Loadable Smart Proxies and Native-Code Shipping for CORBA

Rainer Koster and Thorsten Kramp

Distributed Systems Group, Dept. of Computer Science
University of Kaiserslautern, P.O. Box 3049, 67653 Kaiserslautern, Germany
{koster,kramp}@informatik.uni-kl.de

Abstract. Middleware platforms such as CORBA are widely considered as a promising technology path towards a universal service market. For now, however, no mechanisms are offered for dynamically integrating service-specific code (so-called *smart proxies*) at the client which is a major prerequisite for the development of generic clients that may connect to different service implementations offering different quality-of-service guarantees. In this paper, we therefore demonstrate how support for smart proxies can be integrated within CORBA by means of a native-code shipping service that only relies on the recent *objects-by-value* extension and *portable-interceptors* proposal. The feasibility of this approach is shown by a smart-proxy supported video service.

1 Introduction

Today middleware platforms already play an important role in distributed computing, shielding developers from the particularities of distribution and underlying communication protocols [2,9,10]. Emerging quality-of-service (QoS) requirements, however, can hardly be met while upholding the level of abstraction provided by RPC and remote object invocations. For tasks such as continuous media streaming, parameters such as latency and throughput must be carefully controlled in service-specific ways. Additionally, services are deployed in diverse environments with largely different resource availability. An ATM network with resource reservation capabilities, for instance, requires different QoS control than a best-effort connection via the Internet with large resource fluctuations, and mobile computing requires different functionality than LAN-based stationary computing.

Due to this variety of requirements and environments, one or few protocols built into a middleware platform will prove to be insufficient. Even worse, the middleware platform might become a stumbling block when service-specific communication mechanisms with functionality such as bandwidth reservation, feedback control, and compression are needed. Hence, in CORBA, for instance, only control and management interfaces for streams are defined [7], while application developers need to implement every protocol required in matching stream end points.

C. Linnhoff-Popien and H.-G. Hegering (Eds.): USM 2000, LNCS 1890, pp. 202–213, 2000.
© Springer-Verlag Berlin Heidelberg 2000

This particularly is a problem in view of an emerging universal service market, in which ideally a single, "generic" client should be able to connect to different instances of the same service offered from differed providers, with quality of service becoming an important means of distinction among different offers. With current platforms, however, the client has to implement all protocols of all service instances, that is, protocol independence and location transparency are lost. Furthermore, the client developer must know about the internals and low-level details of each service instance, which considerably complicates application development.

As a consequence, we have proposed *smart proxies*[1] that encapsulate any service-specific functionality required at the client side [4]. Smart proxies are loaded from the service dynamically on demand, replacing the traditional client stub while providing the same high-level service interface as if the remote server were co-located with the client. Access to QoS-supporting services then becomes as easy for the client programmer as to a conventional service; all low-level service-specific development efforts are shifted to the service developer, who implements both the server and its smart proxies using whatever protocol functionality and communications patterns are appropriate. Furthermore, transparently to the client, different implementations may be tailored for particular environments while the service interface remains constant (or at least backward compatible).

In this paper, we discuss how passing objects by value (OBV) and portable interceptors can be used to integrate the required support for smart proxies into CORBA. Particularly, a generic code-shipping service for value-type objects, instanced for C++, is presented that allows native-code implementations to be dynamically loaded. In Section 2, the notion of smart proxies is briefly introduced. Section 3 and Section 4 then show how proxy shipping can be built on OBV and how access to even native proxy code can be provided. Afterwards, in Section 5, these mechanisms are illustrated by an example application. Related work is discussed in Section 6, before Section 7 closes the paper with conclusions.

2 Smart Proxies

A *smart proxy* [4] is service-specific code at the client node providing the same interface to a possibly remote server as if the server were local. Unlike generated stubs, however, proxies can encapsulate arbitrary communication and QoS negotiation protocols and patterns for client-server interaction. Furthermore, if smart proxies can be shipped and linked dynamically, the client is free to choose an appropriate service implementation independently of the underlying protocols. During binding establishment, the server then decides which proxy is actually to be installed at the client, taking resource availability both within the client and network environment into account. Then, proxy and server may conduct further

[1] Note that these smart proxies differ from those offered by Orbix and TAO as discussed in Section 6.

