Design and Implementation of the Portable Object Adapter

Frank Pilhofer

Fachbereich Informatik
Johann Wolfgang Goethe-Universität
Frankfurt am Main

June 1999

Contents

1 Introduction **1**

2 Distributed Systems **3**
2.1 Introduction 3
2.2 Low-Level Distribution 6
2.2.1 Sockets 6
2.2.2 Parallel Virtual Machine 7
2.3 High-Level Distribution 8
2.3.1 Erlang 8
2.3.2 Distributed Oz 9
2.4 Middleware 10
2.4.1 Remote Procedure Call 11
2.4.2 CORBA 13
2.5 Evaluation 14

3 Object Request Broker **17**
3.1 Interface Definition 18
3.2 Addressing 20
3.3 Invocation 22
3.4 Interoperability 24
3.5 Evaluation 24

4 Object Adapters **27**
4.1 Basic Object Adapter 29
4.1.1 Shortcomings 31
4.1.2 Solutions 33
4.2 Portable Object Adapter 34
4.2.1 Policies 35
4.2.2 POA Managers 36
4.2.3 Request Processing 37
4.2.4 Persistence 41
4.2.5 Language Mapping 43
4.2.6 Evaluation 45
4.3 Other Object Adapters 46

5 Reference Implementation **49**
5.1 MICO 49
5.2 Design 52
5.3 Object Key Generation 55
5.4 Persistence 57
5.4.1 POA Mediator 58
5.4.2 Synchronization 60
5.4.3 State Information 63
5.4.4 Usage 63
5.5 Collocation 64
5.6 Other Implementations 70
5.6.1 ORBit 71
5.6.2 TAO 71
5.6.3 ORBacus 72
5.6.4 Comparison 73
5.7 Evaluation 73

6 Conclusions **75**

A Glossary **77**
A.1 Abbreviations 77
A.2 Terminology 79

B Examples **83**
B.1 POA 83
B.1.1 Example 83
B.1.2 Using a Servant Manager 85
B.1.3 Persistent Objects 87
B.1.4 Reference Counting 90

C POA IDL **91**

Bibliography **95**

Chapter 1

Introduction

As systems become more interconnected, there is a growing need for these systems to co-operate. While basic message interchange (like e-mail or the downloading of files) is already common, the possibilities of distributed systems within this interconnected world are yet somewhat uncharted. One reason is the inherent complexity of a distributed system, which makes distributed components much harder to design, debug and maintain than standalone pieces of software.

But the advantages of a distributed system are too promising to ignore. They offer greater speed by spreading complex tasks over many computers working in parallel, fault-tolerance by using redundant servers, or delegation, where a client contacts the server to perform some type of service. The offering and using of services are of particular interest for the Internet today, where companies are not satisfied with presenting information on the World Wide Web, but also want to present their services, online, to the world-wide user community.

Software – easy to use distribution platforms – is needed to make this feasible.

Communication is provided by networks, and it is certainly possible to build distributed systems with only what the network itself has to offer. However, distribution is much easier to achieve, to maintain and to extend, if the distribution platform provides an abstraction of the network and allows a more natural usage of remote components. Chapter 2 presents and evaluates a number of existing distribution platforms.

A common idea is the middleware approach of introducing a new layer into the program that keeps the complexity from the developer by hiding as many details of distributed programming as necessary. To the developer, the middleware presents seemingly local objects, and invocations on the local proxy cause the necessary data to be transparently sent to the recipient.

One popular example of such middleware is the Common Object Request Broker Architecture (CORBA), which is part of the Object Management Architecture as specified by the Object Management Group. CORBA uses the Object Request Broker as the glue between individual pieces, which is responsible for directing a client's method invocation to the appropriate object implementation.

CORBA is already in widespread use, both in education and business. An Object Request Broker comes built into the popular Netscape Navigator, a freely available hypertext browser for the World Wide Web with a sizable market share. With CORBA technology on desktops everywhere, the above scenario of offering world-wide services is at hand, but not yet reality.

While CORBA can indeed hide many details of Client/Server programming from the client, experience has shown that a much tighter involvement with the Object Request Broker is necessary on the server side. For this, CORBA uses the concept of object adapters which mediate between the ORB and the server on how to represent servants to the outside world; servers can choose an object adapter that best fulfills its requirements.

This thesis focuses on the Portable Object Adapter, which was recently added to the CORBA standard to replace the original Basic Object Adapter. The latter proved to be ill-specified and insufficient for many a server's requirements. However, while the POA's specification drew from its authors' experiences with the BOA's shortcomings, the POA had not been implemented at the time of its writing.

The goal of this thesis was to verify that the POA specification is sensible and complete, by attempting a reference implementation.

An object adapter is not a standalone entity, it is but a part of the architecture and requires an Object Request Broker to cooperate with. Rather than reinventing the wheel, the Open Source CORBA implementation Mico was chosen as platform. Mico was designed to be an extensible ORB; in particular, it allows for pluggable object adapters. Yet, as the implementation of the POA was begun, no object adapter existed in Mico but the ones it was originally designed for. Therefore, a side goal was to see how Mico's extensibility could cope with an object adapter whose requirements had not been anticipated.

After presenting and comparing other solutions for distribution in the second chapter, the world of CORBA is explored in more detail throughout the third chapter.

The fourth chapter takes a look at the server side of CORBA programs, explaining the design principles of object adapters in general and their realization by the two object adapters that have been specified by the Object Management Group themselves, the old Basic Object Adapter and the new Portable Object Adapter. The reasons why the BOA was abandoned will be analyzed, as are the areas the POA improves upon.

After all the abstract discussion in the previous text, chapter 5 first takes a brief glance at Mico and its extension mechanisms, and then presents the hands-on experience that was gathered in the attempt for a complete reference implementation of the POA and the extensions that were required.

The thesis then concludes with a look at the goals presented above, to what extent they have been achieved, and which areas of the Portable Object Adapter still need improvement. A few examples of POA-based server programming are included in the appendix.

At the time of writing in June 1999, version 2.2 of the CORBA specification was current, but draft documents of CORBA 2.3 were available and close to become official. This thesis mainly refers to CORBA 2.2, with frequent annotations about changes in 2.3 being based on the December 1998 draft.

Chapter 2

Distributed Systems

2.1 Introduction

The reasons for developing distributed systems are about as manifold as there are definitions for the term "distributed system" in the first place. The most basic one is that of a collaboration of individual components that communicate to achieve a common goal, which could cover everything from Unix shell scripts (where various programs communicate using pipes) to the loose collaborations on the Internet where hundreds or thousands of users contribute computing power to work on crypto challenges. Both extremes do not seem so far-fetched when looking at the most popular design strategies used in distributed systems:

- Parallel processing
- Fault tolerance
- Delegation
- Resource sharing

The first item mentioned here was the historically first, too: when distributing a single task over a number of processors, parts of the overall task can be computed in parallel to speed up the total process. Fault tolerance is relevant in fail-safe systems (e.g. airplanes), and achieved using redundancy: the same computation is performed multiple times to detect and to safely recover from the failure of individual nodes. Delegation is the basis of Client/Server systems, where a server performs a particular service on behalf of the client, like a database retrieval. Resource sharing is a different kind of delegation, where systems offer their special information or hardware resources, for example a graphical X11 display, the operation of a robot or numerical computation capabilities.

Parallel processing, fault tolerance and resource sharing specialize the above definition of a "distributed system" in that they mandate a distribution over interconnected nodes in a network. This is obvious in the case of parallel processing, which only makes sense when distributing the job over a number of distinct processors; and redundancy also can only be achieved with physically separate systems, because the failure of any part of a node makes any of its results suspect. However, delegation is just as useful on a single computer, as in the Unix shell example.

One central keyword in all definitions of distributed systems is *communication.* Communication is necessary to distribute information among the components that make up a distributed system, and to collect the partial results of each component in order to compute a total answer to the common goal mentioned above.

While the term communication can be stretched to the extreme (global variables in a program can be said to be a means of communication just as well as common disk files or shared memory), the usual case of distribution over networked nodes requires some sort of network-aware transport.

The need of communication, potentially over a shared resource like a network, introduces a number of potential problems that a distributed system must be aware of:

Communication Overhead: Transferring data is costly and takes time. Distribution does not make sense where the time for communication exceeds the amount spent performing computations. The bandwidth available for communication may be limited and not constant.

Points of failure: Components can fail individually, or communication links may fail.

Security: Communication over a public network could be intercepted, manipulated or interferred with by a third party.

Synchronization: It may be necessary for a distributed system to wait for a single slow component before computation can continue.

Just as not all of the advantages apply to a given distributed system, the impact of these potential problems also depends on the particular system which is to be implemented: while the interception of data by a hostile attacker might be ignored in a local network, it must be dealt with when private data needs to be transmitted over the Internet. Depending on the application, the manifold failure points in communication must be handled specifically. For non-critical systems, or for small systems in a local network where failure is rare, it may be acceptable for an application to simply crash in the case of a failure: it will then be the task of the administrator to discover and correct the cause of the failure, and to restart the application. In other cases, the software must be prepared to recover reliably from a partial system failure and perhaps use redundant components instead.

The term *distribution platform* shall be used to describe a generic means of distribution provided by an operating system, library or language. A distribution platform at least hast to provide primitives for communication and coordination for use by the programmer to build the distributed system on top of the platform. It should help a developer to utilize the advantages of a distributed system as easily as possible while minimizing or at least controlling the potential problems.

A distribution platform has to provide at least three primitives so that implementing a distributed system becomes feasible: *addressing*, *synchronization* and *encoding*.

Addressing: In order for components of a distributed system to find each other, each must have an address. Servers must be able to publish their address, and clients must be able to acquire a server's address, either by administrative intervention, guessing, or looking it up in a naming service.

Synchronization: All participating parties must agree on a certain protocol when data may be sent or received. Common data transfer protocols are messages, request-response or data streams.

Encoding: Defines the format used for transferring data. This could be uninterpreted chunks of octets or structured data.

A distribution platform can be expected to use abstractions of these primitives to ease development, for example by automated encoding of complex data. In a synchronized system, the developer is not involved in issues of timing, nor has to be concerned with the ordering of incoming data. Abstracting all three issues provides transparency, meaning that the developer does not need to be aware of how components are located and how data is communicated between them.

Apart from transparency, other key features of distribution platforms are:

Mobility: Components can be fixed to one location, move in an inactive state, or even move while being active. Mobility can be coarse-grained (servers) or fine-grained (objects). Location selection could be client-controlled, server-controlled or automated, with an appropriate location selected by load-balancing, network usage or hardware requirements.

Quality of Service: Addresses the non-functional aspects of distribution, e.g. a distributed system's control over different data transports (bandwidth limitations, encryption), timing (real-time requirements), or protocols to handle redundancy or error recovery (automated recovery vs. notification vs. abortion).

Interoperability: Closed systems can live without interoperability, but in a heterogeneous world, it is useful if the distribution platform is not limited to a particular hardware, operating system or programming language. It is unlikely that independent clients and servers can agree on identical hardware and software.

A number of distribution platforms have been developed, some with a specific use like parallel processing in mind, others with the intention of a generic platform for distributed applications. A couple of examples are explored in the following sections, grouped by the level of abstraction provided for the issues above: low, middle or high.

This categorization can be done by applying the popular Open Systems Interconnection (OSI) Reference Model [70] to distribution platforms. The OSI model takes a layered approach to networking, each introducing a level of abstraction. The full model has seven layers, but the lower four, physical, data link, network and transport, deal with low-level issues such as failure detection and routing and are of less interest to the application programmer, who is content with secure communication on top of layer four.

A low-level distribution platform does not abstract any further and allows no more than data transport. Figure 2.1 shows the upper three layers of the OSI model on top of the transport layer, with the distributed system built on the top application layer.

Slightly deviating from OSI terminology, a high-level distribution platform can be said to provide mechanisms naturally built into the programming language, so that locality or remoteness is entirely transparent.

The two layers in between, again in line with OSI, provide session control, the handling of connection between distributed components, and the presentation layer encodes complex

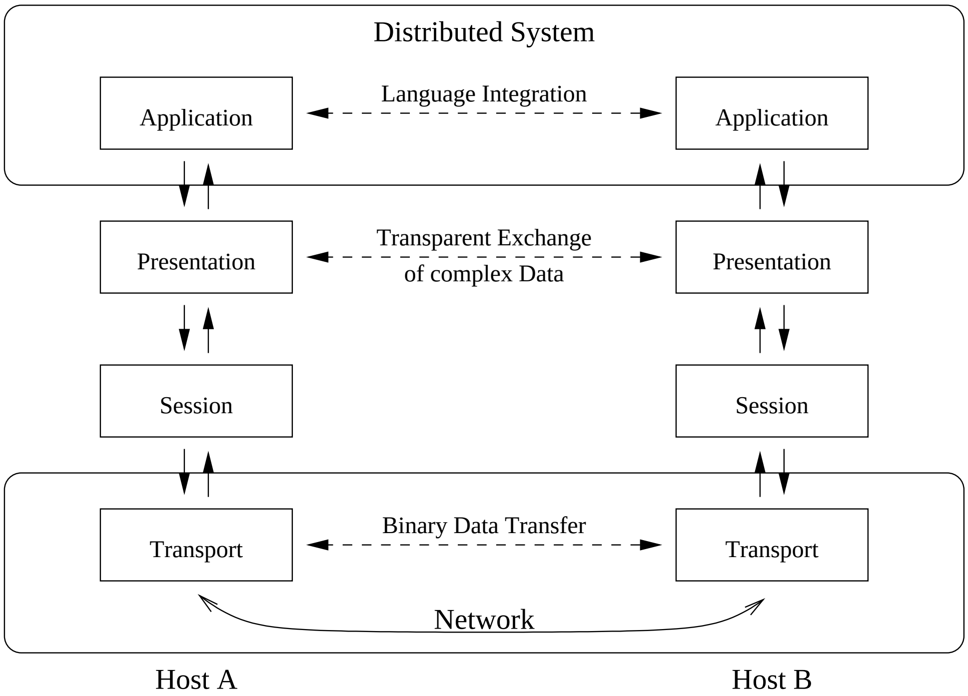

Figure 2.1: The OSI model applied to Distribution Platforms

application-dependent data structures like method parameters into a common format, probably adding data compression or encryption – and this is just what can be expected from middleware.

2.2 Low-Level Distribution

The most basic distribution platforms are no more than communication packages that allow the sending and receiving of raw data between components. Components are contacted at a known location, and data can be exchanged using a predefined protocol. Examples for this kind of distribution are BSD sockets and PVM.

It is something of an exaggeration to describe sockets as a distribution platform, as this is not true to their design as a means of exchanging data. Sockets are presented here because of their significance as a transport for more sophisticated distribution platforms.

PVM, however, is a valid candidate as a distribution platform, if only to demonstrate the limitations of a low-level approach.

2.2.1 Sockets

Sockets [67] were originally introduced in version 4.2 of the BSD Unix operating system, and have since been added first to other Unix's, then to other operating systems as well. Today, support for sockets is ubiquitous and part of both the Single Unix Specification [17] and the upcoming IEEE P1003.1g Protocol Independent Interfaces standard, part of the POSIX family. They provide a common programming interface to TCP/IP (Transmission Control Protocol, Internet Protocol [48, 49]) and UDP/IP (User Datagram Protocol [47]), the basis for the Internet and similar networks.

Addressing is done using a *host name* and a *port number*, a numerical identifier in the range from 0 to 65535. Components are usually stand-alone processes. Servers (processes that expect incoming connections) *bind* themselves to a port, where they can be contacted by clients. Communication can be connection-oriented (data streams) or connection-less (message based). Multicasting can be used in local networks, where messages are sent to a group of recipients.

Host names and port numbers must be agreed on by all components of a distributed system. Port numbers are usually the same on all hosts and can be identified by name, using a database. Well-known port numbers can be registered with the Internet Assigned Numbers Authority [53], which thereby provides a primitive naming service.

Data transfer is done on a binary level, as chunks of uninterpreted octets, and recipients must know how the data is to be read and interpreted. Problems may occur in a heterogeneous environment, because raw integer and floating-point values may differ depending on programming language and hardware. Applications concerned with interoperability must take care to convert their data into a "common format" before sending it, so that it can be changed back into the potentially different native format on the receiving end. Many existing applications use a string-based protocol, where all data is encoded as a string.

Sockets present the developer with little help for distributed programming. Addresses are intuitive, using the Internet host name and a common port number, but no support for synchronization and encoding exists. The application itself must take care of establishing a protocol and packaging data.

Still, the high availability even in a heterogeneous environment and low cost (no cost if systems are connected to the Internet or an Intranet anyway) make socket-based programming attractive.

An example of a socket-based distributed system is the world-wide Usenet network. Most hosts on the Internet run the NNTP (Network News Transport Protocol) service [18], which can be contacted on the well-known port 119. All participating hosts cooperate in the distribution of messages by sending incoming news to other directly connected hosts, so that the messages ultimately arrive everywhere.

Classical IP networks can only offer best-effort capabilities in transmitting data, because cable lines have limited bandwidth and are shared fairly between numerous users, as each datagram is forwarded on a first-come, first-served basis.

Asynchronous Transfer Mode (ATM) [3] is a new "telecommunications concept" that includes support for reliable quality of service guarantees over a shared network. Some operating systems like Linux implement the socket interface over ATM [1], so that sockets can be guaranteed a fixed bandwidth and latency according to a "User-Network Traffic Contract."

Many higher-level distribution platforms build upon sockets, and in the generic sense there is no difference if sockets are implemented over a classical IP network or over ATM, but quality of service features will in most cases require an ATM basis.

2.2.2 Parallel Virtual Machine

Whereas sockets are basically a means of data transport, the Parallel Virtual Machine (PVM) [11] was explicitly designed with parallel distributed systems in mind. Its development started in 1989 at Oak Ridge National Laboratory with the intention of providing an efficient platform for parallel processing, using multiple workstations rather than supercomputers. Despite its roots in number crunching, PVM can be used as a universal distribution platform.

