Using Naming Strategies to Make Massively Parallel Systems Work

HENNING SCHMIDT

German National Research Center for Computer Science, GMD FIRST at the Technical University of Berlin, Berlin, Germany

ABSTRACT

To handle massively parallel systems and make them usable, an adaptive, application-oriented operating system is required. The application orientedness is represented by the family concept of parallel operating systems. Incremental loading of operating system services supports the family character by automatically extending the system's active object structure when it is necessary. This way, also the switching between different operating system family members may be realized. A new active object will be incrementally loaded if its invocation fails because it does not yet exist. This is noticed during object binding while using the naming services. The use of the naming system is exploited and extended to get a flexible and configurable mechanism for triggering incremental loading. This mechanism is built by the freely definable naming strategies and exceptions that result again in a family, namely a family of naming services.

1 INTRODUCTION

The possible number of processing nodes in massively parallel systems with a distributed memory architecture increases continuously. Whereas in the future millions of nodes will become feasible [1], today a number of at least several thousands is already realistic. The more complex a parallel system becomes, the more it will be difficult to survey and to use. The first hurdle for usability is the bootstrapping procedure. Supplying several thousand nodes with a complete operating system could last a disproportionately long time. That is contrary to the purpose of both the parallel computer and the operating system, namely to serve the application and to make the computational power available.

Both levels, the parallel machine and the operating system, offer chances to solve the problem and to decrease time required for bootstrapping. In both cases the solution rests on employing less. There is no general parallel application with general claims to the number of processing nodes and to the extent of operating system services. Thereby it is possible to make a virtue of necessity. The necessity is to supply lots of nodes with an operating system and the virtue is the perception that it is not necessary to do so. Especially during startup of the application neither all nodes are needed nor all operating system services. Therefore, the approach is to start only with few operating system services at a few or at only one node. Further nodes then are *incrementally bootstrapped* on demand when they are needed. This way, the application and the parallel machine are dynamically scaled up (and down) to the requested level or to the largest possible level.

Received April 1994
Revised May 1994

Scientific Programming, Vol. 3, pp. 289–300 (1994)
CCC 1058-9244/94/040289-12

Accordingly, only a *minimal basis* of operating system services is put onto the bootstrapped nodes. *Minimal extensions* then may follow on demand when they are needed. This way an adaptive, application-oriented operating system is built.

Precondition for this minimality concept [2] is a highly modular design and organization of the operating system. Otherwise, the components not needed could not be omitted.

1.1 Operating System Family

With this system organization described above each application gets its own view of the underlying operating system because only those components are existing and thus seen, which are really needed. These different appearances of the operating system build an operating system family as previously described [3]. An application profits from the family concept on the strength of the application orientedness, which is rendered possible by it. For a parallel application the operating system means first of all overhead because it reduces the possible performance and the available memory. This overhead is minimized by the family concept. The result is an optimal process execution and communication environment for the application on the part of the operating system.

This work is based on the PEACE family of distributed/parallel operating systems [4]. In the development of PEACE the family ideas were extended to the demands of distributed and parallel systems. PEACE was originally designed for the SUPRENUM supercomputer [5]. According to the development of MANNA [6] a fully object-oriented design was made. Today, PEACE is a highly portable family of parallel operating systems following the minimality concept.

1.2 Dynamically Alterable System

The consequence of supplying an operating system family is to give an aid to the application programmer for selecting the optimal family member. Because the requirements of the programs usually are not predictable and the programmers must not be forced to know about the operating system structure, the best solution is to spare them that job. Therefore PEACE was made a dynamically alterable system based on the concepts of minimality. Starting with a minimal basis of system services, *incremental loading* of minimal extensions means dynamic adaptation for the operating system and optimal service extent for the application at any time. Thus, there is a close relation between the family concept and incremental loading. With dynamic incrementability the family concept gains a decisive amount of usability.

Incremental loading is the first and the main aspect of realizing a dynamically alterable system structure. Besides more complex concepts, concerning, e.g., migration, load balancing, or fault tolerance another important fact for alterability is the destroying of the objects no longer used. Operating system services that come and go, existing only as long as they are required, thus get the character of *transient objects*.

Applying consequently the ideas of minimality also includes the possibility of omitting all incremental loading features. For a user who does not need them, these features again are needless and only mean overhead and waste of resources. Therefore, the degree and behavior of incremental loading must be configurable. This configuration, the *minimality control*, must be simply applicable. It is, e.g., unacceptable to demand recompilation to switch incremental loading on and off.

1.3 Overview

In the next section object invocation and localization are presented, because in the object-oriented PEACE system objects are the items of incremental loading. Section 3 gives a short overview of the general ideas of incremental loading in PEACE. The PEACE naming facility represents an indispensable precondition for object localization and, in addition, it offers a promising starting point for incremental loading. Therefore, the highlights of naming in PEACE are introduced. Subsequently, a description of the configurable naming strategies follows, which play the leading role for realizing incremental loading. Finally, the conclusion summarizes the results obtained so far.