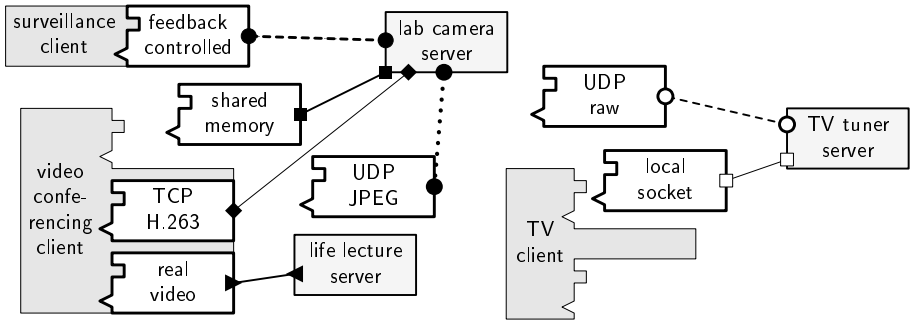

Fig. 1. Exemplified Smart Proxy Usage

QoS negotiation as needed by the application. Moreover with dynamically shipped proxies, an improved communications protocol of an upgraded service can be deployed transparently to the client.

Consider a video service with a standardised control interface including parameter settings such as resolution and frame rate. As shown in Fig. 1, a variety of service implementations may be appropriate depending on the operating environment. If some resource reservation is supported, for instance, the smart proxy can map high-level QoS parameters to low-level reservations in terms of network bandwidth; with best-effort communication only, in contrast, the smart proxy and its server may employ a feedback mechanism to achieve dynamic QoS adaptation. And if bandwidth is scarce, messages can be compressed, while for co-located clients and servers more efficient shared-memory communication might be used. All this functionality, however, is hidden from the client application which is an important prerequisite for a universal service market. Only then, service providers can develop their own sophisticated implementations according to standardised service interfaces that can be dynamically installed at and readily used by any client.

Dynamic loading of smart proxies, of course, effectively requires code shipping from the server to the client, which is significantly simplified if a virtual machine as in Java, for instance, is used for running smart proxies. Proxies then need only to be implemented for the virtual machine instead of for each possible platform in a heterogeneous environment. However, specifically for QoS-constrained services, smart proxies may need to provide good timing predictability and high computational performance and, thus, a virtual-machine approach would be too restrictive, at least for the time being. Yet for platforms that allow dynamic linking of shared libraries at run time, native-code proxies can be built as shared libraries. This approach requires a different proxy version for different client environments and possibly different client languages. Many language implementations, however, provide mechanisms to access libraries in the platform's standard object-code format such as ELF or COFF and, hence, could share the same library code. Developing proxies for each anticipated client environment, however,

is still far less effort than re-implementing the same functionality without smart proxies for each client application individually; particularly, since in the latter case not only the service developer must know about the low-level details of the service but also each client developer.

Of course, code shipping in general causes security risks, and with native-code proxies protection is even more difficult than with using a virtual machine. While this problem requires future research, for now, risks may be mitigated by using smart proxies only inside security domains or loading them only with a crytographical signature from trusted servers.

3 CORBA Objects by Value

In contrast to regular, possibly remote, CORBA objects, *value-type objects* as defined in CORBA 2.3 [9] are always local and invoked without ORB mediation. Value-type objects are copied if passed as parameters to regular objects, and the specification of OBV defines how their state is to be marshalled, transmitted, and unmarshalled.

While smart proxies can be implemented as value-type objects, ideally it should be transparent to the client whether it communicates with a value-type proxy object, or directly with a server if, for instance, no appropriate proxy implementation is available or proxies just are not useful in a particular situation. This transparency, however, is not possible if we need to pass the service object as regular CORBA object declared with `interface` and the smart proxy as a `valuetype`. For this case, the OBV specification defines `abstract interface` types which cannot be instantiated but be used as base class for regular interfaces as well as for value types. If an abstract-interface type is used as a formal parameter either an IOR or a value-type object can be passed depending on the actual argument.

```
abstract interface Service {
  // service operations
};
interface Server: Service {};
valuetype Proxy supports Service {};
interface ServerFactory {
  Service bind( ... );
};
```

The IDL above shows how `abstract interface` can be utilised for a proxy-supported service. The actual server as well as the smart proxies implement an abstract service interface and either a proxy or an IOR to the server is passed at binding establishment.