A virtual machine is composed of one or more hosts, each running a “PVM Daemon.” Some of these hosts can by themselves be multiprocessor machines, each processor is then identified as a *node*. Components of a distributed system, *tasks* identified by a unique *task identifier* (tid), can then be started on any node. Tasks can then *spawn* new child tasks by themselves.

Communication among tasks is done by asynchronous messaging and signaling. The sender fills a message buffer with data and then addresses the buffer to one or more recipients, using their tids. A message buffer can hold a mixture of basic data: octets, integer values, floating-point values and strings.

Interoperability is ensured by encoding messages using the External Data Representation standard [66] to transport messages between hosts. Implementations of PVM first flourished on Unix, but have also been ported to other operating systems; the two supported programming languages are C and Fortran.

Usually, PVM applications are based on a Master-Slave design, in which one master task starts up a number of slaves and then distributes small pieces of a computation to each of them. The slaves then operate in parallel, and the master waits to collect the results and to put the pieces together. In the world of high-performance computing, PVM has become such a success that it has been ported to and optimized for a wide range of massively parallel computers by their vendors, using their proprietary means of transferring messages, or even shared memory.

PVM abstracts the network by using task identifiers for addressing and a common data encoding. Message passing is used as synchronization paradigm. But still, components must agree on the protocol specifying when messages can be sent or received, and must know how to interpret the data contained in a message.

Although difficult to imagine, generic distributed systems could indeed be built upon PVM, but its features are too limited and not easy to use. In fact, PVM is a good demonstration of why an abstraction of a component’s interface and OSI’s presentation layer are needed to provide more transparency in passing data. The composition of messages is complicated enough as it is, and small changes in the message format require changing the code of all other components.

2.3 High-Level Distribution

In distribution platforms that provide a high level of abstract distribution, communication mechanisms are an integral part of the overall system, so that invocations on remote components are transparent, i.e. indistinguishable from local communication. Two examples are Erlang [2] and Distributed Oz [13].

2.3.1 Erlang

Erlang is a functional programming language for, according to the authors, concurrent, real-time, fault-tolerant distributed systems. It was developed by Ericsson and the Ellemtel Computer Science Laboratories for use in Ericsson’s Open Transport Platform, an infrastructure for telephony applications, and has since been made available to the public.

With its focus on concurrent programming, ERLANG provides lightweight *processes* that are executed in parallel, similar in notion to threads in other programming languages. The only means of communication between processes is the sending of *messages*.

The language makes it especially easy to program event-based systems, and a popular design choice is to build a concurrent system as a number of communicating finite state machines. This design aims at implementing particularly reliable systems.

Distribution in ERLANG is realized as a special case of concurrency in that a new process is spawned on a remote node. Communication with a remote process is no different from talking to a local process. Since parallelism and messaging are natural mechanisms in ERLANG, distributed programming looks the same as non-distributed programming.

Special messages are sent by the ERLANG kernel if a remote process exits or fails and thus allows the developer to react to a failure in a fault-tolerant distributed system. Real-time is only handled passively, an application can limit the time spent in waiting for a message, but has no control over the delivery of messages.

With the modern concept of automated code loading, ERLANG programs do not need to be concerned with whether the code they want to execute is actually available on the remote node, as it will be downloaded automatically by the kernel.

ERLANG fully abstracts the network. Addressing is done using process ids, encoding and transferring a message to a remote process is transparent. The control over quality-of-service and real-time issues is, in the words of the authors, "soft," as no guarantees are given. By timestamping messages and using timeouts, an application can only detect the failure case that a message was not received in time. Interoperability, however, is limited to ERLANG programs running under the control of the ERLANG kernel, which must be running on all nodes. A means of integrating code from foreign programming languages is provided, but requires the reading and writing of messages in ERLANG's native data format.

Distributed programming in ERLANG succeeds in a closed environment where the developer has the choice over all parts of the system and where the software is written from scratch, but as integration with other common programming languages is not possible, the developer must submit to both the language and its paradigms.

2.3.2 Distributed Oz

Distributed Oz is an extension to the Oz language developed at the German Research Center for Artificial Intelligence (Deutsches Forschungszentrum für künstliche Intelligenz, DFKI). Oz is advertised as a concurrent, "multi-paradigm" programming language and allows object-oriented as well as functional programming.

Like ERLANG, Oz provides an asynchronous messaging mechanism, *ports*, as a means of communication between threads, and the step from local messaging to the distributed exchange of messages is just as short. Synchronous method invocations on objects are realized in a Smalltalk fashion by sending a message to the object, and could therefore be remote as well.

But instead of using remote method invocations, Distributed Oz uses lightweight object mobility. If a message referencing an object is sent to a remote site, first a *proxy object* is created on the remote end. If a method is invoked on the proxy, it first loads a duplicate of the object's *class record*, which holds the methods' code, and then transfers the object's *state record*, of which only one copy exists. The state pointer on the originating side is itself then replaced by a proxy.

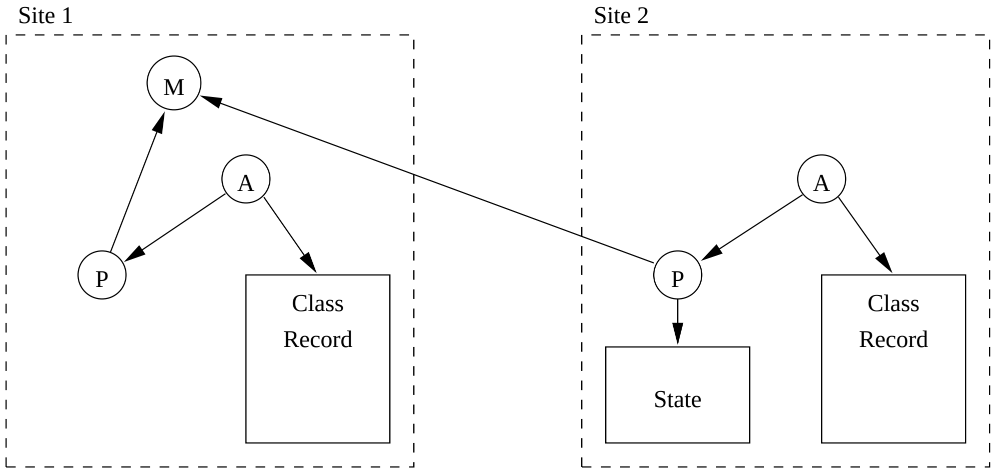

Figure 2.2: Object Mobility in Distributed Oz

A manager exists on the site where an object was initially located to keep track of its state record. Figure 2.2 shows a scenario where an object A has been accessed remotely. The class record exists on both sites, but the state record is kept on site 2 only. Both sites hold a reference to the manager M, and should A be referenced again on site 1, only the state needs to be moved back.

Object mobility is a controversial topic and depends on the situation, specifically, on the size of the object state compared to the data actually required by the remote site. Additional network traffic is needed to locate an object before it can be used. Distributed Oz currently does not offer a distinction between mobile and immobile objects. Oz *threads* are stationary but are limited to message-based communication and do not offer an object interface.

2.4 Middleware

Both low-level and high-level approaches of abstracting distribution are limited in their usability. Low-level distribution platforms like PVM still require the developer to be concerned with many details of the interaction between components, like the synchronization and the packaging of complex data into simple transferable chunks. On the other hand, high-level platforms are, more or less by definition, tightly integrated in a particular programming environment and limited to one programming language. Before a distributed system can be implemented, developers must familiarize themselves with a new language's syntax constructs and programming paradigm.

The *middleware* approach lies in between. The idea is to extend an existing programming language by introducing a new layer "in the middle," between the application and the network, that hides the complexity of communication and data transfer. To the developer, the invocation of a remote procedure should appear no different than the invocation of a local one. According to figure 2.1, middleware provides the distributed system with both a presentation and a session layer, automating the encoding and transfer of parameters.

Many middleware architectures exist; some in common use today are Remote Procedure Calls (RPC), the Common Object Request Broker Architecture (CORBA), the Distributed Component Object Model (DCOM) and Java Remote Method Invocations (RMI).

```
struct Account {
  string name;
  unsigned long balance;
};

program ACCOUNT {
  version ACC_VERS {
    Account deposit (Account, unsigned long);
    Account withdraw (Account, unsigned long);
    long    balance (Account);
  } = 1;
} = 0x42424242;
```

Figure 2.3: Service Interface Description in the RPC Language

Java RMI [14] recycles many of the ideas introduced by RPC and CORBA but is tied to a single programming language. While being widely deployed today, the interest in RMI fades as CORBA is becoming a core feature of Java [69]. DCOM [54] is widespread in the Windows world, but proprietary in design. A comparison of CORBA and DCOM can be found in [44]. The other two, RPC and CORBA, are examined in greater detail.

2.4.1 Remote Procedure Call

Remote procedure calls (RPC) were proposed as early as 1975 [73], their history is documented in [68]. A first implementation appeared from Xerox in 1984 [6]. The following text describes Sun RPC, released by Sun Microsystems, Inc., which is the implementation in most widespread use today. A different RPC mechanism is available as part of the Open Software Foundation's Distributed Computing Environment (DCE RPC).

RPC allows the the implementation of Client/Server distributed systems: clients can connect to a remote server and invoke one of the *services* provided by the server. On both the client and the server side, the invocation of the remote service appears as a normal procedure call. The client can pass a number of parameters that are transferred to the server side. The service evaluates the parameters and returns a result, which is transferred back and returned as result from the remote procedure call.

In order to use remote procedure calls, the collection of services provided by a server program first has to be described using the RPC Language. Each server is assigned both a name and a unique *program number*, and all services are declared with their full parameter lists. The RPC Language provides both simple types like octets or integers, and complex types like structures, enumerations or sequences.

The `rpcgen` program then reads an RPC Language file and produces both declarations and code for the C programming language. A header file contains C type definitions for all the types used in the RPC Language file, and two source code files containing a *client stub* and a *server stub*.

Before invoking services, however, client and server have to find each other using the *portmapper* (more recently called *rpcbind*), which is a permanently running daemon. The `rpcgen`-generated server stub code takes care of registering a server with the portmapper upon startup. Clients need to know a server's program number as contained in the RPC Language file and the name of the Internet host the server is running on. They then contact that host's portmapper, to acquire a *client handle*, which then serves as a server reference.

A client program then uses a local procedure call into the client stub, which provides the same signature[1] as the service itself, taking only the client handle as an additional parameter. The client stub transparently communicates the service's parameters to the server program by sending an RPC request. On the server side, this request is extracted by the server stub, which again performs a local procedure call into the user-provided service implementation. The service's result is then returned the same way.

Data is encoded using the External Data Representation [66] standard, the same as in the Parallel Virtual Machine (see section 2.2.2), ensuring interoperability between different hardware architectures.

RPC demonstrates the important middleware concept of separating interface and implementation. From the abstract interface declaration, the middleware generates code that aids in distribution by presenting the application with the usual local procedure call semantics and that automates data transfer and synchronization, performing as the application would expect a local procedure to behave.

Because of its heritage in the Unix environment, `rpcgen` can only produce stub code for the C programming language. However, the encoding of parameters is known, and the format for RPC messages is well-documented [65], so interoperability with different programming languages, or the implementation over networks other than the original TCP/IP are possible, as demonstrated by PVM.

The Network File System [8] is a successful and widespread example using RPC as distribution platform.

RPC can be used over both TCP and UDP. While the former guarantees *exactly-once* semantics,[2] i.e. the remote procedure is called exactly once, this is not the case with RPC over UDP, where requests are resent by the client stub after a timeout, and invocations use *at-least-once* semantics unless the server activates a *duplicate request cache*. NFS uses RPC over UDP for increased efficiency and must therefore use *idempotent* procedures that can safely be called more than once. A distributed system that does not want to be concerned with execution semantics and idempotence will obviously choose the TCP transport.

One crucial limitation of Sun RPC is its procedure-orientation, which enforces a stateless protocol. A service cannot identify the client [68] and therefore cannot associate state information with a transaction. In a stateful protocol, the state would have to be sent back and forth between client and service as an additional parameter (the `Account` data in figure 2.3).

Servers can be either single-threaded, where procedure calls on the server side are serialized, or multi-threaded, where requests are dispatched in parallel. Unfortunately, when asked to produce thread-safe code, `rpcgen` assumes multi-threading on the client side, too. In thread-safe code, the client stubs and services take an additional parameter, so the client-side interface is incompatible with the single-threaded version.

On the client side, remote procedure calls are always synchronous and block until the service's result is received. It is also impossible to mix client and server functionality. While a server can also act as a client and perform remote procedure calls on its own in order to service a request, a client program cannot offer services on its own. The cooperation is

[1] The signature is the set of all parameter types and the result type.

[2] Under the condition that neither client nor server crash.

therefore strictly limited to concrete *service-providers* and *service-users*.

Lastly, services are identified by a number only. This number is chosen in the RPC Language file and hardcoded by the `rpcgen` program. The developer usually chooses a program number arbitrarily. If a different server uses, by coincidence, the same program number, clients might address a completely different server. It is much easier to avoid name clashes in a namespace using human-readable and descriptive plain-text names.

Despite its shortcomings, RPC is an important member of the middleware family, as it does already use a separate interface declaration and generated stub code while predating other middleware by half a decade.

2.4.2 CORBA

The Common Object Request Broker Architecture [63] by the Object Management Group was first published in 1991. Because CORBA is the focus of this thesis, the entire next chapter is spent on detailing the Object Request Broker, the central part of the architecture. This section presents the basic design of CORBA and compares it with RPC as middleware.

The basic idea of CORBA is much the same as it is for RPC. But whereas RPC is based on a procedural language and allows remote procedure calls, CORBA is based on object-oriented programming and allows *remote method invocations* on objects. Orfali's definition of an object [44] is rather short and vague: "Objects are a blob of intelligence," expressing that an object provides methods on the outside, but that the internals of an object are unknown. Cardelli and Wegener [9] are more verbose and define that a language is object-oriented if and only if it satisfies the following requirements:

- It supports objects that are data abstractions with an interface of named operations and a hidden local state.
- Objects have an associated type [class].
- Types may inherit attributes from supertypes.

It is important to note that the first and last item combined express polymorphism, that an object of a subtype can substitute an object of its supertype, because all attributes are inherited. The same properties must be expected from an object-oriented distribution platform and are indeed realized in CORBA. The first item addresses one of RPC's weaknesses (see section 2.4.1) by providing stateful services. Servers contain a number of objects that can be addressed individually, and the services that operate on objects only need to be implemented once.

Like RPC, CORBA uses a declarative language, the Interface Definition Language (IDL), to describe an object's interface. As with RPC, this description is used by an *IDL compiler* to generate *stubs* for the client side and *skeletons* on the server side. Using this generated code, remote method invocations look like an invocation on a local object, at least in an object-oriented programming language like C++. Also like RPC, the encoding of parameters is hardware-independent, this time called CDR (Common Data Representation).

CDR differs from XDR in that it allows the data stream to use both big-endian and little-endian encoding for numerical values, so that no conversion is needed if sender and recipient have the same endianness, whereas XDR would always encode numbers on both ends if their hardware uses the "wrong" endianness.

Platform	Addressing	Synchronization	Encoding	Mobility	Interoperability
Sockets	no	no	no	no	no
PVM	yes	no	yes	no	yes
Erlang	yes	no	yes	no	no
Distributed Oz	yes	yes	yes	yes	no
RPC	no	yes	yes	no	yes
CORBA	yes	yes	yes	(yes)	yes

Table 2.1: Distribution Platform Feature Matrix

The phrase “Object-oriented RPC” nicely summarizes CORBA’s key ideas, as it was certainly designed with the experience gathered from RPC in mind. Yet it is an unfair understatement, ignoring many features and considerations that do not exist in RPC.

Still, CORBA is based on a Client/Server design. Clients handle objects, but the clients do not need to be objects by themselves: an object-oriented programming language and an object-oriented distribution platform do not imply object-oriented design as well [7].

Remote method invocations cause a request to be sent from the client to the server, and the client usually waits synchronously until the reply is received.

The CORBA location forwarding mechanism provides basic support for coarse-grained server mobility, but a vendor-specific forwarding service (an Implementation Repository) is necessary on the server side to employ the feature [15].

2.5 Evaluation

Each of the presented distribution platform comes with its own unique approach to distribution. Diverse as they are, they have in common that all are in use today, and new distributed applications are being implemented with them. As a summary, table 2.1 shows a matrix of the features provided by each distribution platform:

Addressing: Whether an abstraction of native networking addresses (host name and port number) exists.

Synchronization: Whether an explicit model of synchronization between the components of a distributed system is enforced. “No” means that the developer must take care of proper synchronization.

Encoding: Whether a common encoding of the transferred data is used, thereby abstracting the hardware.

Mobility: Whether the distribution platform allows mobile components.

Interoperability: Whether components can interact regardless of their hardware, operating system or programming language.

Quality of service is not shown for the reason that a distribution platform’s support cannot be measured with a simple yes or no.

One might argue that the distinction made between high-level distribution platforms and middleware in section 2.1 is fuzzy. It was argued that remoteness in a high-level distribution platform is transparent, and that seems just as true for client-side stub objects in CORBA.

Certainly, stub objects reach into the application layer. However, with CORBA, the developer is always well aware of holding no more than a stub object: there is a precise distinction that objects of a certain type can be either local or remote, but not both. From an academic point of view it might seem silly to point out the importance of the IDL file: objects whose interface is declared in IDL are *always* accessed using remote invocation semantics even if their implementation is local, and objects whose interface was not declared in IDL can *never* be remote. The potential remoteness of an object must be decided on at design time.

On the other hand, the Erlang or Distributed Oz developer can choose to make an object remote (or mobile) at implementation time, or even at runtime.

The difference between middleware and high-level distribution becomes more obvious on the server side. If Erlang or Distributed Oz were used to implement a Client/Server system in CORBA fashion, the server program would be the same as if integrated locally with the client. Using CORBA, the programmer cannot use the same code that would be used in a stand-alone, non-CORBA program, but the object implementation must inherit from a special skeleton, must follow a language mapping's rules for parameter passing, and the server program must negotiate with both the ORB and an object adapter before it can offer its services.

