Dual Objects – An Object Model for Distributed System Programming

Jörg Nolte, Wolfgang Schröder-Preikschat[†]

GMD FIRST
Rudower Chaussee 5
D-12489 Berlin, Germany
jon@first.gmd.de

†University of Magdeburg
Universitätsplatz 2
D-39106 Magdeburg, Germany
wosch@cs.uni-magdeburg.de

July 14, 1998

1 Introduction

When parallel processing became popular at the end of the eighties, it became evident that common operating systems were not able to deliver the pure performance of parallel hardware to parallel applications. Much processing power was wasted with complex system call mechanisms and sometimes vast resource consumptions of the operating system itself. Even micro-kernel based systems were often too slow, because these also relied on computing power consuming concepts like address space separation or virtual memory systems. Nevertheless, some applications required exactly those functionalities that others denied for performance reasons. Since this contradiction can hardly be solved within a single operating system, the PEACE operating system family[10] was developed at GMD-FIRST. The most simple family members were represented as highly efficient runtime libraries while the most complex members can be regarded as full fledged micro-kernel based operating systems.

Family based systems can be implemented conveniently by means of object oriented programming paradigms. Thus the PEACE operating system family has entirely been implemented in C++. Operating system services are implemented as classes and users can extend and specialize these system classes by means of inheritance mechanisms. In theory this scenario is sound and straight forward but in practice the conceptual advantages of object orientation are *extremely* hard to exploit without suitable object models and language-level support for object-oriented implementation techniques in distributed contexts.

When users extend and specialize PEACE classes by means of inheritance mechanisms, class hierarchies need to be extended across address spaces as well as network boundaries and objects can be fragmented across address spaces. This in turn can lead to serious performance bugs caused by frequent remote invocations, when application classes closely interact with their system-level base classes. On the other hand it is obvious that client classes cannot have full access to system-level state information to avoid forgery and ease resource sharing amongst many clients. Implementing system services as fragmented objects[7] like in the SOS system [12] would have supported independence as well as encapsulation of object fragments allocated in different address spaces. Nevertheless we considered that model already too complex for those very lightweight system structures we were aiming at, because the fragmented object model partially relied on group communication mechanisms and did not support inheritance-based fragmentation.

Having such problems in mind we designed a general object model for system programming, that transposes the classical coarse grained user/supervisor memory model of monolythic systems to very small objects of language-level granularity. So-called *dual objects* [9] implement system services and encapsulate both user-level and system-level state information in a single object context. Clients are allowed to control the user-level part of a dual object directly and efficiently within their address spaces, whereas system servers have transparent access to both parts during service invocations. Thus much closer interactions between user and system classes are possible as in conventional object models and the model remains simple enough to be implemented efficiently.

2 Dual Classes and Dual Objects

Dual objects are described by annotated C++ classes that specify user-level and system-level class members. To retain a strong backward compatibility to original C++, all `public` and `protected` members are considered to be user-level, whereas all `private` members belong to the system-level. Thus the weak language-level protection of private class members in C++ is enforced by strong encapsulation of private data in the (remote) address-space of the server. If private members shall be part of the user-level state, they need to be specifically annotated as `/*!local!*/private`.

```
class Adviser : ... {
    private:                // system-level
        Actor** actors;
        Actor*  my_actor;
        int num_actors;
    protected:              // user-level
        Ticket   cap;
        AccMode  mode;
        VSMProt  protocol;
        Addr     low;
        Addr     high;
    public:
        ...
        void setProt (VSMProt prot) /*!local!*/ { this->protocol = prot; }
        void setCap  (Ticket cap)   /*!local!*/ { this->cap = cap; }
        ...
        int  handle (int page, Addr addr, AccFault type);
        ...
}/*!dual!*/;
```

Figure 1: A Dual Class

Fig. 1 shows a much simplified example of the dual `Adviser` class that controls consistency protocols in the virtually shared memory system of PEACE[3]. Dual classes are identified using a `/*!dual!*/` annotation following the closing bracket of the class declaration. These classes are fed to a *dual object generator* (DOG) to transform them into functionally enriched C++ classes capable of dealing with the distribution aspects of dual objects. Dual classes can be composed by (even remote) inheritance mechanisms and important C++ features like multiple inheritance are retained.