OBV, however, is *not* about shipping the object code in the first place. Only as an optional feature an IOR of a `CodeBase` object might be included in the messages which can be queried for URLs of object implementations. While this extension seems to be useful for and motivated by Java only, implementations

of value-type objects in other languages must be present at client and server at compile time.

The OBV extension of the CORBA object model, however, still can serve as a basis for the integration of smart proxies. If we are content with distributing proxy code at compile time, we can simply send proxies as value-type objects during binding establishment. Configuration parameters can be set by the server and are transmitted as part of the proxy state. An approach to add dynamic shipping of proxy code is discussed in the next section.

4 Native-Code Smart Proxies

In this section, we describe how C++ smart proxies can be integrated in CORBA by means of a general code-shipping service for value-type objects. This way, implementation code can be shipped as transparently to the client application as possible while the server provides some information where the proxy code can be accessed.

4.1 Run-Time Linking of Native Code

Even though most languages rely on a standard object-code format (e. g., ELF on Linux) and most compilers provide some support for calls to system libraries, binding proxies written in a different language than the client application generally remains language-specific. In this case, wrappers may be needed to deal with different calling conventions and argument types, but could be generated automatically from the IDL. For now, we have implemented an infrastructure for one language only. We have chosen C++, because on Unix-platforms C and C++ are commonly used and CORBA programming in plain C is somewhat more laborious.

The basic prerequisite for dynamically loading native code is the ability to link code at run time. On Unix platforms, for example, this functionality is available by calling the dynamic-linking loader directly via a programming interface for opening a shared library file and manually resolving all symbols required. This way, pointers to C functions provided by the library can be retrieved. Accessing a C++ implementation of a proxy class is slightly more difficult, because calls to member functions require the `this` object pointer. Hence, we add a plain C factory function to each proxy library which then is the only symbol that needs to resolved by the linker. These factories return actual C++ objects implemented by the library for which only an abstract base interface must be available.

```
// In the library:
   typedef CORBA::ValueBase* (*generator)();
   extern "C" {
     CORBA::ValueBase* generate() {
       return new Proxy_impl;
     }
   }
```

```
// Library access:
   handle = dlopen(code,RTLD_LAZY);
   ptr = (generator)dlsym(handle,"generate");
   proxy = (*ptr)();
```

4.2 Integration with OBV

As introduced in Section 3, smart proxies should be transparent to the client. That is, an invocation such as

```
service = service_factory->bind(...);
```

could simply return an IOR to a remote service-object, or a local smart-proxy object by value. For value-type objects the ORB demarshals the object state into a newly created instance allocated by an application-provided factory. These *value-type factories* as well as the object implementation are linked to the client application; the former are registered with the ORB during start up. If no factory is registered for some value type, the ORB raises an exception. For smart proxies, this means that their implementation and an appropriate factory must be available at the client *prior to* receiving a proxy value-type object.

An explicit two-way binding process such as

```
orb->register_value_factory(id,
               service_factory->bind_factory(...));
service = service_factory->bind_service(...);
```

could make sure that both are present at the client (see Fig. 2a), yet is not transparent to the client developer.

Alternatively, instead of returning a pointer to the smart-proxy object, a generic helper object per service interface could be introduced in combination with a custom marshaler, as shown in Fig. 2b. In this case, the factory allocates a helper object, while the custom marshaler dynamically loads the required smart-proxy implementation if not already available at the client. Afterwards, the custom marshaler initialises the helper object to delegate service invocations to the actual proxy. Note that the custom marshaler, the helper object, and its factory could be automatically generated from the IDL specification. This way, code shipping becomes transparent for the client at the cost of an additional indirection for each method invocation.