Server-side programming using middleware is quite different from both local programming and the server-side programming with a high-level distributed programming language.

The question for the "best" distribution platform is impossible to answer. Certainly using communication packages like sockets is cumbersome, but then, they're available everywhere and don't require third-party software, which can also be considered a plus. And some purists will always argue that "real performance" can only be achieved by directly using raw IP.

PVM has found its niche despite its shortcomings, and the same is true for high-level languages such as Erlang. Despite its advantages, high-level distribution will remain in a niche for as long as developers keep sticking to the programming paradigms they are used to, unwilling to commit themselves to a new programming language for various reasons: business, unflexibility or performance concerns.

This is the market exploited by middleware, which allows to use distribution while sticking to the programming environment which the developer is used to. Middleware might not be the best choice in an academic sense, but is a "common denominator" with many real-life advantages.

Chapter 3

Object Request Broker

The Object Management Group (OMG) is a consortium comprised of both companies and institutions, with more than 700 members. Founded in 1989, the focus of the OMG is the continued work on the Object Management Architecture (OMA), an open specification of an infrastructure for distributed objects [62].

The OMG's approach [40] is unique in that it does not publish products, but specifications which are based on the experience of its members. When the need for a new specification arises, a Request for Proposal (RFP) is issued. OMG members can then submit proposals which are discussed in a forum, improved upon, and finally published or rejected. While votes are taken only from members, the resulting documents are freely available on the Internet and can be inspected, commented on and adapted by everyone.

In this process the OMG tries to achieve useful compromises between versatility and complexity. Solutions that have already been implemented are preferred, to avoid overly academic specifications that would be hard or impossible to implement. On the other hand, the discussion and peer review among the many OMG members tries to ensure that such a solution is fit for general use rather than only solving the problems seen by the submitters themselves.

The basic structure of the Object Management Architecture is shown in figure 3.1. The building blocks of the OMA are objects belonging to one of four categories: system-oriented object services (for example, the Naming Service), horizontal common facilities (like collaboration), vertical domain-specific interfaces (like architectures for medical applications or air-traffic control systems) or application-specific interfaces [64].

The key component of the Object Management Architecture is the Object Request Broker, often called "object bus," which allows communication among these objects by directing remote method invocations from client to server.

According to the requirements for a distribution platform as described in section 2.1, means for addressing, synchronization and encoding must be provided. CORBA uses the middleware approach as presented in section 2.4.2 employing three main concepts:

Interface Definition: Fundamental to object-oriented programming, a client needs to know an object's public interface, so that it can invoke available methods with the appropriate parameters. Internal details are private to the object implementation and need not be described or published. In CORBA, interfaces are described using the *Interface Definition Language*, and information about interfaces can be stored in an *Interface Repository*.

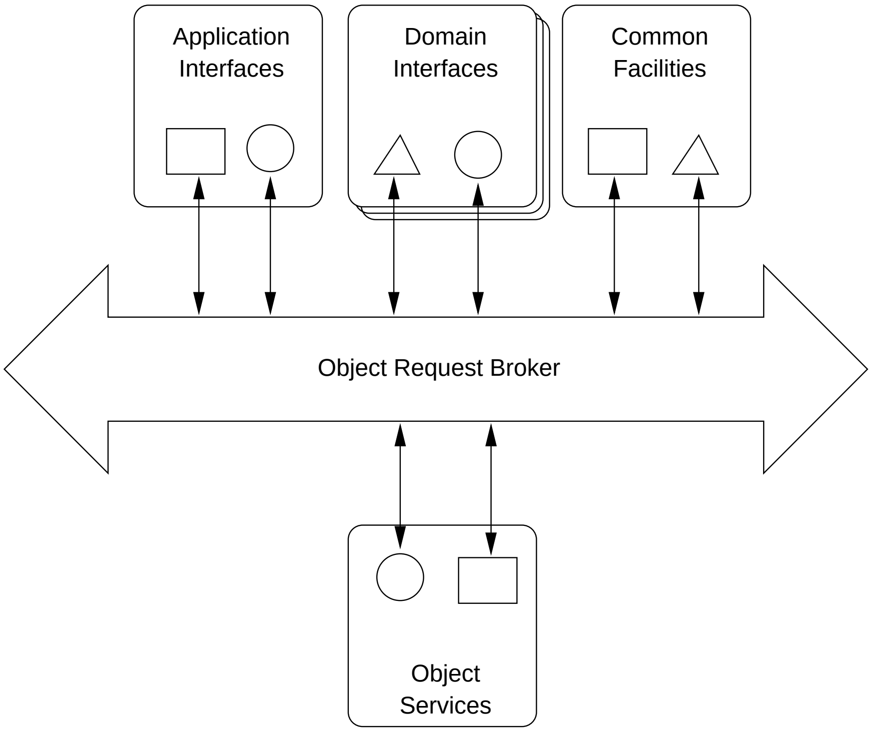

Figure 3.1: The Object Management Architecture Reference Model

Addressing: A client needs some kind of address so that it knows where to send requests to. This address must contain enough information to identify one particular *servant* – a piece of user-provided code – that the client wants to communicate with. CORBA uses *object references* to address an object.

Invocation: A client needs to be able to construct a request which is then transported to the server, evaluated, and returned. This can be done transparently with *stub objects*, which are generated from the interface definition. Invocations on a stub objects cause all parameters to be packaged into a *request* which is then transparently sent to the object implementation. Another option is the usage of the *Dynamic Invocation Interface* to build requests at runtime, possibly using runtime type information from the Interface Repository. Both means of invocation provide the required abstraction for synchronization and encoding.

Orthogonal to all of these three concepts is Interoperability, which, according to section 2.5, refers to the ability to communicate regardless of programming language, operating system or hardware.

3.1 Interface Definition

The Interface Definition Language (IDL) is used to describe an object's interface. IDL solely consists of declarative constructs, foremost of the "interface" keyword used to describe the

```
interface Account {
  exception Overdraft {
    unsigned long balance;
    unsigned long withdrawal;
  };
  void deposit (in unsigned long amount);
  void withdraw (in unsigned long amount) raises (Overdraft);
  long balance ();

  readonly attribute string HolderName;
};

interface Bank {
  Account create (in string HolderName);
};
```

Figure 3.2: Simple IDL File

methods[1] exported by an object and available to the client.

Interfaces are strongly typed, i.e. methods have a specific number of typed parameters and a typed return value. Parameters also have a direction: "in" parameters are sent from the client to the server, "out" parameters are received by the client as a result of the invocation, and "inout" parameters first get passed to the server, where they are potentially modified and then again returned.

Figure 3.2 shows the IDL declaration for an "Account" interface that provides two modifiers to deposit and withdraw, and one accessor to query the balance.

As mandated by the object-oriented programming model, IDL also provides interface inheritance: an interface can inherit from one or more base interfaces and can later, in programming, be used instead of a base type object (polymorphism). Whether the object implementation of a derived object also inherits the base object's implementation or re-implements the base interface's methods is up to the object implementation itself and not addressed by IDL: interface and implementation are kept separate.

IDL provides a number of basic numeric and string types. Additional language constructs allow the declaration of user-specific structured types, like structures, unions and enumerations. Noteworthy are the two basic data types "any," which contains a value of arbitrary type, and "object," which transports an untyped object reference.

Interface declarations are independent of a programming language. IDL files are then processed by an *IDL compiler* to produce the required declarations for the target language. How this is done is specified by a CORBA *language mapping*. For C++, two classes are generated: one abstract "skeleton" class for the server side to which the programmer must add the object implementation, and one "stub" class for the client side.

Type information from IDL files can also be kept online in one or more *Interface Repositories*, either for the purpose of constructing requests dynamically, or also for introspection

[1]CORBA refers to object methods as *operations*, to distinguish them from an object's *attributes*, which are but a special kind of operation. The terms method and operation can be used synonymously.

purposes. The interface of the Interface Repository is itself described in IDL; the contents of an Interface Repository can therefore be accessed through normal CORBA mechanisms.

3.2 Addressing

In order to access a remote object, a client first needs the servant's address. This information is contained in *object references.* While in older revisions the handling of object references was an implementation detail, they have been standardized in CORBA 2.0 in a successful effort to strive for interoperability among different ORBs. These *Interoperable Object References* (IOR) are opaque to the developer: no information can be extracted from an IOR, and neither can an IOR be composed from known components.

It is important to note that the lifetime of an object reference is independent of the lifetime of its server. A servant is not notified of the creation or destruction of its object references (see the "Pacific Ocean" problem in [15]), and a client does not have a means of checking if the server is running save by performing an invocation – which will fail if the server is down and cannot be restarted.

Although the knowledge of a running server might be thought important to the clients, this inability is quite intentional, as it allows a transparent restart of the server without a client noticing. A server might be temporarily shut down to conserve resources or for replacement, and later be restarted on demand. Clients continuously asking "are you asleep yet" would permanently prevent a server from shutting down.

The information whether a server is still active is useless anyway, as such information would always be old: during the time it takes to receive the reply, the server could crash or be stopped just as well.

Even though the Internet is today's most common CORBA environment, other means of transport are allowed, too. Object references contain one or more *profiles*, each one an appropriate identifier to locate the object's implementation – the Internet-specific IIOP (Internet Inter-ORB protocol) profile is but one option.

An object reference with more than one profile can make sense for redundancy purposes (two profiles can point to mirrored objects on different hosts), for quality of service purposes (one preferred profile for encrypted transport) or for optimization (one profile for local inter-process transport if the server is on the same host as the client, and another profile for remote transport).

To learn which information is contained in an object reference, it is useful to write down a formal definition of an IOR. An IOR $I = (t, \{p_i\})$ is a two-tuple with an optional IDL-declared type t and a set of profiles p_i. Profiles have different layouts; as an example, an IIOP profile $P_{IIOP} = (v, h, n, k, o)$ can be further decomposed into five components, one fixed IIOP revision number v (at the moment of writing, 1.2), the Internet host name (or dotted decimal IP address) h, a TCP port number n that the server is listening on, a server-specific opaque *object key* k and optional "tagged components" o[2].

Their opaque nature poses a bootstrap problem for a client, which must acquire an object reference, but cannot compose one by itself. One solution is the exchange of *stringified* object references, which are more or less a hex dump of the opaque data. The servant generates an

[2]Tagged components can carry manifold additional service information, like the native charset to be used for string values. They are in itself opaque, their contents defined by a component's "tag," which identifies them.

```
IOR:010000000d00000049444c3a42616e6b3a312e3000000000020000000000000000
20000000010100000c000000726f73652e6670782e646500a8300000040000004261
6e6b0100000024000000010000000100000001000000140000000100000001000100
000000000901010000000000
```

Figure 3.3: Stringified Interoperable Object Reference

```
    Repo Id:  IDL:Bank:1.0

IIOP Profile
    Version:  1.0
    Address:  inet:rose.fpx.de:12456
   Location:  iioploc://rose.fpx.de:12456/Bank
        Key:  42 61 6e 6b                                 Bank

Multiple Components Profile
 Components:  Native Codesets:
              normal: ISO 8859-1:1987; Latin Alphabet No. 1
                wide: ISO/IEC 10646-1:1993; UTF-16
        Key:  00                                          .
```

Figure 3.4: Contents of the Object Reference shown in Figure 3.3

object reference to itself, and writes its stringification into a file, which can be either on a shared volume or manually transported to the client using e-mail, networked file transfer or a sneakers network. The client then reads the file and uses the inverse transformation to receive the original object reference. A sample stringified IOR is shown in figure 3.3, and figure 3.4 displays the contents of that object reference, interpreted and made visible using a special `iordump` tool.

This way of bootstrapping using "unreadable" strings is rather ineffective, especially if not only one, but many object references must be exchanged. The Naming Service, one of the aforementioned common object services, has been introduced to reduce this problem. Servers can register their objects with the Naming Service, where they can be retrieved by the client using a more natural name. Yet servers and clients still need an object reference to the Naming Service itself.

Unfortunately, this common problem and frequently asked question of bootstrapping has not been answered satisfactorily until recently. CORBA 2.3 will introduce a new, more "human-readable" form of object references as part of the *Interoperable Naming Service* specification [24] that follows the same syntax as the common Uniform Resource Identifiers of the World Wide Web [5], while being limited to objects with only IIOP profiles.

As seen in the formal definition above, the minimal IOR consists of three parts: a host name and a port number as part of the IIOP profile, and a server-selected object key - the version number can be fixed, and both the type information and the tagged components are optional. So the new scheme for IIOP-based object references consists of these three parts written in plain text:

`iioploc://<hostname:port>/NameService`

While this notation is useful and convenient, it already requires careful handling of the object key by the object adapter, which should allow for sensible object keys as not to thwart this new scheme with illegible object keys.

An object reference, be it Internet-based or not, need not be final: GIOP (see section 3.4) allows request forwarding, where an address does not point to the actual servant, but to a forwarding service. This may be useful if the server may change its location (for example, if it is shut down to save resources or for replacement and then started again), as long as the forwarding service keeps running.

In response to frequently used terminology, it should be noted that there are no "persistent" or "transient" object references, as mentioned in some literature or in Internet discussions. There is only one kind of object reference, and while they may point to persistent or transient objects,[3] this cannot be determined from the reference without internal knowledge of the server-side ORB. Some ORBs indeed use specific tags in references to persistent or transient objects and use the information for internal purposes, but the client does not know.

3.3 Invocation

Once both the interface and the address of an object are known, methods can be invoked and will be executed remotely. Again, how this is done is up to the language mapping.

In C++, special *stubs* are generated by the IDL compiler for each interface. These are C++ classes that export the methods declared in the IDL file. In addition, an instance of a stub, a stub object, encapsulates a specific object reference.

Whenever a C++ program acquires an object reference, decoded from a stringified object reference, from the Naming Service or as result from another method invocation, the ORB indeed creates a stub object. Since a stub object cannot exist without an object reference, and an object reference rarely exists outside a stub object, these two terms are often used synonymously.

The generated code in the stub takes care of *marshalling* the method's parameters into a request and passes it on to the ORB, which deciphers the object reference and contacts the server. The remoteness is transparent to the user program: the invocation looks no different than a normal, local invocation.

For the server side, a *skeleton* is generated by the IDL compiler. The user derives from that skeleton and adds the methods' implementation; an instance of such an object implementation class is called *servant*. The generated skeleton takes care of receiving the request from the ORB, unmarshalling the parameters and performing the upcall into the user-provided code. This way, the object implementation also looks like a "normal" class. The full process is shown in figure 3.5. It shows the ORB to be a component of its own, which is true in the abstract sense only. Usually, it is implemented partly in the client and the server – but the user does not have to be aware of such details.

Stubs and skeletons, as described so far, use static type information: the method names and their parameters must be decided on at compile time, for they are hardwired into the generated code. Alternatively, the Dynamic Invocation Interface (DII) is available on the client side, and the Dynamic Skeleton Interface (DSI) on the server side. Using the DII, a client can compose a request using runtime type information, retrieved for example from

[3] The persistence or transience of an object is an Object Adapter concept and explained in the next chapter.

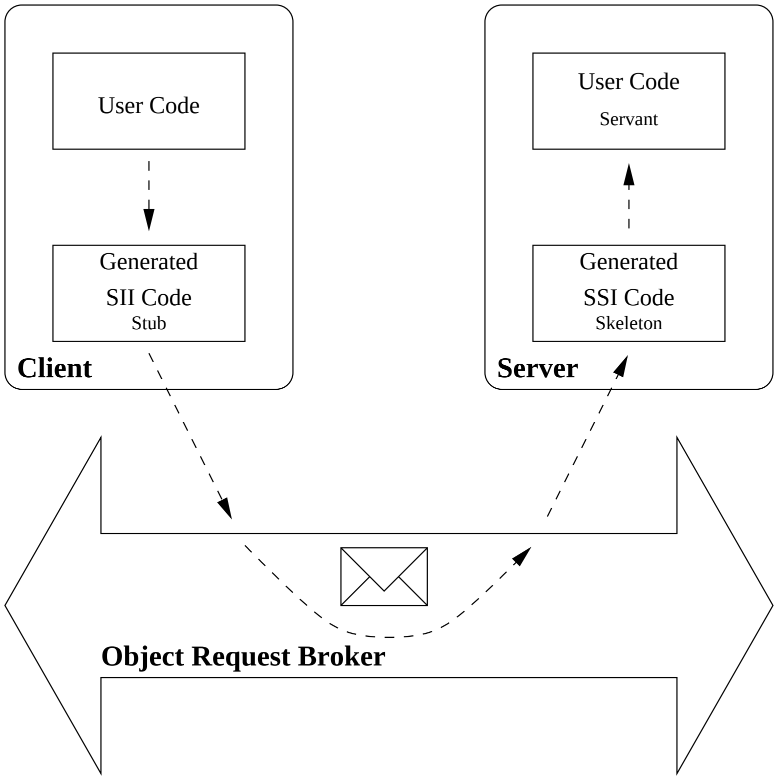

Figure 3.5: Performing a CORBA invocation

an Interface Repository. On the server side, a servant can use the DSI to dissect incoming requests explicitly.

Using the DII on the client side does not require using the DSI on the server side and vice versa. The requests that are sent by DSI or received by DII do not differ from other requests, as both interfaces are used only to perform the work usually done by generated code.

DII and DSI are not only useful for gateways that do not have advance knowledge of the requests they are supposed to forward. Dynamic invocations can also be used in generic front ends, to invoke methods for which no generated code is available, and dynamic skeletons can be used, for example, to support more than one interface with a single servant.