2 OBJECT ACCESS

This section describes the service invocation and localization model in PEACE. Because services shall be automatically loaded when they are needed, and that is when they are invoked, the service invocation must be the starting point for incremental loading. As mentioned above, incrementally loadable operating system services are encapsulated in objects. Using a service means executing an operating defined for an object. In a

parallel environment an object may reside on a remote site. Therefore, two steps are necessary. First, an object has to be identified, and second, the invocation itself takes place. That reminds us of the remote procedure call mechanism (RPC [7]), where the binding, i.e., identification of the remote communication partner, also must be done first. Indeed, it is nothing more than the transfer of RPC ideas to an object-oriented environment.

2.1 Remote Object Invocation

Each traditional operating system service manager administers two kinds of data: public data, which may or must be known commonly, and private data concerning the manager's internal affairs. In our parallel environment this duality has been taken into account with the concept of dual objects [8]. Both the private and the public part are held by the manager in a *prototype*. The public part only, taken as an extract from the prototype, is held in a *likeness* by the client that initiated the object construction. Likeness and prototype are usually physically separated. A prototype then represents the internal view of an object, whereas the likeness represents its outer appearance. For a given dual object more than one likeness may exist, but always exactly one prototype (Fig. 1).

To perform a dual object-related call, first the extract is sent to the prototype's site. For that purpose, the likeness has to contain more information than just the object's public part. At least a destination address must be available. At the prototype's site, the public and private parts will be reunited before the real object-related call is performed. This is done by a server thread called *clerk*. A clerk is a lightweight process responsible for both object creation and invocation request acceptance. Finally, a new extract is built and sent back to the caller. The remote object invocation (ROI) has been done completely.

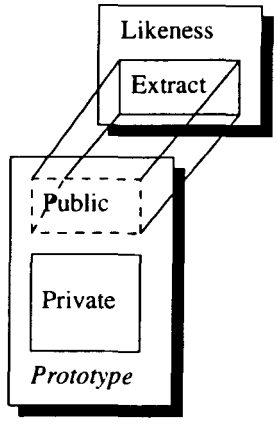

FIGURE 1 Dual object.

The likeness contains all necessary information for the ROI. The caller only has to execute an operation on the likeness. That implies that a likeness may be passed over the network, e.g., by another ROI, and stored by any thread. The operations on the likeness, now executed by the receiver thread, will still concern the original prototype and will be performed correctly.

The concepts of ROI are supported by language extensions added to standard C++ classes by means of annotations. Based on an annotated class declaration, a class generator produces the corresponding declaration of likeness and prototype that both get additional base classes responsible for the remote object invocation. The stubs for accessing the objects are also generated. See Nolte [9] for a detailed explanation of the class generator and the possible annotations.

2.2 Object Placement

Before the scenario with likeness and prototype is reached, the dual object first must be constructed. Therefore, the address of the clerk must be determined. To guarantee access transparency this is done implicitly during object construction. It makes no sense to work with fixed addresses because their values are location dependent and therefore not predictable. Instead, the clerk's address is determined dynamically. For this binding mechanism a decentralized naming service is used.

Naming in PEACE is realized by one or more name servers [10]. Each service providing thread registers a well-defined symbolic identifier in the name space and associates it with its own address (*name export*). The client queries the name space during object construction for that name (*name import*) and learns the address this way.

The names administered by a single name server are always unique, i.e., it is not possible to register a name already known. If it is necessary to replicate a clerk, a structured name space, consisting of several interconnected name servers, is needed because the exported name will be replicated too. Otherwise name clashes are the consequence. A structured name space thus introduces replication transparency. It is a little comparable

to a hierarchical file system, where each name server represents a directory. The names then either are usual entries referring to the clerks or they are special names referring to name serviers (*domain links*) and joining them to a larger structured name space.

Using ROI, name server services are invoked in the same fashion as other services. Obviously, it is not possible to achieve the identification in the same way, i.e., by means of symbolic names. Therefore, a name server is identified by using a *domain identifier* associated with each team (Fig. 2). (A team in PEACE is the unit of distribution and represents a heavyweight process, consisting of one or more lightweight process [*threads*].) Domain identifiers are managed by the kernel. Access to the domain identifier is granted by a lightweight process contained in the kernel. Using consequently the concept of dual objects, the services of this *ghost clerk* again are invoked by ROI. Because the ghost identification obviously neither can take place via the naming nor by the domain identifier, its system wide unique address contains always zeros in the location-independent parts. In this calculable ghost address the PEACE location transparency is embodied.