This indirection can be avoided, if the invocation path is intercepted *before* the ORB invokes the value-type factory and, thus, an appropriate factory can registered just in time. While custom marshallers are called only *after* the ORB has requested a new instance from the appropriate factory, the portable-interceptors specification provides a hook for functions that are called between a reply is received by the ORB and the call of any value-type factory relevant to that reply. At that time the missing smart-proxy implementation and its factory can be dynamically loaded and registered as shown in Fig. 2c and discussed in the following section.

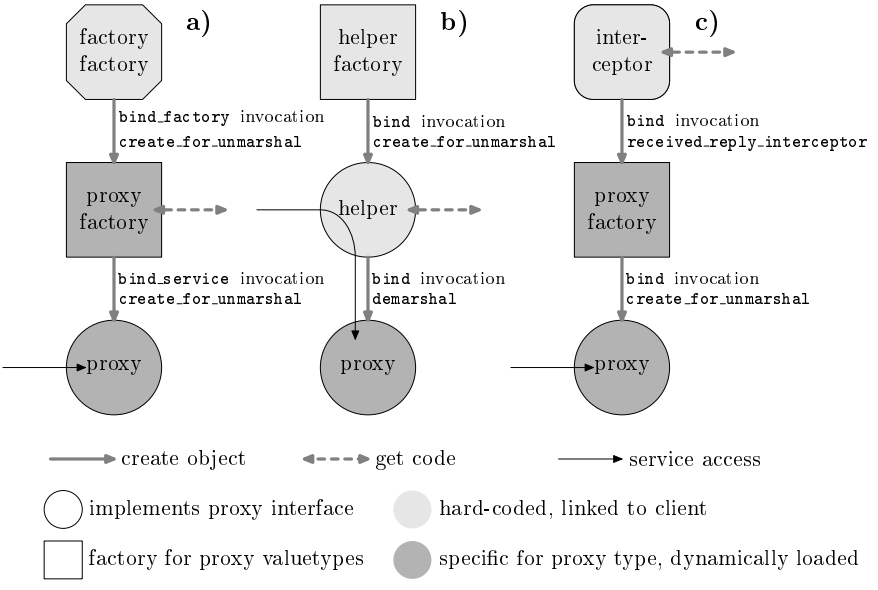

Fig. 2. Design Alternatives for Proxies with OBV

4.3 A Proxy Code-Shipping Service with Portable Interceptors

We have used the *portable interceptors* as proposed to the OMG [8] to build a service that supports downloading of native code. While we have built this service to enable smart proxies, it can be used to obtain implementation code of value-type objects in general. There are three components: Two of them implement interceptors on the client and the server side, repectively, and a third one acts as a code repository.

As illustrated in Fig. 3, the `receive_reply` interceptor of the client checks whether a proxy is being received. If this happens and the corresponding implementation code is not yet available, the interceptor downloads the code from a repository possibly launched by the server on demand, dynamically links the code with `dlopen`, creates a value-type factory, and registers it for that type of proxy; for this, each proxy implementation must be of a different type which is derived from the abstract service interface. Subsequently, the ORB can create the proxy and unmarshal its state. All this is transparent to the client application aside from initialising its shipping-service component.

Two pieces of information are needed by the interceptor to provide this functionality: the repository id of the proxy and the address of an appropriate code repository. While the former is part of the reply, the latter somehow has to be communicated by the server. Access to the return value and parameters, however, is only an optional feature of portable interceptors and, hence, there does not seem to be a portable way of checking for relevant value-type arguments at