Another feature of the DII is that it allows asynchrony. Invocations through the static interface are always synchronous, the client blocks until the request is processed and answered by the server. The DII provides methods for sending a request asynchronously, and later for polling or waiting for the reply.[4]

On the other hand, static IDL-generated code, apart from being much more comfortable to use, is likely to be faster in constructing requests than code using the dynamic interfaces, which must use generic interfaces for marshalling parameters, such as encapsulating each parameter in an `Any` value. IDL-generated code, produced by a vendor-specific IDL compiler, can take full advantage of a vendor-specific *Static Invocation Interface* (SII) if available, which might be able to marshal data much more effectively. This smaller request overhead is especially noticeable with large and complex parameters (such as arrays or sequences) if the server-side

[4]This is not exactly *required* by the DII specification. A compliant ORB may block after sending the request until the reply has arrived, thus defying asynchrony.

computation time is small.

As seen in section 2.2.2, the Parallel Virtual Machine provided asynchronous messaging *only* – asynchrony is essential for the implementation of parallel systems. In a Master/Slave system, a master distributes a number of small tasks to slaves running in parallel, and must then wait asynchronously for the results to arrive. CORBA Messaging [35], scheduled for CORBA 3.0 [71], will introduce a new feature called Asynchronous Method Invocations (AMI), which allows asynchronous invocations based on IDL-generated skeleton code.

3.4 Interoperability

CORBA 2.0 also specifies the *General Inter-ORB Protocol* (GIOP) as part of its ORB Interoperability Architecture. It describes the format for requests, replies, and the representation of data types "on the wire," when transported from one ORB to another as part of a method invocation. It also takes into account several low-level issues, such as the format of floating point values, data alignment and endianness for integral values.

Older versions of CORBA (prior to 2.0) did not address the issue of inter-ORB data transfer, so that ORB vendors had to invent a data format on their own,[5] making inter-vendor communication impossible.

GIOP is an abstract specification itself, requiring only a reliable (no data loss) bidirectional data stream, so that messages can be passed among the ORBs. The aforementioned Internet Inter-ORB Protocol (IIOP) is a specialization of GIOP in that it implements a concrete profile using host name and port number, and a concrete transport mechanism, TCP/IP.

The Security Service [36] describes, among other security-related issues, an implementation of GIOP over SSL (Secure Socket Layer) to provide secure, encrypted transport.

Using GIOP, true interoperability is achieved regardless of operating system or hardware. Since the data stream is agreed on, any programming language that can read and decipher GIOP data can access or implement CORBA objects. Official language mappings for C, C++, Java, Smalltalk, Ada and COBOL are part of the specification [28]. Efforts are underway to provide mappings for scripting languages such as Perl, Tcl or Python. The list of other programming languages that have been used for CORBA programming is long, including Eiffel, Lisp and Basic. An Intercal mapping is planned [52].

This kind of transparency and heterogeneity is unique and makes CORBA an interesting choice as distribution platform.

3.5 Evaluation

After presenting the ideas of CORBA and praising its virtues, it is also important to recognize its weaknesses, what CORBA cannot do.

Like RPC, CORBA is foremost a Client/Server system, based on a sequential, synchronous request-response model of communication. Of the four design strategies for distributed systems mentioned in 2.1, CORBA supports only Client/Server delegation and resource sharing.

Parallel processing requires asynchrony, to distribute many tasks to the worker nodes simultaneously, and to react upon their results as they become available. This could be achieved, even with current means, using asynchronous DII, at the cost of having to construct

[5] One example is POOP, the "Plain Old Orbix Protocol," still available in current Orbix versions for backward compatibility and, of course, for "increased efficiency."

requests manually and using a polling model to wait for results, or using callbacks and event-based programming [56]. Messaging [35] will improve in this area, providing asynchronous method invocations and callbacks.

Fault tolerance is not supported either, but can again be simulated using asynchronous DII. An effort to add fault tolerance by automating the multicast of requests to redundant servers, collecting and evaluating their responses is presented in [20].

Orthogonal to these three strategies are real-time requirements, such as latency, guaranteed response times or precise timing of request delivery. Design considerations for real-time ORBs are presented in [51]; an RFP about real-time issues has been answered by a joint submission [37] and is scheduled for CORBA 3.0. Precise timing will require the mapping of GIOP to a real-time aware transport mechanism like ATM (see section 2.2.1).

Somewhat dependent on real-time capabilities are data streams for audio or video data, which also don't fit into request-response communication. Again, the OMG is discussing possibilities for integrating real-time data streams.

An approach for a more generic quality of service infrastructure by allowing the negotiation of user-defined quality of service issues is presented by the Management Architecture for Quality of Service [4].

It seems fascinating to look into CORBA's future with its many intentions, considerations and forthcoming additions, but the feasibility should not be left behind. From looking at the stacks of paper, the CORBA specification exhibits almost exponential growth. From CORBA 2.0 [27] (July 1995) to CORBA 2.2 [28] (February 1998) alone, the core specification (excluding language mappings and common services) was extended by six chapters and doubled from 233 to 489 pages. CORBA 2.3 will again bring many extensions and two new chapters; the preliminary specification [30] (December 1998) totals 579 pages.

While these were *minor* revisions, adding features one at a time, the 3.0 update will be major, introducing Components, Real-time CORBA, Minimum CORBA, Messaging and Firewalling [23].

Consequently, the Object Request Broker becomes increasingly complex, and the vendor's task of keeping an ORB up to date with the specification and to verify its correctness is becoming harder still. On the user end, the entry into CORBA programming will become more difficult, too, as the possibilities become harder to choose from. With Components, Objects by Value and interfaces, CORBA 3.0 will offer three non-orthogonal paradigms for object implementation.

This progress is certainly due to CORBA's increasing popularity. After CORBA 2.0 ORBs were readily available, they have been put to use in many different environments from embedded systems to large-scale parallel architectures. It is under these extreme circumstances that shortcomings surface.

Maybe it would be prudent to limit the growth somewhat, and instead of extending CORBA in all directions simultaneously, to emphasize and improve upon the abilities that are readily available.

More features also cause the language mappings to grow. Between 2.0 and 2.3, the C++ mapping doubled from 94 to 174 pages, too, growing more complicated with more special cases to obey – the mapping of Objects by Value to C++ [31] is, in one simple word, a nightmare. The C++ mapping is a good candidate for revision from scratch. It still suffers from historical deficiencies in compiler implementations: before 1995, the preliminary ANSI C++ standard was still a moving target. Not using the Standard Template Library – decent implementations have not been available back then – and governed by performance considerations, the mapping contains many shortcuts and different strategies for passing parameters by value vs. by reference: for example, distinguishing “fixed length” and “variable length” structures is bound to cause confusion. It would be interesting to see a revised, straightforward C++ language mapping consequently based on STL containers and classes [22].

The goal to adapt only proven specifications is also not always enforced, and some parts of the CORBA specification have been found to be ill-designed or impossible to implement. One example is the Persistent Object State Service [39]. Well after its addition to the set of common object services, it was found to be “too complex and imprecisely specified” [38], and a new RFP has been issued, euphemistically proposing to apply the “OMG sunset policy” [40] to the original service.

The Basic Object Adapter has also been a source of confusion because of its design flaws.

Chapter 4

Object Adapters

Using CORBA-based services is rather simple and straightforward: After receiving and translating the IDL file into the target language, all that is needed is an object reference. Henceforth, methods on the remote object can be called through the stub object as if they were local.

More thought must be spent on the server-side, where servants are to be provided. Even after registering the actual program code with the ORB – for example by passing C++ objects – the programmer will usually want to have a certain control over ORB processing, such as when a method may be executed. Missing control on the server side is also a frequent complaint about RPC (see section 2.4.1), where a server is simply expected to run in perpetuity, not having any control over the execution of incoming requests.

The OMG concluded that an implementation's requirements are highly dependent on the operating system, programming language, and environment. For the sake of an example, the needs of a server representing an object database would be very different from a server providing a printing service.

In order to allow for different possibilities, the handling of actual user code, servants, is decoupled from normal ORB processing by the introduction of *object adapters*. An object adapter serves as mediator between the ORB and the servant, providing the ORB with a consistent interface for interacting with user code, and being flexible in cooperating with the servant. Different object adapters could then be provided to suit the various server requirements, and an implementation could choose between them and select the most appropriate.

Gamma, Helm, Johnson and Vlissides [10] define an object adapter as a design pattern to achieve a "reusable class that cooperates with unrelated or unforeseen classes." According to their definition, an object adapter receives a client's request and translates it into a request that is understood by the "adaptee," the servant. This description matches that of a CORBA object adapter [57] quite well. The basic task of an object adapter is a simple one of dispatching an incoming request to user code, the servant.

When an invocation is made, the client-side ORB is responsible for interpreting the object's profiles, for locating the server in which the object is implemented, and for sending a request to that server. On the server side, the request is received by the ORB, where three steps of dispatching are necessary:

1. The ORB must find the object adapter that the object is implemented in and pass the request on to that object adapter.

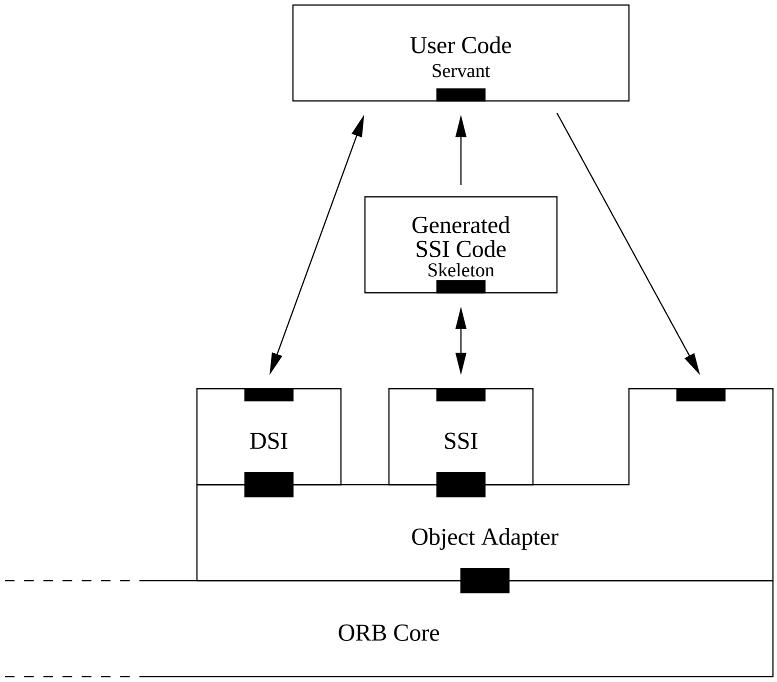

Figure 4.1: Server-side view of the ORB

2. The object adapter must find the servant that implements the object.
3. If the servant uses a static skeleton, the request is unpacked by the IDL-generated code and the desired method is invoked.

But before any of this can happen, the object adapter must first know about the servant. After registering the servant with the object adapter, an implementation must be able to create and export object references that address the servant. So beside merely performing invocations, the object adapter must provide an administrative interface as well. The OMG lists the following white-paper responsibilities [25] of an object adapter:

- registration of implementations
- mapping object references to the corresponding object implementations
- object and implementation activation and deactivation
- generation and interpretation of object references
- method invocation
- security of interactions

The first five items have already been mentioned; the security of interactions is only historical, as it is not a server-side issue only. Security must be an integral part of ORB processing and is now addressed by the Security Service [36] instead.

Figure 4.1 shows the object adapter in relation to the other server-side parts of an ORB. The object adapter is sandwiched between the ORB core and the object implementation and

```
interface BOA {
  Object create (in ReferenceData id,
                 in InterfaceDef intf,
                 in ImplementationDef impl);
  void dispose (in Object obj);

  ReferenceData get_id (in Object obj);

  void change_implementation (in Object obj,
                              in ImplementationDef impl);

  void impl_is_ready (in ImplementationDef impl);
  void deactivate_impl (in ImplementationDef impl);
  void obj_is_ready (in Object obj,
                     in ImplementationDef impl);
  void deactivate_obj (in Object obj);
};
```

Figure 4.2: IDL for the Basic Object Adapter

has to provide three interfaces: one to the ORB, which consists of a single method to receive an incoming request, one to the user code to pass this request to, either using the Dynamic or Static Skeleton Interface, and one administrative interface through which an implementation can cause objects to be activated or deactivated, and can influence the processing of requests.

The most simple object adapter would be no more than a table that maps object keys (the server-side part of an object reference) to servants. Upon invocation, only a single table lookup would be necessary, and activation and deactivation of servants would cause insertions and deletions into that table.

In fact, the Basic Object Adapter (BOA), the first and until recently only object adapter specified by the OMG, is little more than that. As the name suggests, it does not provide much but basic functionality, and it was envisioned that a range of further, more specific object adapters would be added later on.

An object adapter has a certain impact on the programming style to be used, as each brings its own concepts and enforces a concrete structure for servants that an implementation has to submit to. Apart from the very generic description above, object adapters do not have a common behavior, but have to be considered individually.

4.1 Basic Object Adapter

The Basic Object Adapter was introduced with the first version of CORBA in 1991 and has been largely unchanged since then. Its purpose was to be a simple and generic type of object adapter that could be used, as the name suggests, for basic purposes. As such, its interface is intentionally minimalistic, providing a total of 10 methods, conveniently specified in IDL as shown in figure 4.2.

The Basic Object Adapter defines three states that an object can enter during its lifetime:

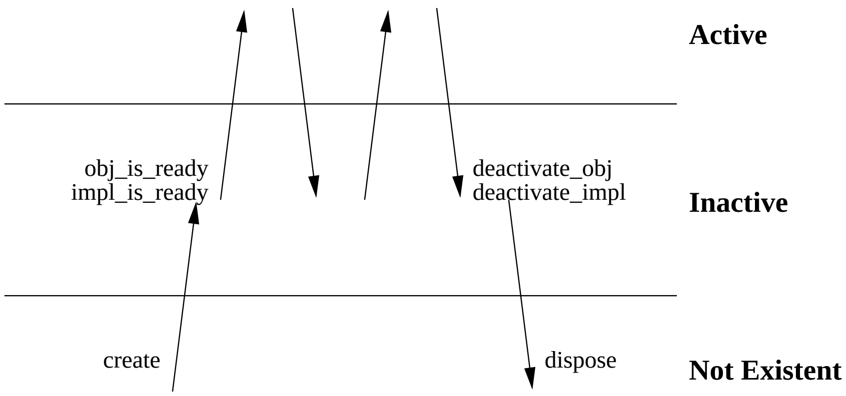

Figure 4.3: Lifecycle for objects using the Basic Object Adapter

not existent, inactive, and active. Not surprisingly, an object is initially in the "not existent" state, meaning that the ORB does not know of the object and that invocations are not possible but rejected with an appropriate error message.

The "creation" calls an object into being. From that point on, object references can be exported. Clients can receive object references for inactive objects, but method invocations will be withheld on the server side by the BOA and block until the object transitions to the active state. After activation, method invocations received by the ORB core are passed on to the implementation until it is deactivated again.

The process of object activation and deactivation can happen more than once and is transparent to the client. A server might wish to disallow upcalls from the ORB for some time while other tasks are done, or to replace the implementation.

Once an object is not needed anymore, it can be destroyed to return to the not existent state, causing the BOA to act as if it had never existed. Destruction is final; although the BOA's interface would allow for the reincarnation of an object – the creation of a new object that answers to the same object references as before, this is forbidden by the specification, which said that two creations, even with the same parameters, will never produce the same object.

Another central concept introduced by the Basic Object Adapter is the "Implementation Repository." Corresponding to the Interface Repository as a database for the abstract declaration of an object's interface, the Implementation Repository was to be a database of program code for an object's implementation. As seen before, an IOR contains both a type and an address to an implementation. The BOA employs this idea; object creation is done by associating an interface with an implementation using the `create` method.

The BOA is aware of processes and servers, and is designed to start up new servers by itself if necessary. One information kept in the Implementation Repository is the *activation policy,* so termed because it determines when a new server must be activated in order to serve a request.

Shared Server A shared server may implement many objects of the same or of different type at once. A shared server is started by the BOA only if it is not already running. Otherwise, the running server is contacted for incoming requests.

Unshared Server An unshared server implements exactly one object. The activation of an object causes the starting of a new server.

Server per Method A new server is started for each incoming request.

Persistent Server A special case of a shared server. Persistent servers, too, can implement many objects at the same time. However, persistent servers are not started by the BOA, but are to be started by user code, or by administrative action.

After start-up and initialization, shared servers and persistent servers will call `impl_is_ready` to activate all objects they implement, an unshared server calls `obj_is_ready` instead. Deactivation is done by calling `deactivate_impl` and `deactivate_obj`, respectively.

The actual contents of the Implementation Repository are not specified by the CORBA standard but must be defined by the ORB implementation. As per the text above, an entry must contain at least the activation policy and information about a server process (e.g. a path name to an executable program).

4.1.1 Shortcomings

From the specification, some implementation details can be deduced. Seen from a process-oriented perspective, the BOA must be realized in two parts. One part must be included in each server to receive requests and to perform upcalls into the object implementations. The other part has to keep track of all implementations in the Implementation Repository and to start new servers if necessary. This second part could be realized either as a permanently running daemon or be integrated in each client.

It is interesting that some over-enthusiastic literature [44] cannot find fault with the BOA and coin it a "fairly flexible adapter" [63]. But problems become obvious when actual servers are to be written according to the specification. As seen in figure 4.2, the `create` operation is used to call an object into existence by associating an interface with an implementation. An Implementation Repository holds information about the server program, but no user code. Once the BOA has started a new process in order to activate an object, the BOA needs to perform upcalls into user code to perform invocations – but the BOA does not provide a means of registering servants. Information about servants cannot be kept in the Implementation Repository itself, for they only exist in a live server (for example, C++ objects). Already in this very important respect, each ORB vendor must use proprietary additions to the standard in order to get the BOA to work.

This error is not mended with the simple addition of one or more methods. Rather, the standard did not even provide a basic structure for server-side programming. While it requires the IDL compiler to generate skeletons from which an object implementation is to be derived, their inheritance, behavior and even name are undefined.

