orientation offers because of their market constraints.

# References

[1] T. E. Anderson, B. N. Bershad, E. D. Lazowska, H. M. Levy, "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism", *ACM Operating Systems Review*, 25, 5, pp. 95-109, Proceedings of the Thirteenth ACM Symposium on Operating System Principles, Pacific Grove, CA, 1991

[2] J. M. Bernabeu Auban, P. W. Hutto, M. Yousef, A. Khalidi, M. Ahamad, W. F. Appelbe, P. Dasgupta, R. J. LeBlanc, U. Ramachandram, "The Architecture of Ra: A Kernel for Clouds", Georgia Institute of Technology, Technical Report GIT-ICS-88/25, 1988

[3] P. M. Behr, W. K. Giloi, W. Schröder, "Synchronous versus Asynchronous Communication in High Performance Multicomputer Systems", IFIP Working Conference 5, Stanford, August 22-26, 1988

[4] R. Berg, J. Cordsen, J. Heuer, J. Nolte, B. Oestmann, M. Sander, H. Schmidt, F. Schön, W. Schröder-Preikschat "The PEACE Family of Distributed operating Systems", Technical Report, GMD FIRST, Berlin, 1991

[5] G. S. Blair, J. J. Gallagher, J. Malik, "Genericity vs Inheritance vs Delegation vs Conformance vs...", *Journal of Object-Oriented Programming*, Vol. 2, No. 3, pp. 11-17, 1989

[6] R. Campbell, G. Johnston, V. Russo, "Choices (Class Hierarchical Open Interface for Custom Embedded Systems)", *ACM Operating Systems Review*, 21, 3, pp. 9-17, 1987

[7] J. Cordsen, W. Schröder-Preikschat, "Object-Oriented Operating Systems Design and the Revival of Program Families", Proceedings of the Second International Workshop on Object Orientation in Operating Systems, IEEE 91TH0392-1, pp. 24-28, Palo Alto, CA, October 17-18, 1991

[8] P. Dasgupta, R. J. LeBlanc Jr., W. F. Appelbe, "The Clouds Distributed Operating System: Functional Description, Implementation Details and Related Work", Proceedings of the 8th International Conference on Distributed Computer Systems, pp. 2-9, IEEE, San Jose, CA (USA), June, 1988

[9] M. Gien, "Micro-kernel Architecture - Key to Modern Operating System Design", Technical Report CS/TR-90-42.1, Chorus systemes, Paris, 1990

[10] W. K. Giloi, W. Schröder-Preikschat, "Programming Models for Massively Parallel Systems", International Symposium on New Information Technologies 91, Tokyo, Japan, 1991

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

[12] D. C. Halbert, P. D. O'Brien, "Using Types and Inheritance in Object-Oriented Languages", *IEEE Software*, 9, 71-79, 1987

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

[14] J. Lions, "UNIX Operating System Source Code Level Six", Department of Computer Science, The University of New South Wales, Second Printing, 1977

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

[16] H. Mierendorff, "Bounds on the Startup Time for the GENESIS Node", ESPRIT Project No. 2447, Technical Report, GMD F2.G1, Bonn, 1989

[17] J. K. Millen, "Security Kernel Validation in Practice", *Comm. ACM*, 19, 5, 243-250, 1976

[18] B. J. Nelson, "Remote Procedure Call", Carnegie-Mellon University, Report CMU-81-119, 1982

[19] J. Nolte, "Language Level Support for Remote Object Invocation", Technical Report, GMD FIRST, Berlin, Germany, 1991

[20] OSF, "Microkernel Program - Background and RI Goals", *Research Institute Notes*, Volume 1, Issue 2, 1990

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

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

[23] H. Schmidt, "Making PEACE a Dynamic Alterable System", Proceedings of the 2nd European Distributed Memory Computing Conference, Munich, Germany, April 22-24, 1991

[24] W. Schröder, "A Distributed Process Execution and Communication Environment for High-Performance Application Systems", *Lecture Notes in Computer Science*, Vol. 309 (1988), 162-188, Springer-Verlag, Proceedings of the International Workshop on "Experiences with Distributed Systems", Kaiserslautern (West Germany), September 28-30, 1987

[25] W. Schröder-Preikschat, "Overcoming the Startup Time Problem in Distributed Memory Architectures", Proceedings of the 24th Hawaii International Conference on System Sciences, Vol. 1 , 551-559, 1991

[26] B. Stroustrup, "The C++ Programming Language", Addison-Wesley Publishing Company, 1986

[27] P. Wegner, "Classification in Object-Oriented Systems", SIGPLAN Notices, Vol. 21, No. 10, pp. 173-182, 1986

[28] 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, pp. 63-76, Proceedings of the Eleventh ACM Symposium on Operating System Principles, Austin, Texas, 1987