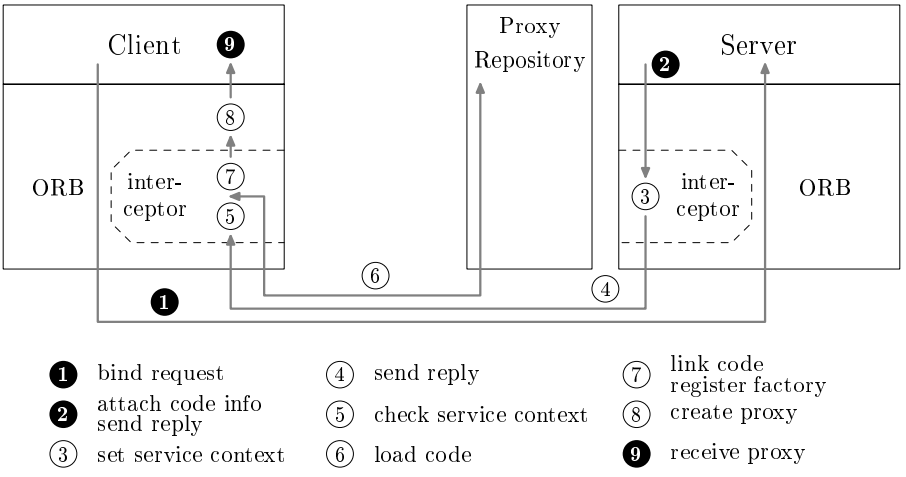

Fig. 3. Binding and Smart-Proxy Shipping

run time. Because of these problems, we send the repository id of the proxies as well as the IOR of the code repository from the server. This information is attached to the reply in a *service context* which is set by a server-side interceptor and read by its client-side correspondent.

For the server application it would be most comfortable if it could simply register code repositories for the value-type objects it may send with the code-shipping service that manages the interceptors. However, also server-side interceptors do not necessarily have access to the parameters of a reply and, thus, cannot check what value-type objects are being sent. Consequently, the server application has to explicitly provide this information for each relevant reply via an `attach` method. That is, before returning a proxy object, the servant notifies the code-shipping service of the proxy's repository id and the code repository's IOR. This information is transferred to the `send_reply` interceptor via the `PICurrent` environment and is sent from there in a service context.

```
typedef sequence<octet> ProxyCode;
interface ProxyRepository {
    ProxyCode get_code(in string RepID);
};
```

Since there is no reason to use an external protocol for the actual code shipping, code repositories are implemented as CORBA objects that encapsulate proxy libraries as a sequence of octets as shown in the IDL description above.

Alternatively to including a repository IOR, the server itself could always be queried for the code. Keeping all code at the server, however, would be less flexible than using one or more separate code repositories. A further alternative is attaching the library code itself to the reply message avoiding an additional

invocation. In this case, some effort has to be put into the server side not to send large libraries multiple times and, again, only the server could store the proxy implementations.

5 Example

To demonstrate how smart proxies can be used in CORBA, we have implemented a live-video service providing access to a camera, and a video-conferencing client. Smart proxies were used to provide several communication mechanisms in addition to IIOP. The IDL of the service interface is as follows:

```
abstract interface Video {
  void start(in unsigned short frameRate);
  void stop();
  boolean getFrame(out Frame f, out Time t);
  void close();
};
```

There are five implementations of the service. All of them use CORBA remote object invocations for controlling the transmission, but transmit the actual data differently.

1. The server sends frames to a proxy via UDP. Frames must be fragmented and packet loss must be handled correctly.
2. Video frames are JPEG compressed and also sent via UDP.
3. Proxy and server communicate via TCP/IP.
4. When client and server run on the same node but in different processes, video data can be efficiently passed from server to proxy via shared memory.
5. A server without a proxy communicates directly with its clients and actually transmits the frames in the `getFrame` calls via IIOP.

The video-conferencing client connects to one or several service implementations and displays the frames received from the respective sources as shown in Fig. 4. In this simple scenario, a server can choose among protocols by simply using information about the network topology and the address of the client.

Servers and client are built with ORBacus 4.0b2 for C++ and run on x86 PCs with Linux 2.2 using a Hauppauge frame-grabber card with a camera as the video source. Fig. 4 displays the frame rates achieved by the respective service implementations demonstrating the different performance characteristics. For the measurements, the client only connects to one video source, either via a 10 Mb/s Ethernet or locally.

