Modeling Replication and Placement in the PEACE Parallel Operating System - A Case for Dual Objects *

Jörg Nolte
Wolfgang Schröder-Preikschat
German National Research Center for Computer Science
GMD FIRST at the Technical University of Berlin
Hardenbergplatz 2, W-1000 Berlin 12, Germany
{nolte,schroeder-preikschat}@kmx.gmd.dbp.de

March 10, 1992

Abstract

Parallel operating systems are designed to specifically support the execution of parallel programs on parallel computer architectures. The most challenging architectures are those which are based on distributed memory. These architectures imply the modeling of both parallel programs and the parallel operating system by an ensemble of interacting objects. Interactions then take the form of remote object invocation. The paper motivates *dual objects* as a mechanism for replication and placement.

1 Introduction

Forthcoming massively parallel systems are distributed memory architectures consisting of several hundreds to thousands of autonomous processing nodes interconnected by a 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 dynamically alterable, and application-oriented. At the same time utmost highest communication performance must be provided.

The communication performance problem is dominated by the message startup time, i.e., the time which is lost for further application program processing. For example, with the scalar performance of a 40 MIPS processor in mind, a process-level message startup must be performed in the order of magnitude of 10 μsec. In this case, a balanced ratio between the per-node computing power and network communication latency is achieved [4]. It is obvious that these performance figures call for extremely *lightweight operating system structures*,

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

i.e., for dedicated parallel operating systems. The solution to these problems is an approach in which an operating system is being understood as a *family of program modules* [6] and not as a monolithic "saurian" of more or less related components.

An example of the family-based approach is PEACE [8], the parallel operating system developed for SUPRENUM [3]. PEACE started in 1986 as an object-based operating system relying on the microkernel approach. Lessons learned from this development led to the conclusion that a microkernel-based system organization is not lightweight enough for parallel architectures ([7] and [1]). The more promising approach is to combine the idea of program families with object orientation [2] and, thus, to abandon the microkernel approach. Since 1990 PEACE therefore runs through a metamorphosis, from an object-based to an object-oriented system.

The paper presents a novel object model which is used in PEACE to compose both parallel programs and operating system family members. This model introduces **dual objects** [5] as a mechanism for replication and placement of objects in massively parallel systems.

2 Transparency vs. Efficiency

A meaningful load balancing scheme must rely on some kind of object mobility. This further calls for transparency which, however, is not free of charge and may have a significant drawback on efficiency. In contrast, efficiency is a predominant issue of parallel computing. A vicious circle.

The art in parallel operating systems design, therefore, is to support transparency without loosing efficiency. A maxim must be not to punish an application by system functions which will never be used. This includes the entire spectrum of computer resources, ranging from memory space to processor cycles. An open, application-oriented operating system structure is required, with the application and not the operating system deciding which functions must be supported and which must not.

In such a context, the application should be viewed as an integrated member of the *operating system family.* Object orientation [10] then is the natural choice to realize such a family. Following this approach, an application is always considered as the final system extension - it is a final specialization of fundamental operating system abstractions.

In the realm of distributed memory architectures, this requires object orientation, i.e., inheritance to be extended into a distributed environment. In addition, *remote object invocation* (ROI) must be supported most efficiently. To speed-up processing in a parallel/distributed system, some kind of object replication is needed. Maintaining consistency of the replicated objects, however, is not the primary issue. It depends on the application and, thus, is a typical example of a minimal system extension and should therefore be introduced by subclassing.

This motivates an object model which places the focus on the separation of replicatable from non-replicatable state information of an object. The notion

of **dual objects** supports directly this approach, without prescribing consistency measures between replicated data.

3 Dual Objects

A dual object consists of two closely related but physically separated parts: the **prototype** and at least one **likeness** (Fig. 1). Conceptually, the prototype is a *mobile object*. At any given time, each allocated dual object has only one prototype, but it may have more than one likeness. The prototype logically has a *private state* and a *public state*. A likeness is built by extracting the public state, encapsulating the *extract* by a proxy object [9] and exporting the *proxy* into the context out of which allocation of the dual object was requested. In addition to the extract, the proxy carries a *system-wide unique reference* as a hint to the storing site of the prototype.