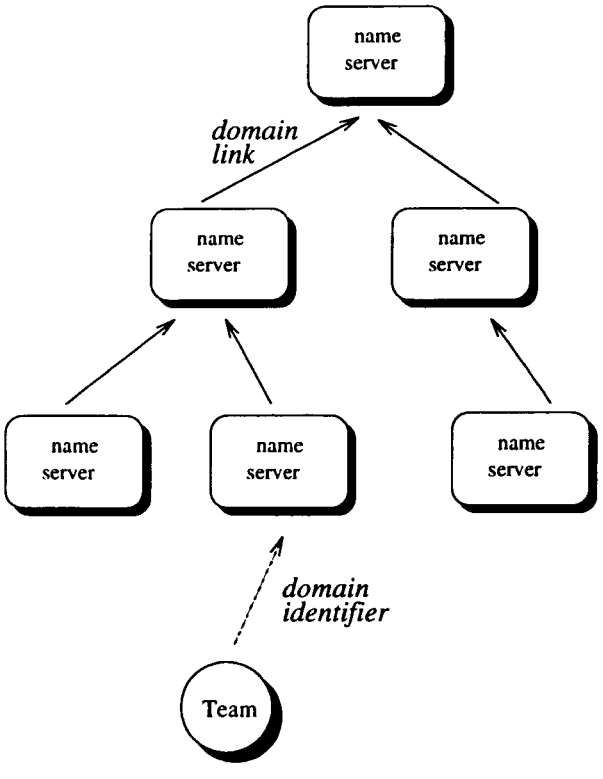

FIGURE 2 Structured name space.

Different teams may have either the same or different domain identifiers. For example, teams mapped onto one node may share a domain identifier, and thus a name server, whereas the teams at a different node share a separate name server. In this manner each process is able to identify the node relative services of its home node.

The structured name space discussed so far implements location and replication for the invocation of operating system services. In case of supporting the concurrent execution of several parallel applications, name space isolation is required. This aspect is considered in PEACE to give parallel applications the opportunity for location transparent addressing of application processes. This addressing scheme must be application oriented to avoid the potential of name clashes if different applications apply the same name with a different meaning, and to maintain security, in that intruder processes are prohibited to join the application. The application-oriented name space adds scaling transparency to the system. It consists of several name servers that belong only to one application and realize name space isolation. In addition, there may exist system servers directly administering underlying hardware components that must not be replicated for different applications. Therefore, access to a shared system name space must be guaranteed. See Schröder-Preikschat [11] for a detailed description of how this double task of name space sharing and name space isolation is performed.

3 INCREMENTAL LOADING

Bootstrapping all the hundreds or thousands of nodes in a parallel system with a complete operating system takes much too long. In addition, neither system nor application need this functionality. What is needed is that services are available at the time they are used. This is comparable to the *trap-on-use* property of Multics [12]. The classical order of presence of application and operating system is, at least partially, reversed this way. First the application or at least one application task is started and subsequently operating system components will follow automatically on demand, i.e., when they are needed. The exception from the reversal is defined by the "traphandling" routines, i.e., the minimal subset of operating system facilities needed to manage incremental loading of system services.

3.1 Server Faults

On demand loading of services will be automatically performed during service invocation if the required server does not yet exist. Generally, this is detected by the remote object invocation level of the service requesting client process in the progress of service name resolution. If name resolution fails, a *server fault* will be raised, comparable to page faults in a virtual memory system. Handling a server fault leads to the loading of the missed object.

Server fault handling is done by the *entity server* and is not limited to the incremental loading of system services. It works just as well for user application services. A server fault is propagated to the entity server by library functions belonging to the client. The library function is activated by the ROI level.

Before loading can be carried out, the entity server has to decide which object provides the expected service. An object is controlled by a team image that is stored in a file. For each load request the entity server has to select the right file. For this purpose it manages a table that maps service names onto file names. In addition to each file name several attributes may be defined like a single-/multi-tasking classification, a service export specification, and target node selection hints. In the same manner, site dependencies of objects to be loaded are laid down. The entity description available for the entity server must be user configurable. Therefore, a declarative description tool must exist that may be applied by both system and application programmer to describe their entities. Generally, with the still-increasing complexity of parallel systems, configuration and corresponding language tools become more and more important. See Section 4.4 for some remarks about configuration languages appropriate for parallel systems and Schmidt [13] for a detailed description of server fault handling.

The drawback of this approach is the raising of the server fault by the ROI run-time system. This straightforward design makes it difficult to adhere to the minimality. The code for raising the server fault would be in the ROI library, a system library, and renunciation of the incremental loading affairs is impossible. In this case, the only solution is the generation and maintenance of different ROI libraries. Configuring the user program, i.e., switching incremental loading facilities on and off, then means relinking the program with the appropriate ROI library. That is possible but unpleasant.

The same problem also arises in other proposed solutions where code shall be executed, which is not known at a certain moment in a certain address space. For example in Dorwald et al. [14] it is possible to create objects of a class that is unknown at creation time. The solution is founded on base classes. A preprocessor inserts a new declaration of the classes to be loaded containing the inheritance of the base class. It is a very dynamic solution and portable, too, but it is again difficult to configure its employment. Omitting it for a certain run of the application means recompilation. The minimality concept, which for PEACE is used to gain performance and usability of massively parallel systems, can only be met with tough effort.

This article deals with another approach to raising a server fault, also founded on base classes to be inherited. The invocation is placed one step sooner, directly into the name server, by building a family of name services. This is described in Section 4.