Due to the smart proxies, the implemented communication mechanisms with their diverse characteristics are uniformly accessed by the client. More elaborate smart proxies can be used to transparently handle QoS management including QoS mapping and resource reservation as we have demonstrated with the non-CORBA version of this service [4]. Note that with the various communication

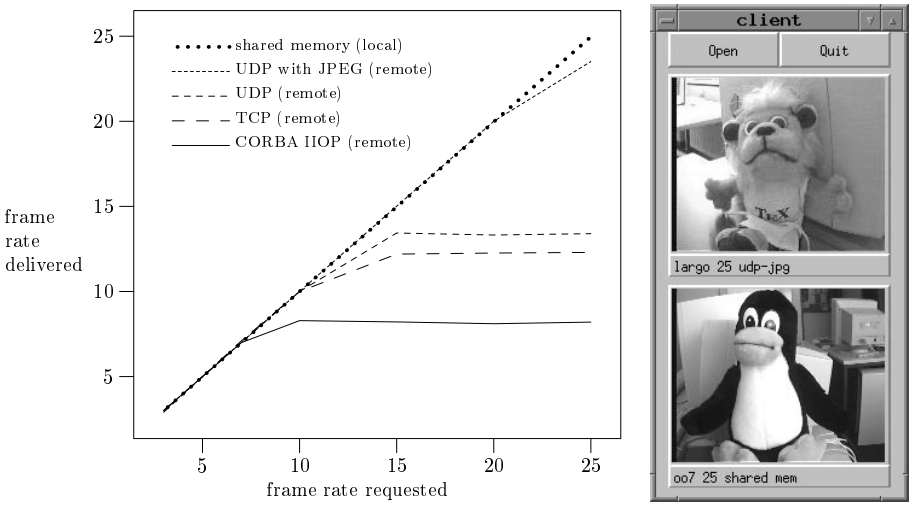

Fig. 4. Frame Rates for Various Service Implementations and Screen Dump

mechanisms being encapsulated in the smart proxies, other clients such as a video surveillance application can also transparently use the service without modification.

6 Related Work

Conceptually, smart proxies are related to fragmented or distributed objects as conceptionally proposed by Makpangou et al. [6] and applied by Globe [14, 15] and AspectIX [3], for instance. These objects are physically distributed and consist of fragments on different nodes. Distribution and inter-fragment communication are hidden from other objects and clients. Globe aims at improving scalability for distributed applications, for instance by providing replication and caching for web documents. AspextIX is extending CORBA to enhance QoS and support for mobile and reconfigurable object fragments. In contrast to the symmetric model of fragments in a distributed object, smart proxies and servers have distinct roles, still similar to the familiar roles of client and server. Hence, the use of smart proxies may be more easily adopted by programmers than the development of services as distributed objects. Moreover, smart proxies require less platform support and even can be built on a standard platform such as CORBA as shown in this paper.

Some CORBA ORBs such as Orbix [1] and TAO [11] also support a kind of smart proxies. They allow application programmers to manually replace the IDL generated stubs with “smarter” ones, which may implement additional fun-

ctionality such as caching, load balancing, or merging remote invocations for efficiency. While this approach can improve performance, improvements must happen independently of the server and, hence, possibilities are limited by the regular remote object interface. In our model in contrast, smart proxies are server-specific and can particularly benefit from improved communication protocols between them and their servers.

Another way of integrating advanced functionality in proxies at compile time is used in the QuO architecture [5,16]. This platform uses QuO definition languages (QDL) in addition to an IDL for specifying QoS and adaptive behaviour of a service. From these descriptions so called delegates are generated and linked to the client like regular stubs are generated from an IDL. In this way, resource reservation, QoS monitoring, and adaptation can be easily integrated within delegates. Functionality beyond the capabilities of the code generator, however, can only be added to the QDLs as source code, exposing implementation details in the service interface.

In Sun's Jini environment [13,17] proxies also are an important concept. When accessing a service, the client receives sort of a smart proxy from the lookup service. This proxy then handles communication with the server. This mechanism allows devices and services to be dynamically added and removed from the system. Also, proxy and server can choose their own protocol for communicating with each other. The underlying Java and RMI infrastructure [12] provides advantages of security, ease of code shipping, and platform independence, but also incurs the drawbacks of restriction to one language, and the potential performance penalties and unpredictability of a virtual machine.