The private member slots of a dual class belong to the system-level whereas all `protected` members are user-level data members. Thus the methods `setProt()` and `setCap()` can be executed directly within the client's address space as indicated by the `/*!local!*/` annotation whereas methods like `handle()` are executed remotely under the control of the server using remote object invocation techniques. During invocations the user-level members are made available for the server. Thus the `handle()` method has transparent access to the actual `cap`, `mode`, `protocol`, `high` and `low` members. The implementation of this concept causes a little additional overhead, because we need to transfer the user-level data to the server in order to grant efficient access to these members. Since this overhead is comparable to implicit parameter passing it can be neglected if the transfer costs of user-level state information is small compared to the execution time of the method.

3 Resource Sharing and Individual Customization

A process can grant access to its resources by creating clones of its dual object instances and transferring these clones to other processes. Whenever a dual object is cloned, only the user-level part is copied, whereas the system-level part is (transitively) shared amongst all clones. Thus in the `Adviser` example (fig. 1) the `private` system-level members are shared amongst all instances transitively cloned from the same origin. In contrast all `protected` user-level members are independent copies that will be manipulated independently from each other. Thus the user-level part of a dual object becomes a client-specific *context* that is implicitly provided to servers during service invocations. Consequently the `cap`, `mode`, `protocol`, `high` and `low` members (fig. 2) will always refer to the user-level part of the client actually calling a specific `Adviser` instance.

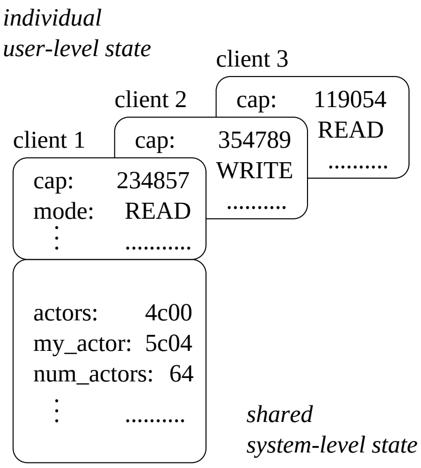

Figure 2: Resource Sharing with Dual Objects

As a result servers are not enforced to maintain client-specific data themselves and clients are able to customize specific aspects of system-services dynamically at runtime. Robustness of services is therefore enhanced and system-level data can be shared conveniently and economically amongst many clients. In fact, these resource sharing facilities are comparable to delegation based models [6] with the major difference that the sharing facilities of dual objects are statically defined through dual classes whereas common delegation schemes allow dynamic sharing.

4 Runtime Model and Implementation Issues

Dual objects have two representations at runtime, one for clients and one for servers. We call instances of the client's representations *likenesses* and instances of the server's representation *prototypes*. A likeness reflects the public interface of a dual class and consists of public and protected members only, whereas the prototype consists of all members. Furthermore, the likeness holds a (remote) reference to its prototype. Thus a likeness is both an object that can be manipulated locally as well as a proxy[11] for a remote prototype.

Prototypes are passive C++ objects that are kept in so-called *domains*[8]. Domains are a concept for local object spaces that are managed by active server objects we call *clerks*. These clerks are able to instantiate new prototypes upon request and control access to all objects within their domains. Many domains may share an address space or may have separate address spaces either on the same machine or somewhere in a network to constitute a global distributed object space (fig. 3). Furthermore, a domain may either be sequential or concurrent. Sequential domains are monitors and allow exactly one object in the domain to be manipulated at a time. Concurrent domains manage a dynamic thread pool and implement read/write monitors on single object instances.

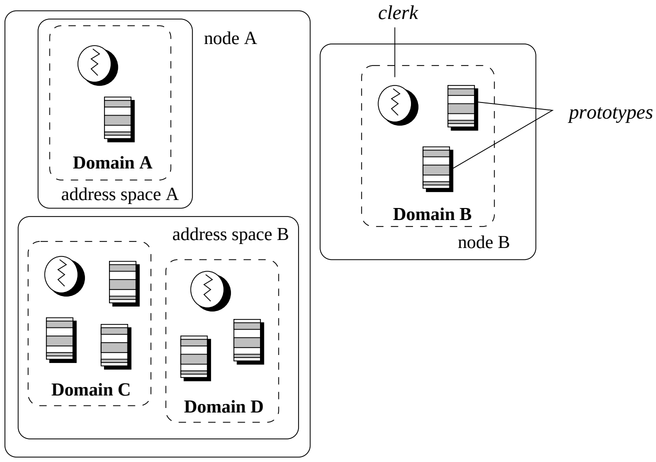

Figure 3: Domains

When a dual object is created, an instantiation request is sent to the clerk of the domain selected to host the new prototype. Domains are selected either by name contacting a name service or by a *unique identifier* denoting the communication address of the clerk controlling the domain. The clerk in turn creates the prototype and executes its constructor.