3.2 Name Server Faults

In PEACE name servers play a distinctive role. This concerns both identification and incremental loading. Nevertheless, the idea of a minimal subset can also be applied for name servers. Initially, it is not necessary to establish the whole structured name space by starting all the constituting name servers. Unlike usual server faults the occasion to start a name server appears in another way. Whereas a usual server will be started if any of its services is needed, a name server is loaded if its export service is required. Only if at least one name is registered by a name server, its existence will make sense. Therefore, a *name server fault* will be raised only if a server detects the absence of the name server specified as its export specification.

The *name usher*, a dedicated system server, handles name server faults by editing the name space. It is not inevitable during startup because it may be incrementally loaded in the usual way when it is needed. Name server fault handling works on the basis of a name space specification, which describes the maximal number of name servers building the name space as well as the global name server structure. Besides, the name usher knows about the actual name server structure produced by previous name server faults. For the name space specification as for the table handled by the entity server a configuration tool is

useful. Handling a name server fault means to start a new name server object and to arrange it into the actual name server structure. The existing name space is dynamically altered and restructured. For this purpose *domain links* have to be created and/or replugged. Where to arrange the new name server is determined by the name space specification.

4 NAMING SYSTEM FAMILY

In contrast to naming systems like Profile [15] or HNS [16], which work on top of a complete operating system, naming in PEACE is one of the most basic functionalities. Utilization of most PEACE operating system services is only possible with the aid of the naming scheme.

The naming service is most frequently used by the ROI level that associates object addresses with the registered names (Fig. 3). The meaning of the associated values as object addresses is only defined and interpreted by the "user", i.e., by the ROI run-time system. From the name server's view, only structureless byte strings of variable length are stored. The naming service does not know anything about object addresses.

A name server is a very fundamental and low-level system server, but it is therefore versatile and flexibly usable. This flexibility offers a good precondition for the minimality control mentioned above because it is usable at run-time. Furthermore, the naming service is used in any case for the binding during remote object invocation. It provides the first indication that a required system service is lacking. For these two reasons, the flexibility and the inevitability, the naming facility is well suited as a configurable starting point for incremental loading. Remember, it is a must that all incremental loading features are completely omitable from the system.

To bypass the difficulties of controlling the degree of incremental loading, the flexibility must be exploited in a way so that differently behaving naming services are made available without large configuration overhead. Differently behaving naming services may concern both name import and export. Different import routines, for example, could initiate incremental loading in case of a failed name search or only announce the nonexistence of the requested name or block the client until the requested name is registered. Further import behavior is possible and depends on the user's requirements. In other words, user-definable

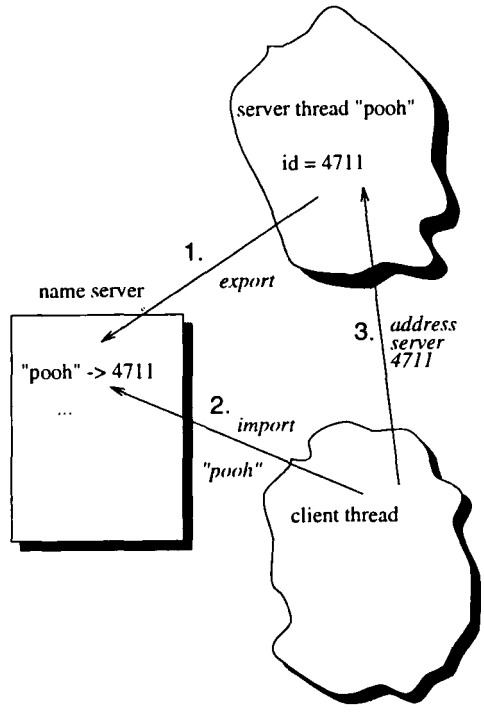

FIGURE 3 Usual name server scenario.

import behaviors are needed, supported by a set of frequently used defaults. If it is possible to flexibly define and redefine different behaviors at run-time, it will be simple to switch on and off incremental loading. In the above three import examples, the first deals with incremental loading, the second manages its duty without it, and the third may be used with or without incrementability.

The same considerations concern the name export, where different behaviors could select different name servers for the export, possibly accompanied by incremental loading of a new name server. This way, a family of naming services is built, matching the general family concepts pursued in PEACE. The following describes how the flexibility of the naming facility itself is exploited to implement the naming family that then supports the implementation of incremental loading.

A name server usually offers a set of services. Beside name import and export some others like removing and replugging a name are possible. Nevertheless, for the considerations of incremental loading only import and export are interesting. In both cases, import as well as export, the client consults the name server with a name, in the former case hoping to learn an associated value and in the latter intending to associate the name

with a value. The name server then has to check the name's existence in the name space. The result of this checking has a different meaning for import and export. If the name already exists, the import routine will be satisfied, whereas export will run into an error. Otherwise—the name does not yet exist—export will be able to fulfill its duty and import indicates this lack. These four individual cases offer different possibilities for defining individual behavior. A behavior depending on the existence of a name is classified as a *strategy*. If the name does not exist, the behavior will be called an *exception*.