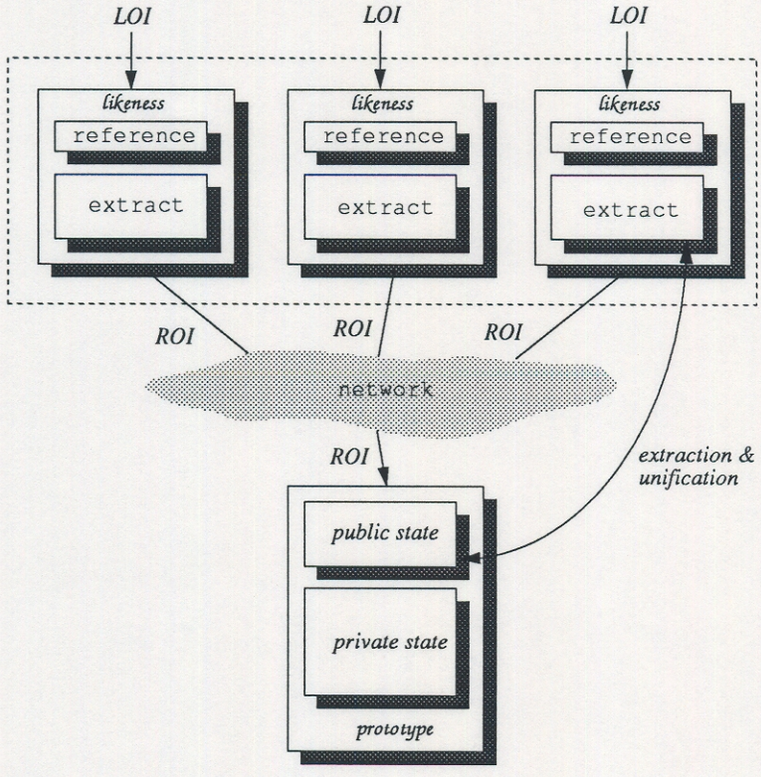

Figure 1: Prototype vs. Likeness

Operations on a likeness (i.e., the extract) are performed by standard *local object invocation* (LOI), i.e., without crossing protection domains. Operations on a prototype (i.e., the private state) are performed via ROI. Whether LOI or ROI takes place is transparent to the applying entity, except for the different execution times.

The manipulation of an extract leads to the (tolerated) inconsistency with

its original public state. When the prototype is invoked the extract is unified with the private state to re-establish consistency. Unification is being done at the storing site of the prototype. First, the public state is replaced by the extract and then the operation on the prototype is executed. When the operation terminates, a new extract is generated, replacing the extract at the storing site of the corresponding likeness. A new likeness is built and *vertical consistency* established.

The proxy is a *prototype capability*. When a likeness is drawn from its prototype the public state gets to be replicated by exporting the encapsulating proxy. It is the individual decision of the entity which receives the proxy to initiate replication of the same and, thus, granting prototype access to others.

The extract of a prototype may be empty, meaning that the prototype has a private state only. Consequently, proxy replication will not evoke possible extract inconsistencies. In this case, ROI is the only means to interact with the dual object. Only in case of a non-empty extract proxy replication becomes the reason for possible inconsistencies between the various likeness instances of the same prototype. To (re)establish *horizontal consistency* between these instances, the model motivates customization (i.e., specialization) on a per-proxy basis. If required by the parallel application, a likeness and not its prototype inherits the capability for consistency maintenance.

A dual object is specified by adding *annotations* to its C++ class. These annotations describe the intended use of the entire class, the runtime behavior of individual methods, and parameter passing semantics. They are not interpreted by a standard C++-compiler, but by the *P++ class generator* [5].