More problems arise when considering an object's lifetime, which can exceed the lifetime of the server it was created in. Objects may need persistent storage for their internal state while a server is not running. When a server is started by the BOA, the program cannot determine which objects are to be restored. Only within a method, during execution of a request, can a program determine the object reference which is the target of the request[1] – at which point it is too late, or rather very inconvenient, to restore state data.

If an implementation tries to take saving and restoration of objects into its own hands, it is likely to run into a race condition. There's no point in time at which the data could be

[1] In the C++ language mapping, the skeleton provides a `_this` method.

written, because the transition from the active to the inactive state is instantaneous to the BOA. Before deactivation, requests are still delivered and state data may change. After calling `deactivate_impl`, the BOA may immediately start up a new server before the old server has finished writing its data.

Once the problems of registration and state restoration are taken care of, a server runs into another omission. After initialization, a server needs to wait for incoming requests, but the specification does not define how this is done. Some BOA implementations make `impl_is_ready` and `obj_is_ready` block and wait for requests until `deactivate_impl` or `deactivate_obj` is called. While this decision is sensible, the process of waiting for incoming requests it is not referred to in the documentation.

In summary, the problems with the BOA are numerous and can be put into three categories.

1. Design omissions that make the BOA specification unfit to work as is:
    - The contents of the Implementation Repository are unspecified.
    - The layout or name of skeleton classes is unspecified.
    - Servant registration is unspecified.
    - Request processing is unspecified.
2. Design problems that are not fatal but inconvenient:
    - No interface or hooks for saving or restoring object state.
    - The distinction of being "active" or "inactive" is too coarse. A third state is needed for server processing unhindered by incoming requests or the starting of a new implementation.
    - The ties between objects and processes as defined by the activation policy are too limiting. A server should have more control over when, why and how new servers are to be started. Unshared and per-method servers are rarely needed.
3. Unadressed issues causing problems in more complex environments:
    - Handling of object groups, where a number of similar objects are served by the same user code.
    - Request to thread assignment; in a multithreaded environment, some servers will prefer multithreading to continue request processing while other lengthy method invocations are in progress, others are not prepared for multithreading.

While the third category is easily explained by the "basic" nature of the Basic Object Adapter, the others require fixing by the ORB vendor. However, since these fixes are outside the specification, they introduce incompatibilities. Portable server programming, so that server code is compatible among ORB vendors, is impossible, because a program must rely on the vendor's custom addition to the standard.

Many vendors have chosen to make usage of the BOA as transparent as possible: registration is performed in a servant's constructor, and creation of object references is made transparent by skeletons inheriting their own object reference. Some vendors have chosen to abandon the Implementation Repository completely and only provide the persistent activation policy, like ORBacus [41].

This way, the existence of the BOA is mostly hidden from the developer, and calls into the BOA like `impl_is_ready` are left over as part of initialization "magic," unrelated to the original intention of object adapter programming.

4.1.2 Solutions

The Basic Object Adapter has failed twice. It was never designed to be a general-purpose object adapter, but only intended as a first option. The OMG expected more specialized object adapters to appear over time. Their reasoning was that no all-powerful object adapter was achievable, as each brings its own design limits, so that object adapters would have to be tailored to specific purposes.

But even in this respect, the BOA did not fulfill its expectations because of its severe design flaws.

The combination of both points did have the unfortunate side effect that since ORB vendors did have to tamper with the BOA specification to make it work in the first place, they did not finish with incompatible but still recognizable Basic Object Adapters, but extended their BOA adaptation with new features as well, like synchronization, threading primitives, or an interface for storage and restoration of persistent data.

In this the vendors left the intended path of the OMG. The most effective way of dealing with an insufficient BOA would have been to introduce an object adapter of their own design. Rather, CORBA was now stuck with many incompatible object adapters that still called themselves "BOA," but that did come, apart from the necessary fixes, with individual custom features.

The Object Management Group recognized the impossibility to write portable server code, and as a result issued the "ORB Portability Enhancement RFP" [25] in June 1995, in parallel to the publication of the second version of the CORBA standard. It called for ideas on how to improve the situation, naming four possible solutions:

1. Improvement of the existing BOA specification to eliminate the need for interpretation.
2. Multiple versions of the BOA for different environments (POSIX, Real-Time, Embedded etc.).
3. A new "universal" object adapter.
4. Eliminating the standardization of object adapters .

As a result of this effort, the "ORB Portability Joint Submission" was presented in May 1997 by a number of OMG member companies, foremost BEA Systems, Inc. This submission, set in stone with the release of the CORBA 2.2 specification in February 1998, indeed introduced the new, universal Portable Object Adapter (POA).

The final reasoning of the OMG was that the problems with the BOA were too numerous to be fixed with minor adjustment, and that any changes would break existing server code that based on proprietary BOA extensions.

By applying the "sunset policy" and removing the BOA specification entirely, the BOA did after all become a vendor-specific extension, allowing existing server programs to live on unchanged, but with the definite tag of being unportable.

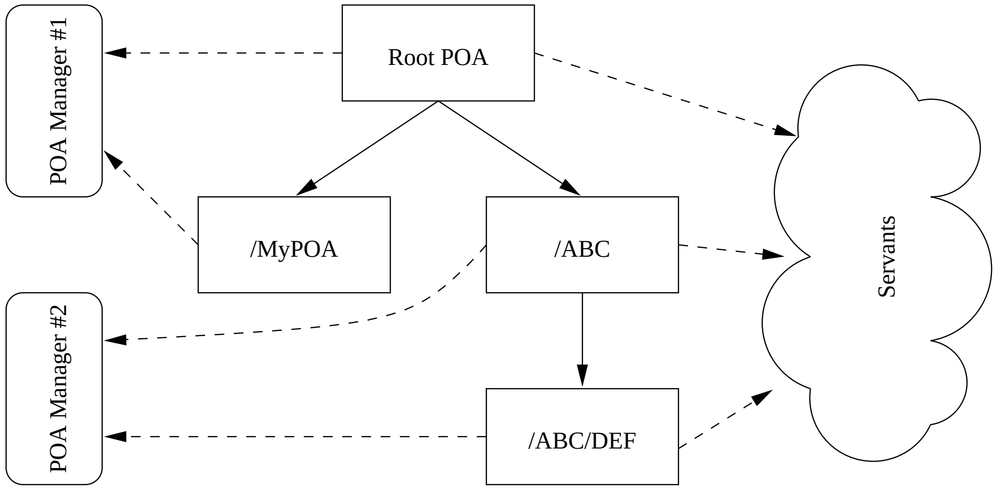

Figure 4.4: Many POA instances can exist in a server

The newly introduced Portable Object Adapter was set in its place as the new "one size fits all" general-purpose object adapter. However, when the ORB Portability Joint Submission was released, it once again violated the OMG's policy of adopting only existing technology. While the original RFP stated that "submitters are expected to have working implementations of their proposed specifications" at the time of initial submission, the final submission still mentions that only "some prototyping and detailed design work" had been done, and offers the vague hope that "products based on this design will be available within the OMG's timelines."

When the CORBA 2.2 specification was released, this had not happened.

4.2 Portable Object Adapter

Compared with the ten pages of BOA specification in CORBA 2.0, the 63 pages for the Portable Object Adapter are impressive. Indeed it is not a simple replacement as a new "basic" object adapter, it is rather designed to be universal. Its two foremost design goals are

- Portability. It should be possible to use source code with a different vendor's ORB without source changes.
- Flexibility. By providing hooks to control a servant's life cycle and readiness to receive requests, the POA should be usable even for servers with exotic requirements. Sensible defaults exist to make the programming of simple servers as easy as possible.

Again, the POA comes with its own set of concepts. Unlike the BOA, it is not a singleton component [10]. Many POA instances can exist in a server, organized in a hierarchical structure. The *Root POA* is created by the ORB, new POAs can then be created by the user as children of existing ones.

Each POA maintains its own *Active Object Map*, a table mapping the currently active objects to servants. Objects are activated within a particular POA instance, and henceforth associated with "their" POA, identified by a unique *Object Id* within its namespace.

Synchronization between POAs is achieved by *POA Managers*, which control the readiness of one or more POAs to receive requests. Figure 4.4 shows an example of using many POAs:

the Root POA has two children, and the POA named `ABC` has another child, with which it shares a separate POA Manager. Servants can be registered with any of the four POAs.

Apart from having control over synchronization, the POA provides many hooks that enable a user to influence request processing:

- The life cycle of servants can be controlled and monitored by *Servant Managers*.
- *Default Servants* can service many objects at once.
- *Adapter Activators* can be used to create new POAs if necessary.

The following sections will examine the POA's key concepts more closely.

4.2.1 Policies

Each POA has its own separate set of *policies*, which adjust different aspects of a POA's behavior. Policies are selected by the user upon POA creation and cannot be changed over its lifetime. Since objects are associated with a fixed POA instance, some policies can also be said to be that of an object's, most obvious with the lifespan policy.

Thread Policy Addresses multithreading issues. Can be set to either "Single Thread" if the servants are not thread-aware and that requests must be serialized, or to "ORB controlled" if servants are reentrant and requests can be processed regardless of ongoing invocations.

Lifespan Policy Can be either "transient" or "persistent." The lifespan of transient objects (i.e. objects that are registered in a POA with the transient lifespan policy) is limited by the lifespan of the POA instance they were activated in.[2]Once their POA is destroyed, for example as part of server shutdown, object references to transient objects become permanently invalid. Persistent objects can outlive their original POA and even their original server. Missing POAs for persistent objects can be recreated, and if their server is shut down, it may be restarted at a later time and continue serving the object.
The usage of the term "persistency" here is very different from its meaning as a BOA activation policy. All BOA servers satisfy the POA's persistent lifespan policy, because servers could be stopped and restarted.

Object Id Uniqueness Policy Slightly misnamed, "Servant Uniqueness" might be more appropriate, as it controls whether a single servant can be registered (activated) with the POA more than once to serve more than one object.

Id Assignment Policy Defines whether Object Ids are generated by the POA, or selected by the user, for example to associate objects with "human-readable" names or to hold identity information: if a single servant is registered more than once to serve multiple objects, it could use a user-selected Object Id (which would be different for multiple activations) at runtime to discriminate between them.

[2]Note: This was changed in CORBA 2.3. According to CORBA 2.2, a transient object's lifespan is limited by the lifespan of its *process*.

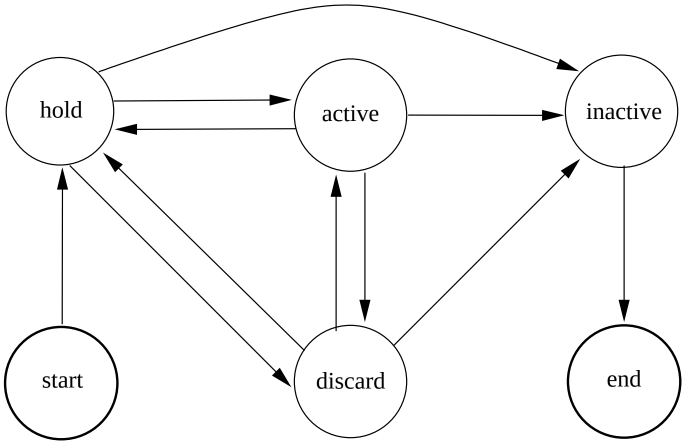

Figure 4.5: POA Manager State Machine

Servant Retention Policy Normally, when an object is activated, the association between the object (or rather, Object Id) and the servant is stored in the Active Object Map. This behavior can be changed if desired, for example if a default servant is prepared to handle requests for newly-activated objects.

Request Processing Policy Specifies if the POA will only consult its Active Object Map when trying to serve a request, whether a servant manager is available, or whether the request should be delegated to a default servant.

Implicit Activation Policy Whether servants can be activated implicitly. Some operations on a servant require its activation, for example the request of an object reference. Depending on this policy, performing such an operation on an inactive servant can either cause an error, or transparent activation.

Certainly, some policies are of more interest than others. The implicit activation policy or the Object Id uniqueness policy can be considered sweeteners; they do not provide actual behavior but are mere safeguards against the careless developer. Implicit activation saves but a line of code that would be necessary with the enforcement of explicit activation.

The Id assignment policy can relieve the developer of the need to generate unique names. While this feature is also redundant, it is nonetheless useful, as the Object Id is not of interest in many simple servers, where automated selection of a unique Id saves a lot of unnecessary code.

The other policies are more crucial and will be referred to again in later sections.

4.2.2 POA Managers

One problem with the BOA was that no mechanism existed to synchronize servants, or to control a servant's readiness to receive requests: as soon as an object was activated, invocations were performed, and as soon as it was deactivated, a new server would be started by the BOA.

The POA provides "POA Managers" for this purpose. A POA Manager is a finite state machine associated with one or more POA instances that can enter one of four states, with state transitions as shown in figure 4.5.

Active This state indicates normal processing; incoming requests are dispatched to the servant.

Holding Incoming requests are not processed immediately but queued. When the holding state is left, queued requests are handled according to the new state: e.g. if the active state is entered, delayed requests are dispatched.

Discarding Requests are discarded, and the client receives a "transient" exception, indicating a temporary failure. Useful for example in real-time environments instead of the holding state, when the queuing of requests and their delayed handling is not sensible.

Inactive This state is entered prior to the destruction of associated POAs and cannot be left. Incoming requests are rejected, indicating permanent failure. With the POA Manager in the inactive state, a server can perform cleanup work, such as storing objects' persistent state.

For simplicity, it can be said that a POA is active if the POA Manager associated with that particular POA is in the active state.

By using one POA Manager for more than one POA, for example only one POA Manager for the complete server, it is possible to control more than one group of objects with a single call, thus working around possible race conditions if each POA had to switch state individually.

4.2.3 Request Processing

When a request is received by the ORB, it must locate an appropriate object adapter to find the responsible servant. POAs are identified by name within the namespace of their parent POA; like in a file system, the full "path name" is needed to locate an object's POA – obviously, it must be possible to derive this information from the object reference. One implementation option is that the request is delivered to the Root POA, which then scans the first part of the path name and then delegates the request to one of its child POAs. The request is handed down the line until the right POA is reached.

In this respect, the user can already influence the selection process whenever a necessary child POA is not found. The programmer can implement and register an *Adapter Activator* that will be invoked by a parent POA in order to create a non-existing child. This can happen if a server has been, to save resources, partially deactivated by stopping part of its implementation, or if the server has been restarted and now reinitializes object adapters and servants on demand.

Creating child POAs cannot happen without user intervention, because the programmer usually wishes to assign a custom set of policies: if a new POA has to be created but no adapter activator has been registered with its parent, requests fail.

Within the adapter activator, the user cannot only create further child POAs, but also activate objects within, possibly by reading state information from disk. This process is transparent to the client, which does not notice any of this server-side activity.

Once the object's POA has been found, further processing depends on the POA's servant retention policy and its request processing policy. The POA uses the Object Id – which must again be part of the object reference – to locate the servant.

Figure 4.6 shows the three possibilities that exist to find the responsible servant. In the most simple case, servants are activated explicitly and entered into the Active Object Map. If

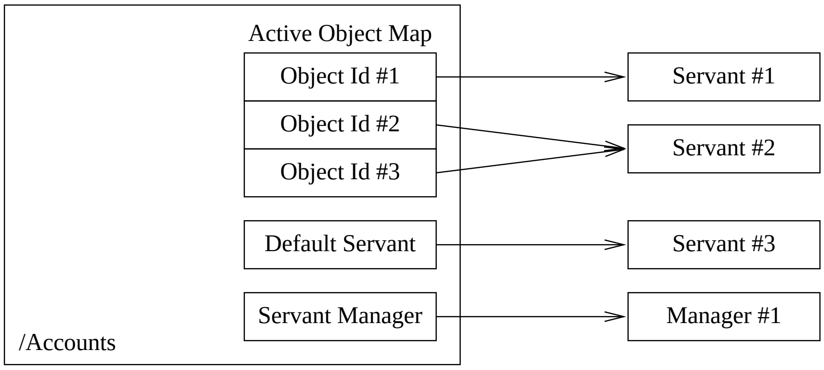

Figure 4.6: Each POA has its own Active Object Map

a POA has a servant retention policy of "retain," this table is consulted first. If an appropriate entry is found, the request is handed to the registered servant.

If the servant retention policy is "non-retain," or no matching entry is found in the Active Object Map, the request processing policy is considered. If its value is "use default servant," the user can provide a single *default servant* that will be used regardless of the request's Object Id. This allows an implementation to handle a group of – usually identical – objects with a single servant. This default servant can use the POA's context-sensitive introspection methods to query the Object Id of the current request and behave accordingly.

Default servants provide scalability using the flyweight pattern [10]: the server does not grow with the number of objects. Rather, the server can produce arbitrary numbers of object references while the number of active servants is constant.

A database server is an example for the usage of a default servant. Each table would be represented using a different POA with the table's same name, and the table's key value is used as Object Id. This way, all table rows are objects with their own object reference. Only a single default servant is needed per table; in an invocation, this default servant would query the request's Object Id and use it as table index. By using the DSI, the default servant could even be identical for all database tables, examining the table structure to select its parameter's types.

Even more flexibility, albeit of a different kind, is possible if the request processing policy is set to "use servant manager." If the POA's own search for a servant in the Active Object Map is unsuccessful, it then delegates the task of locating a servant to the user-provided *servant manager*. The servant manager can then use intelligence of its own to find or activate a servant.

Like adapter activators for POAs, a servant manager can be used to activate servants on demand after a partial shutdown or after server restart. The servant manager receives the request's Object Id and could use that information to read back the object's state from persistent storage.

Another interesting feature possible with servant managers are *virtual objects*, i.e. of object references that refer to non-exisent servants. References to virtual objects can be passed to a client; on the server side, a servant manager is then registered with the POA to incarnate the virtual object on demand.