4.1 Naming Strategies

A naming strategy is definable for import and export, if the delivered name is already known by the name server. Import strategies will be discussed next.

If the name server handles an import call, it first must be able to distinguish an existing regular name and a name concealing a strategy. It was always possible to distinguish regular names and names referring to another name server. For that purpose *name attributes* exist for each name. A new registered name initially gets the attribute classifying it as a regular name. The attribute then may be changed and requested by the clients. Under this viewpoint it is simple to classify a name as a strategy. If such a name is imported, the name will act as a "contact mine." Hitting that mine means that the name server will perform or initiate the activities defined by the user with that name.

Two features of PEACE are exploited to realize the mine concept. The first feature is the inheritance contained in the object orientedness. PEACE is an object oriented system implemented in C++ [17]. One characteristic of object orientation is the inheritance. A class may be derived from a base class and it inherits this way all the properties of the base class. An operation defined for the base class may be used in the derived class. In a derived class a base class operation may be redefined to give the operation a different semantic for the new class. Because a derived class object always is a base class object too, a pointer or reference to a base class object may refer to a derived class object. With that pointer it is impossible to decide which level is meant. Therefore, only the base class is seen and concerned if methods are executed. If level-specific methods shall be invoked the *virtual functions* must be used, which are bound dynamically at run-time. Handling base class objects thus always results in the execution of the correct operation even if the object is actually an instance of a derived class and the operation is redefined for the derivative.

The second fact used to realize the mine concept is the concept of dual objects. Using a dual object is transparent to the user. An operation is executed on the likeness, and from the user's viewpoint a likeness is a normal object. The remote object invocation is not seen except for a longer execution time or in the case of a node crash. This access transparency is possible because the likeness contains all necessary addressing information. Thus, what is needed in the name server is the association of the contact mine name with an object on which an import operation is defined. To realize a freely definable action, it has to take place outside the name server. Therefore, the object has to be a likeness. It is of course necessary to specify an exact interface for the import operation defined on that likeness. This interface is already given by the usual import operation implemented by the name server, which must be used. Besides, the likeness class object known by the name server cannot be freely definable. This implies the necessity of defining a general base class for the name server, from which the user definable versions have to be derived. The import operation on this base class then has to be declared virtual to realize the correct execution for different derivatives.

The whole procedure can be elucidated by regarding the most important lines of the specifications. The import routine on a name server object is defined as `int import (Stream name, Stream result);`, where class `Stream` contains a start address and a length information. This way data of any structure and length can be described and the stored name and the associated value belong to the same class. See Figure 4 for the specification of the contact mine class and Figure 5 for a possible user-defined import strategy.

```
// includes...
class impstrat {
public:
   impstrat();
   virtual ~impstrat();

   virtual int import (Stream, Stream);
   ...
};
```

FIGURE 4 Base class for import strategies.

```
// includes ...
class myimpstrat : public impstrat {
                   // inheritance
public:
   myimpstrat();
   ~myimpstrat();

   int import (Stream name, Stream result) {
      printf ("No!\n");
      return NAME_MATCH;
   }
};
```

FIGURE 5 User-defined import strategy.

To integrate this user-defined routine into the name server's affairs, a user process has to execute the following steps:

1. Declare an object of class `myimpstrat` that must be capable of being a dual object.
2. Start a `myimpstrat`-clerk and assign the object (i.e., the prototype) to the clerk.
3. Create a new likeness object from the prototype object.
4. Describe the likeness with a `Stream` object.
5. Register the name for which the strategy shall be defined in the name server and associate it with the `Stream` object containing the likeness.
6. Change the name's attribute to classify it as an import strategy.

Figure 6 shows a simplified and partially prosaic representation of the name server's activities during name import.

The name server, searching for the name, will see the associated strategy and execute `imp->import`. What then happens is controlled outside the name server. Indeed, for the execution of `imp->import` the name server will create a new thread that then performs the import with its own control flow. Otherwise the uses relation could be violated because the name server is the most basic system server in PEACE and is used by almost all other components. Usually the name server must not use any other system server. Besides, with a new thread for the external import, the name server itself immediately will be ready again for work.

```
if (!exist (name))
   return NAME_MATCH;

if (attribute (name) == NAME_ISTRAT){
   impstrat* imp = associated_value (name);
   return imp->import (name, result);
}
else {
   result = associated_value (name);
   return 0;
}
```

FIGURE 6 Name import code in the name server.

The principles for export strategies are just the same. The name server must know about a general base class containing the declaration of a virtual export operation. This base class, taken as the likeness part of a dual class, can be associated with any name and equipped with the attribute "export strategy". A user then may build customized export classes by inheriting the base class and defining the export operation in a more or less intelligent way. A self-defined export strategy will be useful if a very complex structured name space consisting of many name servers comes into play. Choosing a proper name server would be the duty of an export strategy, possible implemented by the name usher.