7 Conclusions

Loadable smart proxies can be used to encapsulate service-dependent client-side code in self-contained modules. The service developer has both ends of a connection under control and may choose the most appropriate communcation mechanisms for a particular service implementation. This way, different network protocols and QoS management functionality can be integrated. These different service implementations are hidden from the client developer, who only accesses a high-level interface defined in IDL. Hence, location transparency can be provided even for services requiring specific functionality on the client node. Moreover, this functionality is available to all client applications using a particular service and need not be re-implemented in each of them, which is an important prerequisite for the formation of a universal service market.

In this paper, we have shown how dynamic loading of native-code proxies can be implemented in CORBA. The mechanism proposed, however, is effectively a generic native-code shipping service for value-type objects. This service does not depend on proprietary ORB extensions but can be used on any CORBA platform supporting OBV and portable interceptors. The implementation of a video service has demonstrated the feasibility and utility of our approach.

Acknowledgements. We are indebted to Marcus Demker for implementing the video service in part. Moreover, we thank the anonymous reviewers for their helpful comments.

References

[1] S. Baker. *CORBA Distributed Objects Using Orbix*. Addison Wesley, 1997.
[2] Microsoft Corp. *Distributed Component Object Model Protocol*, 1998.
[3] F. Hauck, U. Becker, M. Geier, E. Meier, U. Rastofer, and M. Steckermeier. AspectIX, an aspect-oriented and CORBA-compliant ORB architecture. Technical Report TR-I4-98-08, Friedrich-Alexander-University, Erlangen-Nürnberg, September 1988.
[4] R. Koster and T. Kramp. Structuring QoS-supporting services with smart proxies. In *Proceedings of Middleware'00 (IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing)*. Springer Verlag, April 2000.
[5] J. P. Loyall, D. E. Bakken, R. E. Schantz, J. A. Zinky, D. A. Karr, R. Vanegas, and K. R. Anderson. QoS aspect languages and their runtime integration. In *Proceedings of the Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR98)*, volume 1511 of *Lecture Notes in Computer Science*. Springer Verlag, May 1998.
[6] M. Makpangou, Y. Gourhant, J.-P. Le Narzul, and M. Shapiro. Fragmented objects for distributed abstractions. In T. L. Casavant and M. Singhal, editors, *Readings in Distributed Computing Systems*, pages 170–186. IEEE Computer Society Press, July 1994.
[7] OMG. CORBA telecoms specification. `http://www.omg.org/corba/ctfull.html`, June 1998. formal/98-07-12.
[8] OMG. Portable interceptors revised submission. `http://www.omg.org/cgi-bin/doc?orbos/99-12-02`, 1999. orbos/99-12-02.
[9] OMG. *The Common Object Request Broker: Architecture and Specification (Release 2.3)*, June 1999.
[10] The Open Group. *Introduction to OSF DCE 1.2.2*, November 1997.
[11] Kirthika Parameswaran. TAO release information: Smart proxies.http:// `www.cs.wustl.edu/ schmidt/ACE_wrappers/TAO/docs/Smart_Proxies.html`, September 1999.
[12] Sun Microsystems. Java remote method invocation specification, October 1998.
[13] Sun Microsystems. Jini architectural overview, 1999. Technical White Paper.
[14] M. van Steen, P. Homburg, and A. S. Tanenbaum. Globe: A wide-area distributed system. *IEEE Concurrency*, pages 70–78, January-March 1999.
[15] M. van Steen, A. S. Tanenbaum, I. Kuz, and H. J. Sips. A scalable middleware solution for advanced wide-area web services. In *Proceedings of Middleware '98 (IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing)*, pages 37–53. Springer Verlag, September 1998.
[16] R. Vanegas, J. A. Zinky, J. P. Loyall, D. A. Karr, R. E. Schantz, and D. E. Bakken. QuO's runtime support for quality of service in distributed objects. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*. Springer Verlag, September 1998.
[17] J. Waldo. The Jini architecture for network-centered computing. *Communications of the ACM*, 42(7):76–82, July 1999.