An example for this mechanism is a file service. A "Directory" object might provide a method to return the directory's contents – a list of files – as a sequence of object references

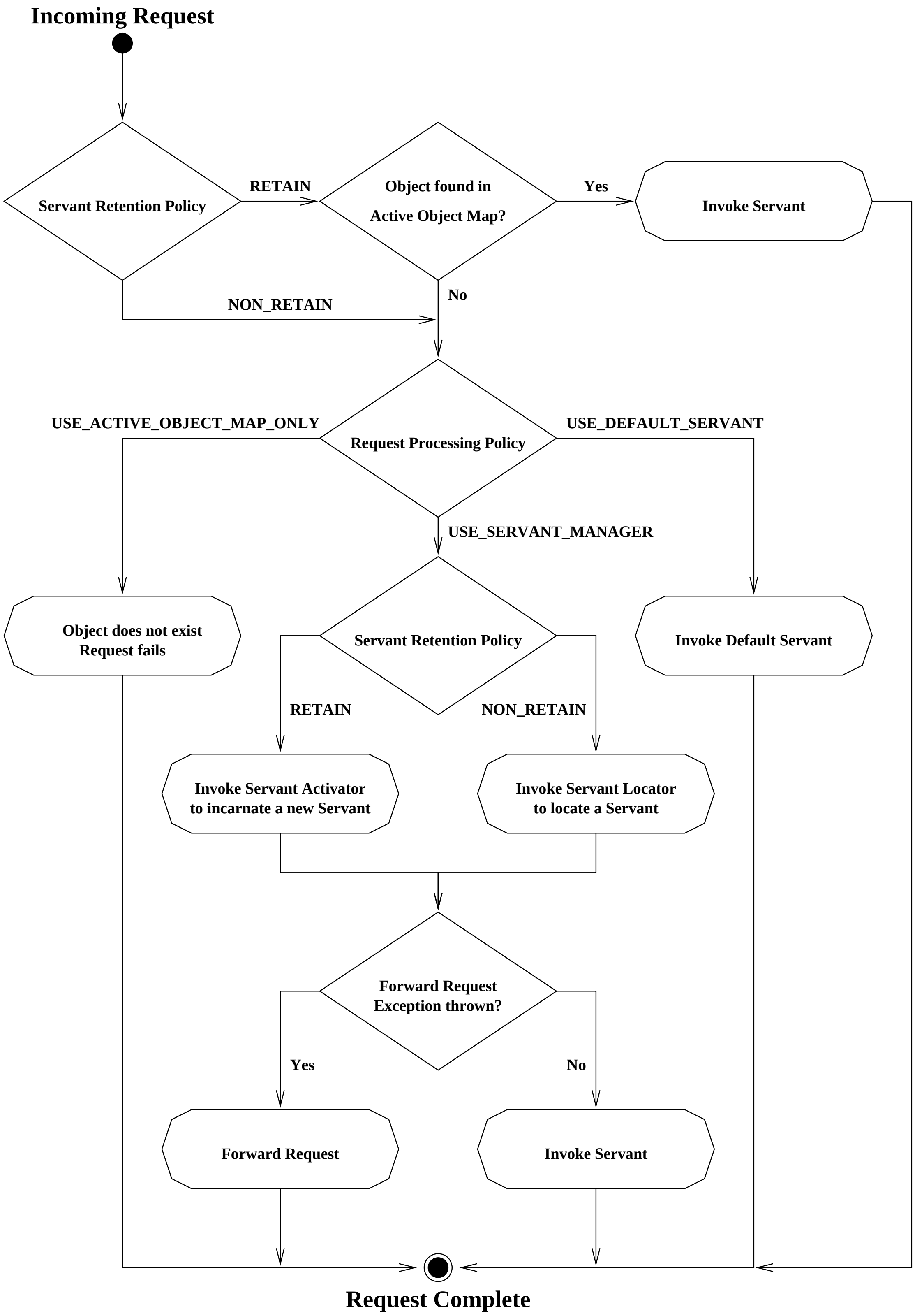

Figure 4.7: Request Processing within the POA

	Servant	Default Servant	Servant Activator	Servant Locator
Object State	yes	no	yes	no
Scalability	no	yes	no	yes
Virtual Objects	no	no	yes	yes
Lifecycle	no	no	yes	no

Table 4.1: Design Patterns and different types of Servants

to “file” objects. It would be inefficient to call all file objects into existence just for the purpose of returning their references, as only a small part of them would be actually read. Instead, the Directory would create virtual object references, encoding the file name in the Object Id. Once a file is opened, the file object would be activated by the servant manager.

Servant managers come in two different flavors with slightly different behavior and terminology, depending on the servant retention policy. If this policy’s value is “retain,” a servant *activator* is asked to *incarnate* a new servant which will, after the invocation, be entered into the Active Object Map itself – this would be sensible in the sketched file service example, since the newly incarnated file object would be needed more than once. If an object is deactivated, either explicitly or because of server shutdown, the servant activator’s *etherealize* method is called to get rid of the servant – at which point the object’s state could be written to persistent storage.

If the servant retention policy is “non-retain,” the servant manager would have to be a servant *locator*, tasked to locate a servant suitable only for a single invocation. A servant locator supplements the default servant mechanism in providing a pool of default servants; it is the flyweight factory according to the flyweight pattern. It can also be used for load balancing, as the example of a print service shows, in which the `print` method is directed to the printer with the shortest queue.

Both kinds of a servant manager can also throw a special “forward exception” instead of returning a servant. This exception contains a new object reference to forward the request to, possibly to an object realized in a different server on another system, employing the GIOP location forwarding mechanism. Forwarding allows, for example, the implementation of load balancing or redundant services: the servant manager would check its replicated servers and forward the request to an available one.

Another idea, already realized in the K Office Suite, is that of a “Meta Service,” in which a servant activator checks the request’s Object Id, launches the appropriate server, for example a Web browser or mail tool, and then forwards the request to the new server [72].

The flowchart in figure 4.7 shows in full how a POA tries to locate a servant according to its policies. To summarize a developer’s choices, table 4.1 shows a matrix of the different types of servants and their possible design patterns:

Object State: Manually activated servants and servants managed by a servant activator encapsulate their own state, whereas default servants must be stateless or store an object’s state externally.

Scalability: For stateless objects, default servants can serve many objects without memory overhead, and servant locators can dispatch requests to a small number of servants.

Virtual Objects: The idea of virtual objects can be realized by both kinds of servant managers.

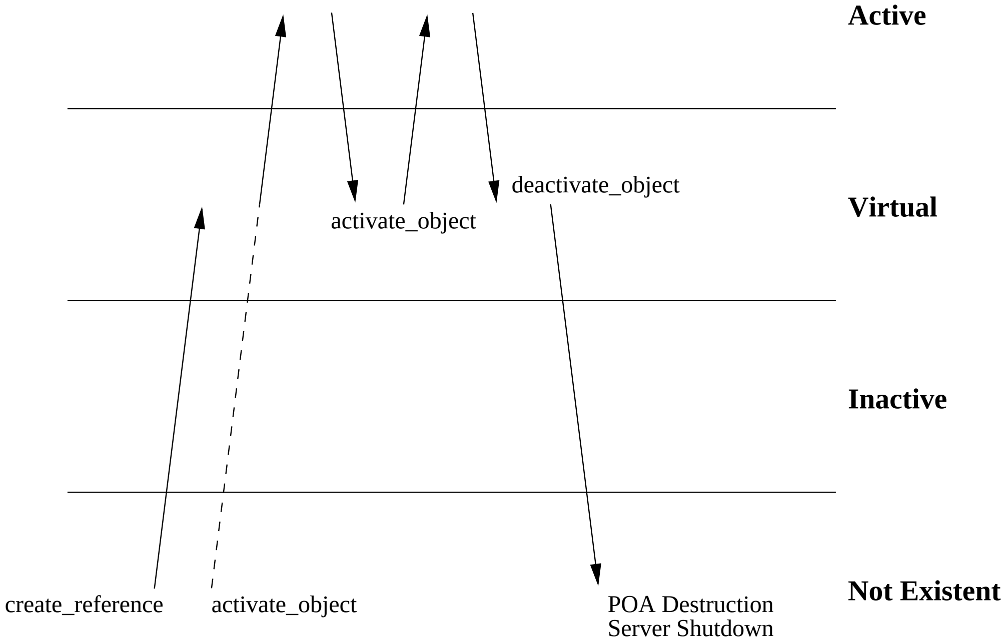

Figure 4.8: Lifecycle for transient objects using the POA

Lifecycle: An object's lifecycle can be monitored by a servant activator.

4.2.4 Persistence

Figure 4.3 showed the lifecycle of objects that used the Basic Object Adapter. The POA adds object virtuality as a fourth state. A virtual object is not currently associated with a servant; virtual objects are usually monitored by a servant activator so that they can be activated on demand. The transition from the virtual to the active state is called *incarnation*, and the transition from the active to the virtual state *etherealization*, corresponding to the methods called in the servant activator. An object is inactive if its implementation is not running.

The lifecycle of POA-based objects also depends on the POA's lifespan policy, which can take the values "transient" or "persistent." The lifetime of a transient object – an object activated in a POA with the transient lifespan policy – is bounded by the lifetime of its POA instance: once it is destroyed, all object references pointing to that POA's transient objects become invalid. It must not be possible to either intentionally or unintentionally reactivate a transient object when the same or another server is started again. The lifecycle of a transient object is shown in figure 4.8. Transient objects cannot become inactive, because if their server is shut down, they cease to exist and can never become active again.

On the other hand, persistent objects can, according to definition, outlive their server. It must be possible to start a server and activate objects that respond to object references exported from an old server. Their lifecycle is shown in figure 4.9 and differs from the lifecycle of a transient object in that a persistent object never fully ceases to exist. If their server is shut down, persistent objects become inactive, and can be activated again in a new server.

It is certainly possible that an inactive object is not associated with an implementation, or that a new server does not reactivate an object. In both cases, clients attempting an invocation will receive an exception and will probably consider the object non-existent, but other clients

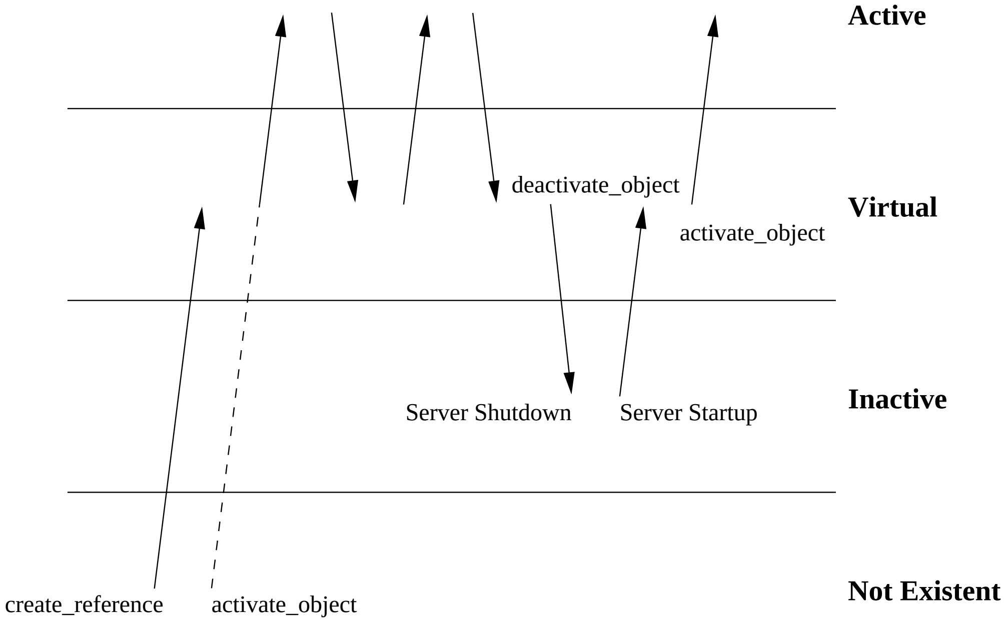

Figure 4.9: Lifecycle for persistent objects using the POA

do not notice, and if the object is reactivated at any point in time by a later servant, those clients can again perform method invocations.

Persistent objects have several advantages over transient objects, as it is possible

- to shut down a server to save resources,
- to update an implementation transparently by stopping the old server and starting the new version, and
- to recover from a server or system crash.

However, persistent objects need more managing. As described in the previous section, servant managers provide hooks to save and restore object state, which is a necessity with stateful persistent objects.

The BOA, where all objects satisfied the POA's definition of persistence, provided the Implementation Repository for the purpose of on-demand server restart. The POA specification does not address this issue and deems it to be the vendor's problem. From the description of the persistent lifespan policy:

> "Administrative action beyond the scope of this specification may be necessary to inform the ORB's location service of the creation and eventual termination of existence of this POA, and optionally to arrange for on-demand activation of a process implementing this POA."

An external POA component, like a daemon, is useful for the same reason as with the BOA, to receive requests while the actual server is not running, to start a new server if necessary or simply to wait until a new server is started manually, and then to forward the request. Without such a service, requests are failed with an exception indicating temporary failure while the server is down for replacement.

4.2.5 Language Mapping

Since its interface is specified in IDL, most of the POA's mapping into a programming language is automatic. Left open, though, is the servant type, which in IDL is defined as "native" and must therefore be mapped specifically by each language mapping.

Currently, the two most widely used languages used for CORBA programming are C++ and Java. This thesis centers on C++ for other reasons, yet a glance into the CORBA 2.2 Java language mapping [29] shows that the POA's servant type is not defined there, indeed, that Java's server-side mapping is very minimalistic and does not even provide any kind of object adapter.[3] By containing a comment about waiting for the availability of the POA specification, the Java language mapping appears rushed and out of date.[4] This will be fixed in CORBA 2.3 [32].

The C++ mapping does already contain the necessary mapping rules and definitions, including two different approaches how servants can be implemented, by inheritance or by delegation, according to the two forms of the Adapter pattern [10].

A servant is defined as an instance of a C++ class that, directly or indirectly, inherits from the system class `PortableServer::Servant`. Using inheritance, the IDL compiler generates a skeleton class with the same name as the IDL interface prefixed by "`POA_`." The user derives from the skeleton and implements the operations and attributes. With delegation, the IDL compiler generates a "tie" template; the user then provides a C++ class that implements all operations and attributes, but does not need to inherit any CORBA classes – a questionable feature, supposedly for easing the migration of existing code into a CORBA server, where the source code for existing libraries is not available and their inheritance therefore unchangeable. However, in that case tie classes aren't of much help, too, since they require the signature of existing code to match the C++ language mapping, or manual template specialization.

Examples for the implementation of servants using inheritance or delegation are shown in [59]. The text describes the same problems with delegation and also cannot present reasonable motivation for delegation either.

Still, some minor complaints remain. One is an omission with respect to implementing derived interfaces. If an IDL interface `Derived` inherits interface `Base`, the inheritance of the generated skeletons is undefined. Figure 4.10 shows two possible inheritance graphs: on the left, `Derived`'s skeleton inherits `Base`'s skeleton, while it can also be directly derived from the system base class, as shown on the right side. The important difference to the user is that the first possibility on the left allows for implementation inheritance by deriving the implementation of `Derived` from both the skeleton and the implementation of `Base`, while the other requires the user to re-implement `Base`'s operations in the implementation of `Derived`. Clearly, the first possibility is more sensible, but no word is spent on the subject, so a CORBA-compliant ORB implementation could choose the "wrong" inheritance on the right.

Another uncomfortable issue is that of reference counting. Obviously, a servant must live as long as it is active and therefore still referenced in a POA's Active Object Map. Frequently, however, the user does not want to keep track of a servant and delete it manually after its deactivation. If a servant manager is not used, it would be more comfortable to have a reference counting mechanism that deleted servants when the last reference is lost. After a servant's activation, the user could then release the reference immediately and forget about it.

[3] In Java, the ORB itself provides `connect` and `disconnect` methods for registering servants.

[4] Interestingly, while the Smalltalk and Ada server-side mappings share the Java mapping's status, the POA is readily available in the COBOL mapping.

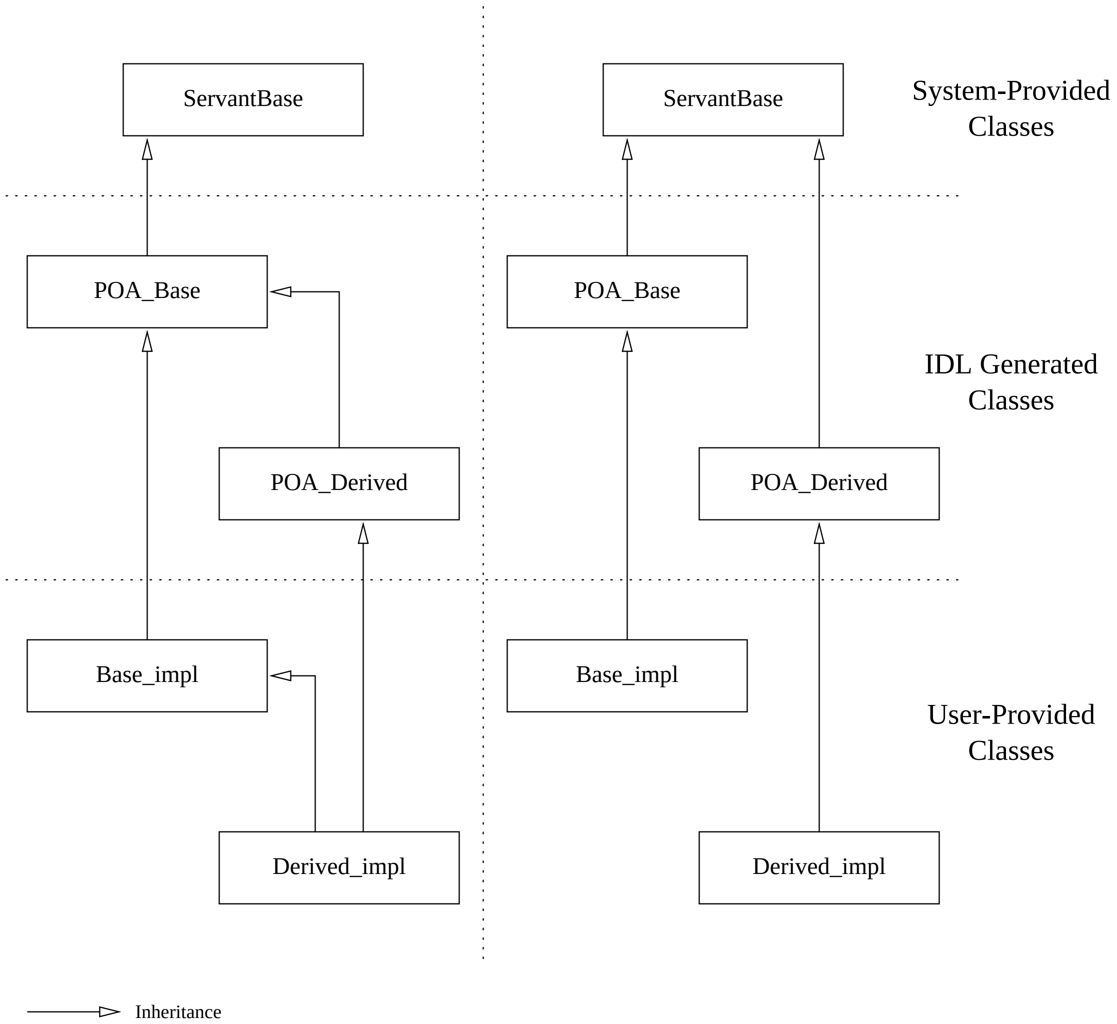

Figure 4.10: Two possibilities for the inheritance of skeleton classes

Feature	Basic Object Adapter	Portable Object Adapter
Complete Specification	no	yes
ORB Synchronization	no	yes
Scalability	no	yes
Flexibility	no	yes
Persistency	no	yes
Implementation Repository	yes	no

Table 4.2: Comparison between BOA and POA

In a multithreaded environment, reference counting is essential, because many invocations can be active at a given time, and a servant must not be deleted while any of them is in progress.