4 A Case Study

To exemplify the concepts described so far, the modeling of a neural network is considered. A neuron is a cell (a passive object) with some inputs and some outputs. It is represented as a dual object (fig. 2). The neural network thus is implemented by a distributed data structure.

```
class Neuron/*!::neuron!*/: private Tree(Neuron) {
private:
    double  threshold;                          // when to fire
    double  sum;                                // added inputs
public:
    double  weight;                             // output weight
    neuron(double thresh);                      // constructor
    void connect(/*!copy!*/Neuron*, double); // linkage
    void fire()/*!trigger!*/;                   // activate net
}/*!global!*/;
```

Figure 2: Annotated Neuron Class

In the scope of P++, the `global` annotation splits the original class de-

finition into a *likeness class* and a *prototype class*. Due to the annotation `/*!::neuron!*/`, the prototype class is referred to by `neuron`, whereas the likeness class will be named `Neuron`. The public member `weight` represents the extract of the dual object of that class.

The inherited `Tree(Neuron)` defines the outputs of a neuron. Each output stands for a neuron likeness, encapsulated by a neuron proxy. A proxy then is passed by `copy` to `connect` it as output to the applied neuron. For simplicity, inputs are not considered in this example.

Each time a neuron is fired, the `weight` will influence the `sum`. Note, since `weight` is the replicated extract (encapsulated by a proxy), it will have different values for different connections. If the `sum` exceeds the `threshold`, the neuron starts firing recursively all its outputs. Since it is not necessary to wait for the termination of `fire`, the `trigger` annotation is used to unblock the caller directly after the callee received the invocation message: no explicit response is expected.

Neuron prototypes are managed by a multitude of active objects. Each of these objects is termed *clerk*. When applying P++, a clerk is automatically generated on behalf of the `global` annotation. To construct a neural network, neuron clerks may be spread equally over a computer network. Afterwards, neurons will be placed onto the clerks using *placement constructors* (fig. 3). Note that P++ extends the original signature of a constructor with arguments, which control object placement in different ways. It also adds to the annotated class a superclass which deals with ROI. This superclass provides the feature `location()`, which returns the clerk reference of the dual object (i.e., the proxy).

```
...
Neuron *x, *y, *z;
balance* site = new balance("Neuron");  // determine clerk
...
x = new Neuron (3.2);                   // default placement
y = new Neuron (8.4, site);             // allocate at clerk
z = new Neuron (0.1, y->location());    // collocate y and z
...
x->connect (y, 1.7);                    // link x with y...
x->connect (z, 7.2);                    // ...and with z
...
x->fire ();                             // start network
...
```

Figure 3: Neural Network Construction

To achieve load balancing, the idea is not to migrate (active) clerks between the nodes, but (passive) neurons between the clerks. This approach directly reflects the needs of massively parallel systems, which require efficient mechanisms for lightweight resource management. The object paradigm supports this desire,

since addressing of objects instead of processes introduces a finer granulation for load balancing strategies and efficiently supports migration techniques.

5 Upcoming Work

Objective PEACE with dual objects is currently running as guest level implementation under SunOS 4.1. Ports onto a i860-based parallel computer, a Transputer system, and SUPRENUM are in progress.

Future work encompasses the integration of fine grain load balancing, virtually shared memory with strong and weak consistency, and persistency. Originally developed as a testbed, the guest level implementation will be extended to support parallel computing on workstation networks. The goal is to provide a single platform concept for a number of distributed memory architectures ranging from local area network systems to dedicated parallel computers.

References

[1] R. Bryant, Hung-Yang Chang, B. Rosenburg, "Experience Developing the RP3 Operating System", *Computing Systems, The Journal of USENIX Association*, Vol. 4, No. 3, pp. 183-216, 1991

[2] 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

[3] W. K. Giloi, "The SUPRENUM Architecture", CONPAR 88, Manchester, UK, Cambridge University Press, pp. 10-17, 1989

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

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

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

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

[8] W. Schröder-Preikschat, "Scalable Operating System Design", Technical Report, GMD FIRST, Berlin, Germany, 1991

[9] M. Shapiro, "Structure and encapsulation in distributed systems: the Proxy Principle", Proceedings of the 6th International Conference on Distributed Computing Systems, pp. 198–204, Cambridge, Mass. (USA), 1986

[10] P. Wegner, "Classification in Object-Oriented Systems", *SIGPLAN Notices*, 21, 10, pp. 173-182, 1986