4.2 Naming Exceptions

Whereas a naming strategy only may be defined for a certain name, an exception is generally valid for the whole name server and cannot be associated with a name. This fortunately implies that it is only possible to define exactly one import exception and one export exception. Under these circumstances it is easy to provide the name server with an operation that can be used to define an exception.

What then follows is already well known. The definable exception is the likeness part of a dual class. On this likeness exactly one operation is executable, implementing the exception activity. The operation is declared as virtual. A user program may inherit the exception class and redefine the activity. The name server then will execute this redefined operation if an exception is defined and the requirements for an exception are present, i.e., the delivered name does not exist in that name server.

4.3 Strategies for Incremental Loading

All user-definable naming behavior is only efficient if defaulting strategies and exceptions are

```
// includes ...
class inc_load : public impstrat {
                    // inheritance
public:
   // constructor and destructor

   int import (Stream name, Stream result) {
       // load a new server, handling "name"
   }
};
```

FIGURE 7 Incremental loading strategy.

provided and implement the most useful activities. Not the least of these is the activity of incremental loading, which originally motivated the development of the individual naming behavior. Incremental loading is possible for both strategy as well as exception. An incremental loading exception generally leads to the loading of a service that is requested but not known, whereas an incremental loading strategy is only switched on for a dedicated service.

This section describes an example of how freely definable naming behaviors can be used for incremental loading. First, a class definition is needed, which inherits the class `impstrat`. The `import` method must be redefined in such a way that it will be able to load a new service object from nothing but the service name (Fig. 7).

For this purpose, some knowledge is necessary, at least a mapping from service object names to file names that designate the code implementing the service. Acquiring this knowledge is a question of configuration. See Section 4.4 for a summary of configuration in PEACE.

Subsequently the activities enumerated above must be performed:

```
1. inc_load il; // create prototype
2. inc_load_clerk ilc; // start clerk
   ilc = il; // assign prototype to it
3. Inc_load Il(il)[1];
        // make likeness from prototype
4. Stream value (&Il);
        // make Stream from likeness
5. Stream name (''pooh'');
        // make Stream from string
   Ident namer = new Ident[2];
        // create namer interface
   namer->export (name, value);
        // export string/likeness
6. namer->resign (NAME_ISTRAT);
        // declare it as strategy
```

For such a usual naming strategy as incremental loading, these activities may be executed by a system server, e.g., the entity server (see Section 3) customized with the user's requirements. Figure 8 shows the scenario up to now.

In this situation nothing will happen unless a client requires a Pooh object for which the strategy has been installed. Before a Pooh object is used, the binding via the name server takes place, i.e., `import` is performed. Now, the code in Figure 6 will be executed: The namer recognizes the strategy, starts a thread for handling the call, and the new thread will call the likeness-related import operation. This leads to an ROI handled by the `inc_load_clerk` (Fig. 9).

At the clerk's site, it will be checked by means of the delivered name "pooh" if an appropriate server is known. The Pooh server will be installed. Usually, one of the first actions of a new server is to register its name in the name space. In the case of the Pooh server, the export cannot work because the name to be registered (Pooh) is already known by the name server. There are two possibilities to avoid this name clash: First, it is possible to define an export exception that can do something convenient. Second, the import strategy concerning the incremental loading of the Pooh server can be deleted because the server is going to be started and the strategy will not be needed anymore. It is surely not ingenious to really withdraw the strategy because in this case its reinstallation would be

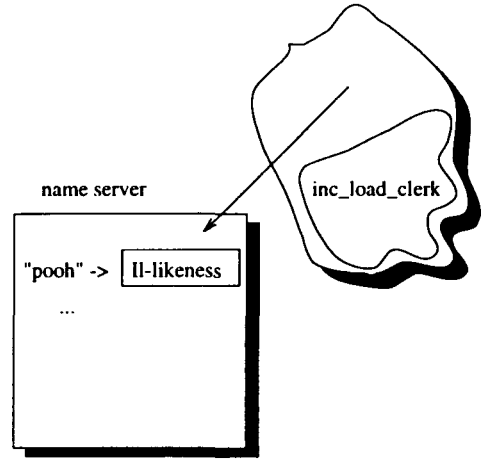

FIGURE 8 Anchoring a naming strategy.

[1] Usually the likeness class gets the same name as the corresponding prototype, except with starting capital letter.

[2] `Ident` is the ROI interface class to the name server.