Reference counting for servants did not exist in CORBA 2.2, but will be added to the C++ language mapping in 2.3 [31]. The OMG decided on a backwards-compatible solution using a "mix-in" class. Without special effort, servants are not reference-counted, resulting in the same behavior as in 2.2. To employ reference counting, the user derives a servant class from `PortableServer::RefCountServantBase`.

While being a nice recovery, it is surprising why this mechanism was not already used in CORBA 2.2. To achieve backwards-compatibility, reference counting is now much more complicated as it could have been. And, since it depends on inheritance, it only works with the inheritance-based approach, not with delegation.

4.2.6 Evaluation

The most striking difference between the BOA and the POA is the information they manage. While the BOA dealt with servers and completely ignored the subject of servants, this approach is turned upside-down in the POA, which deals with servants but completely ignores the handling of servers and processes.

However, the POA does solve its primary task, of adapting servants to the ORB and managing them, very effectively. The complaints with the BOA described in section 4.1.1 are solved: the POA provides a clear definition of the term "servant," their activation and deactivation, and the layout of skeleton classes. In this respect, the specification is *complete*; no incompatibilities need to be introduced by vendor's interpretations. Table 4.2 compares the features of BOA and POA.

Default servants provide scalability, and adapter activators and servant managers allow for much flexibility. Using these hooks, user code can be involved at many steps during request processing. For example, servant managers can be used to realize persistency by saving and restoring an object's state. POA Managers provide the required means of synchronization with the ORB.

All in all, the POA indeed achieves its goal. Not so much as being "portable," though. The unportability of BOA-based server code was only a secondary result of its inadequate specification. Rather, quoting the Portability Enhancement RFP, the POA succeeds as a universal object adapter.

And yet an incompleteness remains with the persistent lifespan policy. Resulting from the design decision not to cover the uncomfortable area of server handling, the automated restart of servers is not addressed as it was with the BOA's Implementation Repository. While transient objects can be implemented in a portable manner, incompatibilities may arise again with persistent objects. This would be particularly annoying, since it is to be expected that many "real life" servers will require persistence.

The next chapter will show that persistence not necessarily introduces *source-level* incompatibilities, but TAO, for example, intends to extend the POA interface with methods for registering objects with an implementation repository.[5]

Frequently brought forward as a pro-POA argument is that it supports multithreading. It is true that the POA specification addresses the issue, but the threading policy is coarse and insufficient for many multithreading requirements. It does not allow user control over threading issues but simply selects whether servants are or are not thread safe, either serializing requests or dispatching them as controlled by the ORB on a per-POA basis. Ideally, a "thread manager" would be registered with a POA to assign requests to threads, either by selecting from a thread pool or by starting new threads.

A small inconvenience from a technical rather than conceptual nature is the implementation of adapter activators and servant managers. According to the specification, they follow the same semantics as normal servants. As such, they must be activated, and method invocations are subject to their POA's readiness. Apart from allowing embarrassing situations like the need for invoking an adapter activator and a servant manager in order to incarnate another adapter activator, this can cause problems if only one POA Manager is used: if the POA Manager is set to the inactive state prior to server shutdown, objects could not be etherealized, since the servant manager, too, cannot receive requests any more. While this effect is surprising,[6] it is not a bug and can be avoided by the careful programmer.

Handling adapter activators and servant managers as normal objects seems natural, as their interface is specified in IDL. But both take "native" parameters, so automated stub and skeleton code generation does not work anyway. It would have been less troublesome, if less consequent, to declare adapter activators and servant managers as "pseudo objects," like the POA itself.

A few open issues against the POA remain on the OMG Web site [33, 34], but apart from those mentioned here, they are of a minor technical nature, for example whether the special case of nested method invocations on the same object would violate the single-thread threading policy.

Of course, the POA is bound by the current limitations of CORBA itself (see section 3.5). It supports request-response communication, but cannot adapt data streams, and does not provide real-time features.

The mentioned quirks in the C++ mapping and the remaining open issues will hopefully be resolved in a future version of the CORBA standard.

4.3 Other Object Adapters

One object adapter not specified, but hinted at by the CORBA 2.0 specification [27], is the Library Object Adapter, which was supposed to adapt servants that are not running within their own server, whose implementation is not an executable program but contained in a

[5] The Implementation Repository documentation of version 0.3.23 mentions the additional method create_-reference_with_virtual_server. It has not been implemented yet.

[6] And usually hard to notice, as failures to invoke a servant manager are to be ignored. Developers would get suspicious that object state is not saved while everything else continues to work smoothly.

library that can be dynamically linked into client programs:

> "This object adapter is primarily used for objects that have library implementations. It accesses persistent storage in files, and does not support activation or authentication, since the objects are assumed to be in the clients program."

Obviously, the motivation was to provide servants that would always be local to the client. Despite this vague description, library-based servants were actually possible with MICO, but based on the BOA and a MICO-specific "library" activation policy.

It is not possible to realize the Library Object Adapter's intention with the POA design. Rather, Objects by Value will allow objects with a guaranteeable local implementation in CORBA 2.3.

Obsoleted by the POA is the idea of an Object-Oriented Database Adapter, also described by CORBA 2.0:

> "This adapter uses a connection to an object-oriented database to provide access to the objects stored in it. Since the OODB provides the methods and persistent storage, objects may be registered implicitly and no state is required in the object adapter."

A concrete implementation for an Object-Oriented Database Adapter is available in Orbix, which provides an *Object Database Adapter Framework*. However, the same behavior can now be achieved with the POA, using default servants and/or virtual objects, as described in section 4.2.3.

Both paragraphs have been removed and do not exist in the CORBA 2.2 specification.

Chapter 5

Reference Implementation

The previous chapter concluded that the POA design does look sensible. But as already noted, it had not been implemented at the time of its submission, and so the uncomfortable doubt remained that it might be hard or impossible to implement. A similar situation had occured with other parts of CORBA. The Persistent Object State Service, for example was never fully implemented and later found to be altogether buggy [38]. The goal was therefore to attempt a reference implementation of the POA in order to verify the specification's realizability.

Efficiency was to be secondary to a complete implementation, but not to be disregarded.

5.1 MICO

The MICO ORB [21] was used as platform for the reference implementation. MICO is an Open Source [46] CORBA implementation from the University of Frankfurt, originally designed and implemented by Arno Puder and Kay Römer. Version 2.2.7 was recently certified for CORBA 2.1 compliance by the Open Group [45] and therefore provided a solid and stable basis.

One design goal of MICO was to be easily extensible, as detailed in Kay Römer's thesis [55]. With its microkernel approach, the MICO core provides abstractions for object adapters, transports and schedulers. By derivation from abstract base classes, new components can be plugged in at runtime.

The microkernel ORB itself is reduced to the "broking" of requests. After a request is passed to the ORB from an invocation adapter (like the DII or SII), it simply tries all object adapters and asks whether they feel responsible for the request (see figure 5.1).

An object adapter is not restricted to adapting local objects: sending a request to a remote server satisfies the object adapter concept – the location of an appropriate servant – just as well. Consequently, MICO uses *IIOP Proxy* object adapters to address remote objects on the client side, and an *IIOP Server* invocation adapter on the server side (figure 5.2). With this innovative design, the ORB is not aware of an object's remoteness – a remote invocation is merely a side effect of passing the request to the right object adapter.

While IIOP Proxy and IIOP Server implement IIOP, they do not transfer data by themselves, but use *GIOP Connection* objects to perform GIOP encoding, and finally use instances of *Transport* and *TransportServer* for unidirectional[1] data transfer.

[1]CORBA 2.3 introduces bidirectional GIOP, so Transport and TransportServer will ultimately have to provide bidirectional data transfer, too.

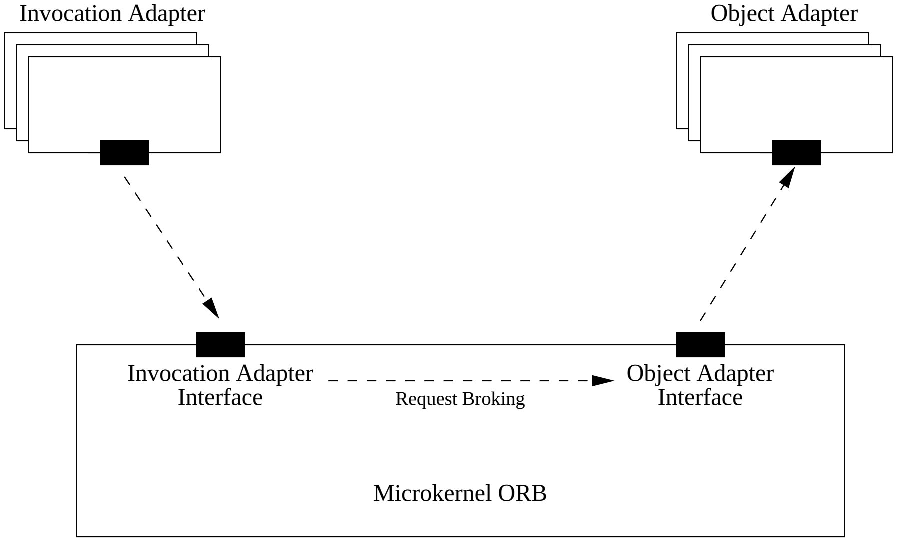

Figure 5.1: The MICO microkernel ORB

With this modular approach, the ORB is indeed easily extensible, as each component can be replaced. A new transport, for example one that compresses GIOP data before sending, can be added just by subclassing the Transport class and providing methods for reading and writing data.

The interface for object adapters is of particular interest here, as it will be used to add the Portable Object Adapter. To the MICO core, an object adapter must provide a small interface of three methods:

- Location: on the basis of an object reference, an object adapter must decide whether it can or cannot serve that object, i.e. if its servant is registered with that object adapter. Since MICO allows many object adapters to coexist, the ORB needs this location mechanism to choose the right object adapter in order to dispatch an invocation.

- Invocation: once an object has been located, the ORB asynchronously passes the request to the object adapter. Once request processing has finished, the object adapter notifies the ORB and returns the result.

- Cancellation: to cancel a request during execution.

MICO already fully implemented the Basic Object Adapter with all its activation policies, while taking its own liberties with the specification. A special *BOA Daemon* was used to contain the Implementation Repository and to activate implementations on demand. To allow object persistence, object implementations could overload the method "`_save_object()`" and later install special restoration objects.

Like other CORBA implementations, MICO decided to make the BOA as invisible as possible; the BOA was not so much programmed as simply "there somewhere." As only user-callable methods, `impl_is_ready` and friends remained, but more or less stripped of their original meaning, as they were usually called without any parameters. Server programming with the MICO BOA is described extensively in its manual [50]. Optionally, MICO allows to

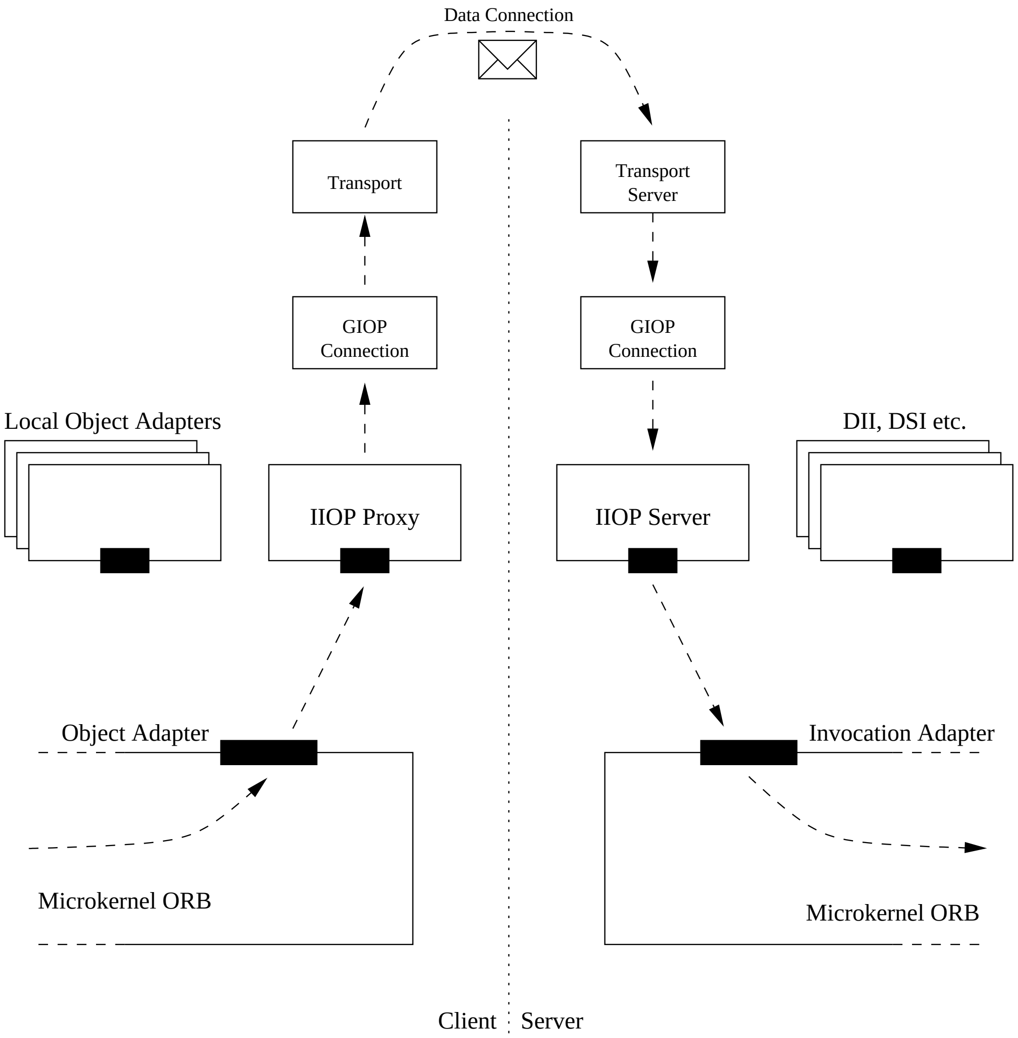

Figure 5.2: Remote invocation with a microkernel ORB

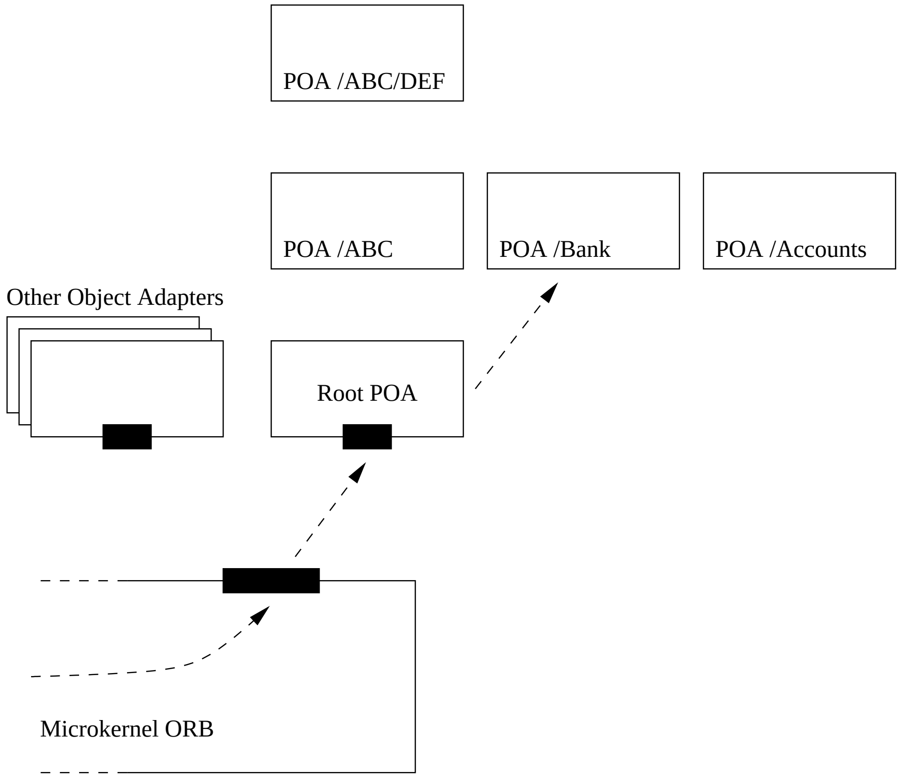

Figure 5.3: Only the Root POA is registered with the MICO ORB

bypass the BOA Daemon and the Implementation Repository, and to connect clients directly to their servers.