After initialization all user-level parts of the prototype are extracted and sent back to the requesting client. Here a likeness is initialized with the user-level data and the remote reference to the newly created prototype. In fact, any time a client declares a new likeness instance which is not a clone of an existing likeness, the instantiation procedure described above is transparently executed.

Methods that access user-level data only are executed locally on the likeness leaving the prototype untouched. All methods that are executed on the likeness that involve system-level data will in fact be remote object invocations on a remote prototype. Since the user-level part may have been changed by previous local calls, it is transmitted along with the arguments of the call (fig.4).

The receiving clerk then will update the prototype by means of the actual user-level data before the method is executed[1]. When the method returns the user-level part is extracted and sent back to the client along with the results of the method.

[1]This is necessary because we use standard C++ compilers as back-end. Otherwise user-level members could be referenced differently from system-level members.

The likeness in turn is updated with the actual data. This protocol causes some

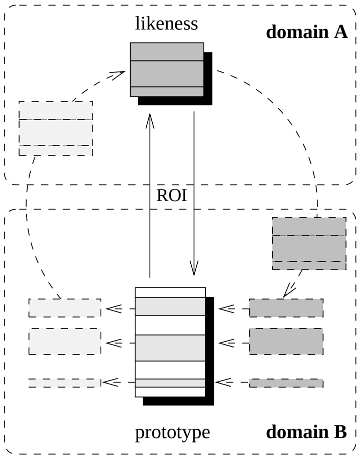

Figure 4: Remote Invocation

additional overhead for those methods which access both user-level and system-level data simultaneously. Since that overhead is comparable to implicit parameter passing it can be neglected if the transfer costs of user-level state information is small compared to the execution time of the method. Other methods are either not affected or even executed locally at the client site if no system-level data is accessed at all.

5 Performance

All performance measurements have been performed on a 16 node Manna computer. The Manna is a scalable architecture based on a 50MHz i860 processor and is interconnected via a hierarchical crossbar network with 50Mbytes/s links. Up to 16 nodes can communicate directly over a single crossbar switch. Therefore the communication latency between any two nodes is the same in the single cluster we used. Table 1 shows the basic timings for elementary communication costs as well as remote object invocation (ROI) times.

Operation	Time (μsec)
Inter Process Communication (IPC)	157
Async. IPC	18
bulk data transfer (fetch)	122
bulk data transfer (store)	28
ROI with argument size not exceeding an IPC packet	165-168
async. ROI	ca. 26
ROI with implicit bulk data transfer	308
remote object creation	266

Table 1: Performance of basic operations

The Peace nucleus provides synchronous and asynchronous packet-based mechanisms (64 bytes) for inter process communication (IPC) as well as primitives for end-to-end bulk data transfer (`fetch()`, `store()`). The ROI mechanisms are built upon these mechanisms. When the argument size (including the user-level part of a dual object) exceeds an IPC-packet, bulk data transfer primitives are automatically applied to transmit bigger messages.

The typical ROI overhead (compared to the basic commmunication mechanisms) is in the vicinity of 8μsec and drops to less than 5μsec in case of local communication that implies a better cache utilization[2]. Notably we cannot measure significant differences with varying parameter sizes up to the size of a packet. The Dog generates statically typed message formats for most methods and thus marshaling costs for simple data types and aggregate types are *extremely* low. Only array data types and aggregate data types that contain arrays imply some copying overhead (as compared to direct bulk data transfers) caused by copying loops in the stubs.

The overhead caused by the bidirectional update protocol needed to keep the user-level state consistent (refer to section 4) is basically the same as the overhead of passing value-result parameters of the same size. If user-level state and the arguments of a method fit into a single packet the update protocol is a pure piggyback protocol that only adds very small copying overhead to the basic ROI protocols.