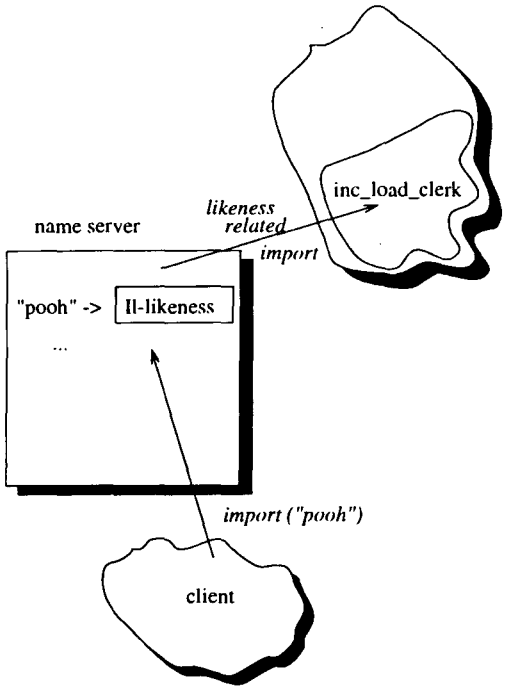

FIGURE 9 Handling a naming strategy.

avoided and the service then could not be characterized as a transient object. When a loaded server terminates because it is not needed anymore, it must be possible to reload it on the next demand. For that purpose, the strategy may be hidden in the name server. It will then be automatically reinstalled if the name with which it was associated is removed. The name removal is performed on server termination, corresponding to the initial export. Nevertheless, in this example, for simplification reasons the strategy is just deleted. Subsequently, the Pooh server comes up exporting the name Pooh, and the client retries the import and learns the address of the needed object (Fig. 10). Now the whole procedure is complete. A new object is loaded on demand at the time when it is used.

4.4 Configuration

Along with the increasing complexity of parallel systems, the need for configuration and customization increases, too. An application programmer must b able to describe the application's components as well as its needed environment. This usually results in a configuration language with which these descriptions can be expressed [18]. Special purpose configuration languages will often suffer from acceptance problems if the programmers do not want to learn a second language beside their habitual programming language.

For a native and flexible system like PEACE a second problem arises: Too much knowledge about the system and its behavior is needed at the configuration level. It is often necessary to entirely declare which components will participate at which nodes and which communication relations will exist. An application programmer using ROI without directly seeing PEACE may not know about this information. Besides, the information may vary depending on the used family member and on the actual system status. A lot of the system services may either be only a library function or may effect on or more system servers. Therefore, there must be different levels of configuration dealing with different amounts of knowledge about the system and taking the flexibility into account. For PEACE, a configuration approach based on the C++ programming language is selected. Using a C++ class library with dedicated constructors and destructors, the programmer can write a simple program in which the application is described in a declarative manner, i.e., ignoring most of the language features [19]. Because the usual programming language is used, the acceptance problems will not arise. Exploiting the inheritance feature of C++, new class libraries can be constructed, thus supporting different configu-

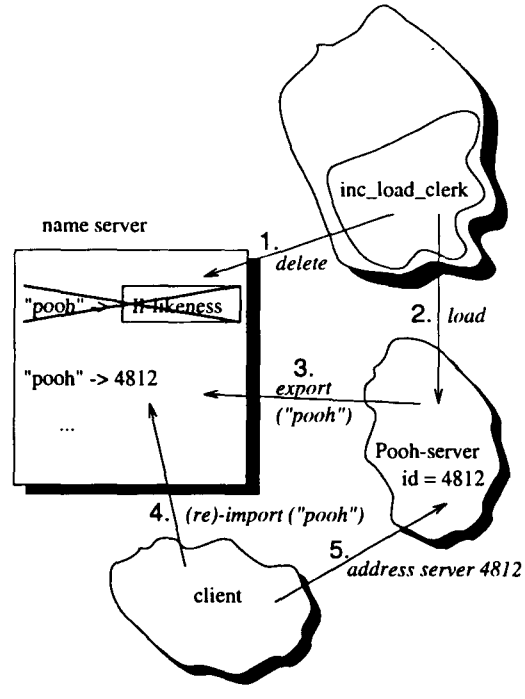

FIGURE 10 Terminating a naming strategy.

ration demands by building minimal configuration extensions. The corresponding minimal system extensions are simply matched this way. So far in PEACE very good experiences are made with this approach. All configuration descriptions ever used in the project or in earlier versions could be generated in each case by a very small program using simple library classes.

5 CONCLUSION

With user-definable naming strategies and exceptions a family of naming services is founded. It is possible to define import strategies as well as import exceptions in a way that implements or triggers incremental loading of new system services. Figure 11 summarizes the relations of the concepts presented in this article. With the freely definable naming strategies and exceptions a family of naming system is founded. The naming family may be used among other things for incremental loading, which is one of the major aspects of building a dynamically alterable system. With a dynamically alterable system structure the usability of operating system families is supported because the application optimal family member can be selected automatically and dynamically. Last but not least operating system families handle massively parallel systems.

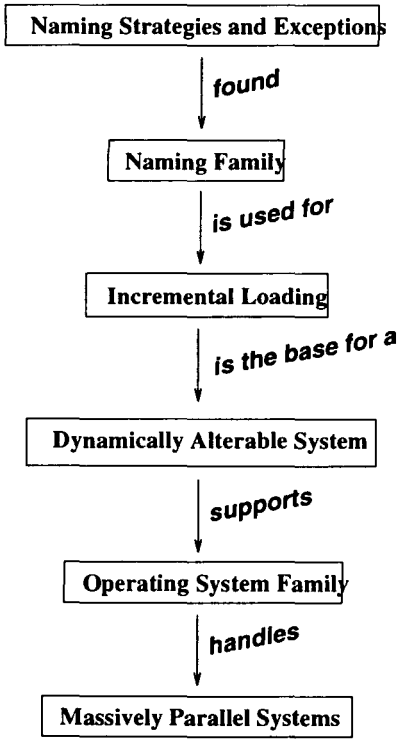

FIGURE 11 Summary.

Shortening this chain brings the perception that the naming strategies make massively parallel systems usable. This statement with its absolute predicate seems a little overacted, but the crux of the matter is surely true.

Another useful definable behavior is the blocking of an importing client until the requested name has been exported. For incremental loading it is in any case wise to calm down the client until the new server is ready to work. Even without incremental loading the calming of the importer also could be useful, especially during startup and initialization time when possibly lots of importers and exporters address themselves to the name server. Multiple unsuccessful imports before the corresponding export succeeds are avoided this way. For this behavior the import and the export exception must be defined, the former to block the importing client if the name is not yet known and the latter to unblock the client if the requested name is exported to the name server.

Along with the incrementability, mechanisms of decrementability belong to the concept of transient objects. An object that is not needed anymore should be released. At the object level it is simple to destroy an object if no reference to it exists. In C++ this is done automatically by the destructors, which are also taken into account by the dual objects. The remote destruction works as expected. At the server level it is a little more difficult to decide on a release. A server that is frequently used for a very short time could be continuously destroyed and recreated. Thus, transient objects must be configurable to control different destruction requirements. With the concepts of incrementability and decrementability PEACE becomes a dynamically alterable system.

One of the most important motivations for all these concepts was to make massively parallel systems usable. How well this goal is reached depends on the scale of the system. A small system consisting of a handful or perhaps up to some dozens of nodes may be used efficiently without incrementability. The scalability of such a system is not unlimited. For highly scalable systems and for very big systems, i.e., real massively parallel systems, the incrementability will be very useful. Today it is difficult to say from which number of nodes the usability is measurable, but it surely depends on the underlying hardware system. Factors like availability and speed of bootstrapping

devices and performance of the network will influence this number. In the future, the effectiveness of the incremental PEACE operating system will be tested and measured for the MANNA [6] system, a highly scalable parallel computer with distributed memory.

ACKNOWLEDGMENTS

This work was supported by the Ministry of Research and Technologie (BMFT) of the German Federal Government, grant ITR 9002 2.

REFERENCES

[1] Feasibility Study Committee of the Real-World Computing Program, "The master plan for the real-world computing program," *Draft*, March 2, 1992.

[2] D. L. Parnas, "Designing software for ease of extension and contraction," *IEEE Trans. Software Eng.*, Vol. SE-5, 1979.

[3] A. N. Habermann, L. Flon, and L. Cooprider, "Modularization and hierarchy in a family of operating systems," *Comm. ACM*, Vol. 19, pp. 266–272, 1976.

[4] W. Schröder-Preikschat, Ed. *PEACE—The Evolution of a Parallel Operating System.* St. Augustin, Germany, 1992.

[5] W. K. Giloi, *CONPAR 88.* Manchester, UK: Cambridge University Press, 1989, pp. 10–17.

[6] W. K. Giloi and U. Brüning, *Proceedings of the Second NEC International Symposium on Systems and Computer Architectures.* Tokyo: Nippon Electric Corp., 1991.

[7] B. J. Nelson, "Remote procedure call," *Technical Report* CMU-81-119, Carnegie Mellon University, 1982.

[8] J. Nolte and W. Schröder-Preikschat, *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences.* IEEE Computer Society Press, 1993, pp. 134–143.

[9] J. Nolte, "Language level support for remote object invocation," *Technical Report* GMD FIRST, Berlin, Germany, 1991.

[10] M. Sander, H. Schmidt, and W. Schröder-Preikschat, *International Workshop on Communication Networks and Distributed Operating Systems within the Space Environment.* 1989, pp. 293–302.

[11] W. Schröder-Preikschat, "*The Logical Design of Parallel Operating Systems.*" Englewood Cliffs, NJ: Prentice Hall (in press).

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

[13] H. Schmidt, *Proceedings of the 2nd European Distributed Memory Computing Conference.* Munich: Springer-Verlag, 1991, pp. 421–431.

[14] S. M. Dorward, R. Sethi, and J. E. Shopiro, *Proceedings of the USENIX C++ Conference.* 1990, pp. 279–292.

[15] L. L. Peterson, "The profile naming service," *ACM Trans. Comput. Systems*, Vol. 6, pp. 341–364, 1988.

[16] M. F. Schwartz, "Naming in large, heterogeneous systems," Technical Report 87-08-01, University of Washington, Seattle, 1987.

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

[18] J. Kramer, J. Magee, and M. Sloman "Constructing distributed systems in conic," Imperial College Research Report DOC 87/4, 1987.

[19] E. H. Siegel, Proceedings of the International Workshop on Object-Orientation in Operating Systems. 1993.