6 Related Work

Other high-level approaches from the distributed systems area such as CORBA [4] tend to "eat up" the performance of lower layers for the sake of convenient heterogeneous computing and interoperability issues. In the high performance systems area more recent parallel C++ versions such as CC++[1], ICC++[2], C++//[13] and MPC++[5] seem to be promising. Nevertheless, system programming still needs significantly more control over runtime issues than languages designed for application level programming usually provide. MPC++ and C++// provide powerful meta-level programming facilities that could have beneficial impact on system programming in the near future.

7 Conclusion

Family based operating system services are hard to implement without suitable high-level paradigms and language-level support. Design and implementation of Peace was strongly influenced by the concept of dual objects. This object model transposes the classical coarse grained user/supervisor memory model of monolythic systems to very small objects of language-level granularity. Therefore dual objects encapsulate both user-level and system-level state information in a single object context to encourage much closer interactions between user and system classes as in conventional object models. Servers are not enforced to maintain client-specific data themselves. Robustness of services is therefore enhanced and system-level data can be shared conveniently and economically amongst many clients.

In the future we are going to extend our research in two major directions. First, in the high performance area we'll exploit the impact and *costs* of modern meta-object protocols to system design. Secondly, we are aiming at developing a portable and universal runtime executive (PURE) for deeply embedded (parallel/distributed) systems. The goal is bringing object-oriented operating system technology into a car. Today, a typical automotive environment consists of a fairly large number (e.g. 63) of 8 or 16 bit μ-controllers interconnected by some network

[2]The generated code as well as the code in the ROI runtime system is the same in both the local and the remote case.

system (e.g. CAN-bus) and equipped with a comparably very small amount of memory (e.g. 1–2 MB) for the entire system. This calls for extremely downward-scalable system solutions—and dual objects provide for the appropriate foundation.

References

[1] K. M. Chandy and C. Kesselman. CC++: A Declarative Concurrent Object-Oriented Programming Notation. In *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, 1993.

[2] A. Chien, U.S. Reddy, J.Plevyak, and J. Dolby. ICC++ – A C++ Dialect for High Performance Parallel Computing. In *Proceedings of the 2nd JSSST International Symposium on Object Technologies for Advanced Software, ISOTAS'96*, Kanazawa, Japan, March 1996. Springer.

[3] J. Cordsen, Th. Garnatz, A. Gerischer, M. D. Gubitoso, U. Haack, M. Sander, and Schröder-Preikschat. VOTE for PEACE — Implementation and Performance of a Parallel Operating System. *IEEE Concurrency*, 5(2):16–27, 1997.

[4] Object Management Group Document. The Common Object Request Broker: Architecture and Specification 2.0. Technical report, OMG.

[5] Yutaka Ishikawa, Atsushi Hori, Mitsuhisa Sato, Motohiko Matsuda, Jörg Nolte, Hiroshi Tezuka, Hiroki Konaka, Munenori Maeda, and Kazuto Kubota. Design and Implementation of Metalevel Architecture in C++ – MPC++ Approach –. In *Reflection '96*, 1996.

[6] H. Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object–Oriented Systems. In *Special Issue of SIGPLAN notices*, volume 21, pages 214–223. ACM, November 1986.

[7] Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Le Narzul, and Marc Shapiro. Structuring Distributed Applications as Fragmented Objects. Rapport de recherche 1404, Institut National de la Recherche en Informatique et Automatique, Rocquencourt (France), January 1991.

[8] O. M. Nierstrasz. Active Objects in Hybrid. In *Special Issue of SIGPLAN notices*, volume 22, pages 243–253. ACM, December 1987.

[9] J. Nolte and W. Schröder-Preikschat. An Object-Oriented Computing Surface for Distributed Memory Architectures. In *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, volume 2, pages 134–143, Maui, Hawaii, January 5–8, 1993. IEEE Computer Society Press.

[10] W. Schröder-Preikschat. *The Logical Design of Parallel Operating Systems*. Prentice Hall International, 1994. ISBN 0-13-183369-3.

[11] M. Shapiro. Structure and Encapsulation in Distributed Systems: the Proxy Principle. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 198–204, Cambridge, MA, 1986.

[12] Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Céline Valot. SOS: An object–oriented operating system – assessment and perspectives. *Computing Systems*, 2(4):287–338, December 1989.

[13] The Europa WG. EUROPA Parallel C++ Specification. Technical report, http://www.dcs.kcl.ac.uk/EUROPA, 1997.