Also realized as object adapters are the OAMediator and POAMediator objects from the BOA Daemon, which receive requests for persistent objects and forward them to a running server. More on this appears in section 5.4.

Implementing the Portable Object Adapter consisted of four different parts:

- Providing the framework, such as type definitions and base classes for static and dynamic skeletons.
- Implementing the POA itself, and registering it through the generic object adapter interface.
- Modifying the IDL code generator to reflect the changes in server-side skeleton classes.
- Support for persistent objects and the on-demand starting of servers.

5.2 Design

Of the many POA instances that can exist in a server, only the Root POA is registered with the ORB (see figure 5.3). It is made responsible for receiving requests for any POA, inspecting their target object reference and dispatching them to child POAs as necessary. Since MICO allows for many object adapters, another design would have been to register each POA instance directly with the ORB. However, that would pose a problem with non-existent adapters, as a non-existing POA instance does not mean that the request is undeliverable. Rather, if a POA does not exist, the request must be delivered to the parent POA, which can then invoke its adapter activator to recreate the missing POA. Only the Root POA is guaranteed to exist throughout the server's lifetime – it can be destroyed, but not be reactivated. By handing requests down from the Root POA along the line of descendants, adapter activators can be invoked at each step.

As a slight improvement in configurations with deeply nested child POA's, the Root POA keeps a map of all existing POAs. For an incoming request, the Root POA checks the request's object key, and dispatches the request directly if the responsible POA does exist. Only if the lookup in the global POA map fails does the Root POA need to dissect the POA name into its components in order to locate the most specific existing POA among the missing POA's ancestors.

Once the target POA is found, it proceeds as shown in figure 4.7 to find a servant. Some care was taken in designing the Active Object Map, so that many active objects can be handled efficiently. The Active Object Map is a C++ class of its own, actually consisting of two STL maps, one mapping Object Ids to *object records*, the other is a multimap mapping servants to object records.[2] With STL's runtime complexity guarantees, entries can be found with a runtime of $O(\log n)$ either by Object Id or by servant.

Beside the servant, an object record holds a full object reference, so that, when requested, this reference can be merely duplicated instead of recomposing it from scratch each time.

Figure 5.4 shows an UML class diagram [42] of the classes used in the reference implementation. The four classes at the top of the diagram are provided by the ORB. `POA` is the IDL-generated base class, `ObjectAdapter` the abstract base class for Mico object adapters, `ORB` the ORB core class, and `ORBRequest` encapsulates an incoming request. At the bottom, `Servant` is the base class of all skeletons.

POA_impl Implements the `POA` and the `ObjectAdapter` interfaces, though only the Root POA is actually registered with the `ORB`. Each POA keeps an `ActiveObjectMap` and a number of `InvocationRecords`, in case requests must be queued while the POA Manager is in the holding state. If the POA has the "system id" id assignment policy, it keeps a `UniqueIdGenerator`.

ActiveObjectMap Maintains a list of `ObjectRecords`, indexed by `ObjectId`.

ObjectRecord Associates a servant with an object reference; object references are encapsulated in a `POAObjectReference` value.

InvocationRecord Associates an incoming request with an object reference.

POAObjectReference Provides methods for the efficient composition and decomposition of an object reference (or rather the object key) into its components.

ObjectId An efficient encapsulation of an Object Id, mainly to serve as the key in the Active Object Map.

[2] Object Ids can only be activated once and therefore map to at most one object record. Servants can be activated more than once (depending on the id uniqueness policy) and therefore map to more than one object record.

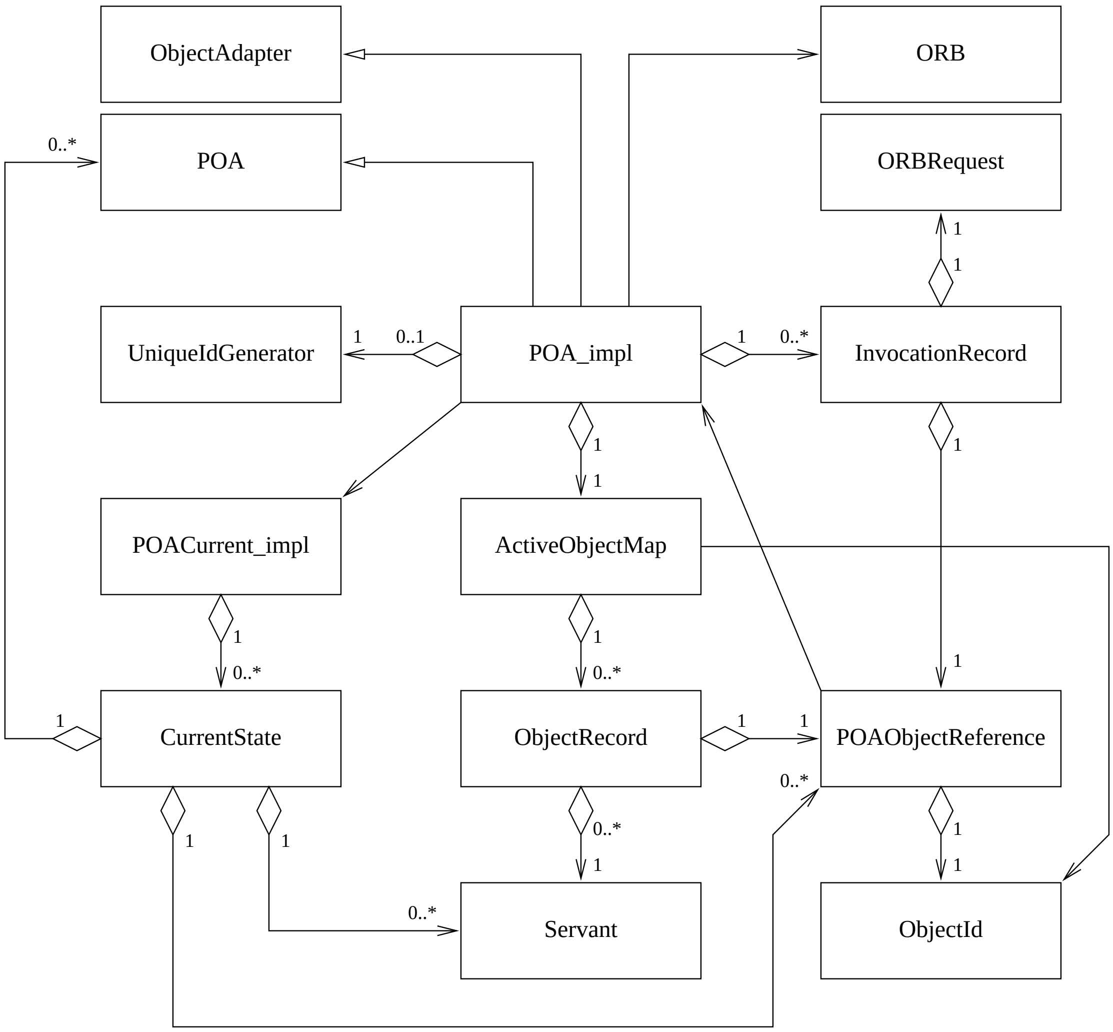

Figure 5.4: Class Diagram for the Portable Object Adapter Implementation

POACurrent_impl Implements the `PortableServer::Current` interface, whose purpose is to provide context information during a method invocation. Because more than one invocation may be in progress (nested method invocations), it keeps a stack of `CurrentState` objects. The inheritance from the IDL-generated base class is not shown.

CurrentState The currently executing object reference, POA and (indirectly) servant must be available as context information.

For the most part, the implementation of the Portable Object Adapter did not hold any surprises. After deciding on the design and the class layout, putting the precise wording of the specification into code was for the most part straightforward.

More thinking had to go into the unspecified part of constructing the object key, and into realizing the missing control over persistent objects and their implementations. Both subjects are detailed in the following sections, as is the collocation optimization mechanism to reduce the overhead of method invocations if client and server are in the same process.

The IDL code generation for skeleton classes was taken almost unchanged from the BOA, with minor adjustments for the inheritance and naming convention: the format of requests and their dispatching is the same as before. Without much innovation, the POA skeletons use the same optimizations that already existed for BOA skeletons to use the Static Skeleton Interface.

5.3 Object Key Generation

Two central tasks of an object adapter are the creation of object references, and later the demultiplexing of a request and delivering it to the right servant. In order to perform the latter, the object adapter must take care in composing object references so that they contain all information necessary to find the one and only responsible servant.

First of all, the object reference must contain one or more profiles with information on how to contact the server; for the IIOP profile, this would be an Internet host name and TCP/IP port number. However, since an object adapter does not know about profiles or transports, this information must be provided by the ORB. In Mico, the ORB keeps an `ior_template` containing all available profiles but no object key.

The POA then constructs the object key in a way that allows

- identification of the servant,
- identification of the POA instance, and
- identification of the object adapter.

The first two components have already been mentioned in the abstract description of the POA. The servant within a POA is identified using its Object Id, and the POA instance is identified using the "full path name" of the POA, that is, the name of the POA itself and all of its ancestors, up to the Root POA.

To identify the object adapter, a globally unique identifier (GUID) is used. In a transient POA, this GUID is generated of the server's host name, the process' pid and a numerical timestamp. Host name and pid are included in this GUID because the object adapters' location mechanism is also used to disambiguate between local and remote objects; with only

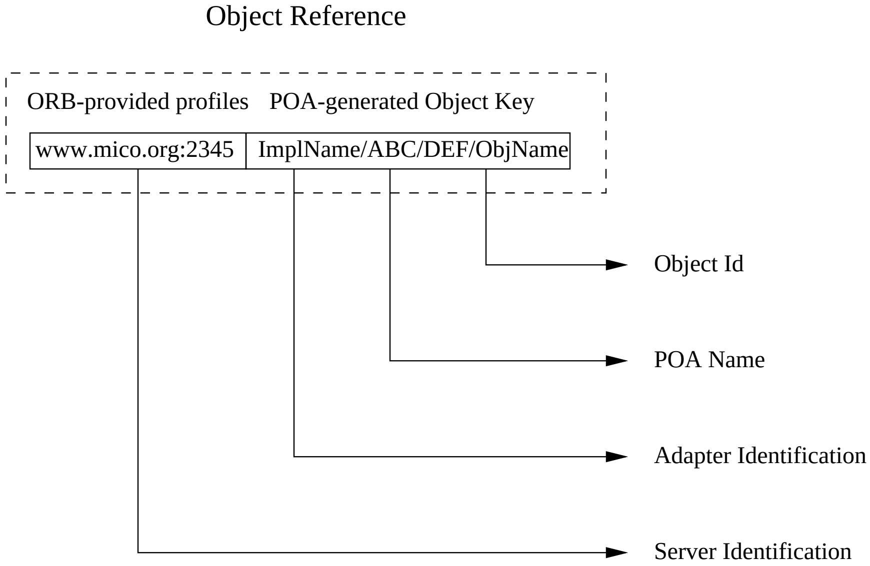

Figure 5.5: Components of an object reference with a POA-generated Object Key

the timestamp, a local POA might also be held responsible for remote objects, if their POA were created at the same moment in time. The composition of the GUID is unique among the current object adapters in Mico, the BOA for example always prefixes its object keys with the three letters "BOA."

In a persistent POA, a user-provided identifier, the "implementation name," is used as GUID instead. The following section deals with persistent objects in detail.

A potential object reference with an IIOP profile and a POA-provided object key is shown in figure 5.5.

The object key is defined as sequence of octets with no internal structure. Special care must therefore be taken that the object key can be decomposed unambiguously: the three parts are concatenated, separated by slashes "/." Since a user is free to use slashes in the name of a POA instance or in a user-selected Object Id, these must be "escaped" by the POA in constructing the object key.

The location testing required as part of the Mico object adapter interface can be implemented efficiently. For this test, the object key does not even need decomposition, the Root POA just compares the object key's prefix with the two known GUIDs, one representing all transient POA instances, the other all persistent POA instances within the server.

The mechanism of using slashes as separators was a late addition. Previously, a null octet was used. Since neither GUID nor the POA name can contain null characters, this greatly simplified decomposition of an object key at the cost of their legibility.

This became a concern after the OMG unveiled the preliminary specification of the Interoperable Naming Service (INS) in October 1998, which finally addresses a client's bootstrapping problem of receiving initial references in a portable manner, introducing new, simplified, formats for Interoperable Object References as already mentioned in section 3.2. Its idea of having URI-style object references mandates short, legible object keys that can be easily written down, entered by hand, or exchanged over the phone. Using the new notation, the object reference for the object key shown in figure 5.5 could look like

`iioploc://host:1234/ImplName/ABC/DEF/ObjName` provided that the server implementing the object was running on "host", port 1234.

To achieve even shorter object keys, the POA uses a special case of encoding the object key if the (user-selected, in the case of a persistent object) GUID, the POA Name and the Object Id are identical, and "collapses" the three identifiers into one. If, therefore, a persistent object with the Object Id "NameService" is activated in the POA "NameService," a direct child of the Root POA in the server identified by the GUID "NameService," the complete object key would be `NameService`, too.

References to transient objects don't look nearly as elegant, as the GUID for transient objects is already large on its own, but then, the new style of object references is most likely to be used for persistent services, such as the Naming Service[3] itself.

The contents of an object key in general are unspecified, subject to the ORB vendor. Having a specification impose requirements about the object key – according to the INS, the Naming Service *must* have an object key of `NameService` – leads to the interesting situation that it will be impossible to write a *portable* implementation of the Naming Service, as each ORB will introduce different mechanisms to achieve this object key. While this implementation happens to rely on consent with the programmer alone, who must choose the POA name and the Object Id accordingly, Vishal Kachroo, a member of the TAO team, once mentioned on the Mico mailing list that his implementation of the INS would involve a proprietary additional POA method that explicitly sets an object key, bypassing the usual object key decomposing and dispatching mechanism.[4]

Consequently, TAO's Interoperable Naming Service will not compile with Mico, and while Mico's implementation will compile with TAO, it will not yield desired results and won't answer requests at the expected location.

5.4 Persistence

Transient objects require that object references to objects in a particular server become invalid if that server is shut down. In other words, the object keys for transient objects must be unique to a server instance.

This is guaranteed by the composition of the GUID that is used as a prefix for the object key, which, as seen above, contains the host name, the server process' PID and a timestamp. Together, these three components uniquely identify a server. If two instances of the same server are running on the same host simultaneously, the PID is different, and if a server is shut down and run again, at least the timestamp (with millisecond resolution) differs. The GUID therefore serves a dual use as adapter identifier and transient property.

For persistent servers, a different GUID must be used that *is* reusable across server instances, but as the POA specification ignores the administrative issues of persistent objects, the *how* of assigning such GUIDs is undefined.

As a solution for the Mico POA, the user must use the `-POAImplName` option on the command line to set an *Implementation Name* when starting a server that implements persistent objects – otherwise, the Invalid Policy exception is thrown when trying to create a POA with

[3]The preliminary INS specification itself confuses the usage of the identifier "NameService" vs. the name "Naming Service" several times. The current distinction when to use which is the result of a newsgroup discussion.

[4]In TAO 0.3.23 (May 25, 1999), the TAO ORB core maintains a table mapping "short" user-selected object keys to default, POA-selected object keys. An implementation explicitly adds entries into that table using the `_tao_add_to_IOR_table` method.

the persistent lifespan policy. This identifier is then used as adapter identifier in the object key. It is the user's responsibility for keeping the Implementation Name unique.

Transient objects allow for direct communication between client and server: if the server's transport end point is disconnected as a result of server shutdown, all the client's object references into that server become invalid, and no further communication is possible.

Persistent objects, however, require a mediator [10] that intercepts requests while the server is off-line, starts up the server on demand, and forwards the request when the server is again ready to receive them [15]. As mentioned in section 4.2.4, this issue is ignored by the POA specification and must be solved by the vendor.

5.4.1 POA Mediator

For this implementation, it was decided to re-use the already available component of the "BOA Daemon," which already did much of the same work for the Mico BOA. The BOA Daemon manages the Implementation Repository, which contains information about available servers and the objects they implement.

The BOA Daemon contains the "OAMediator" singleton object, and cooperates with the BOA to

- produce object references,
- keep track of active persistent objects,
- synchronize delivery of requests and to
- support object migration into a different server.

Obviously, the most basic requirement is that clients must not communicate with a server directly but always contact the mediator, which is expected to run permanently, beyond the lifetime of the persistent object.

This is a problem in the server, where object references are generated as part of object activation. In the OAMediator design, the BOA does not generate object references on its own, but invokes the OAMediator, which uses the object key provided by the BOA and its own profiles to produce a new object's reference pointing to the OAMediator.

After object activation, OAMediator and BOA follow an elaborate protocol which is described in section 5.4 of [55], where it is referred to using its historical and still accurate name "BOA Mediator."

However, the existing OAMediator did not prove sufficient for persistent objects realized through a POA. It did not provide persistent storage, and it required the registration of *all* persistent objects. This would have been a real constraint when taking into account the concept of virtual objects, and objects realized through a default servant. While such objects do not require resources on the server side, they would then require an entry in the OAMediator's tables without ever being able to clean them up.

Rather than extending the OAMediator, it proved easier to introduce a new "POA Mediator." Though it serves much the same purpose, it was implemented with less code than the original by using a new design and the GIOP location forwarding mechanism, which hadn't been available to the OAMediator.

GIOP location forwarding allows to update a client's object reference with new profiles and conveniently matches our mediator's requirements. After starting a new server, the mediator

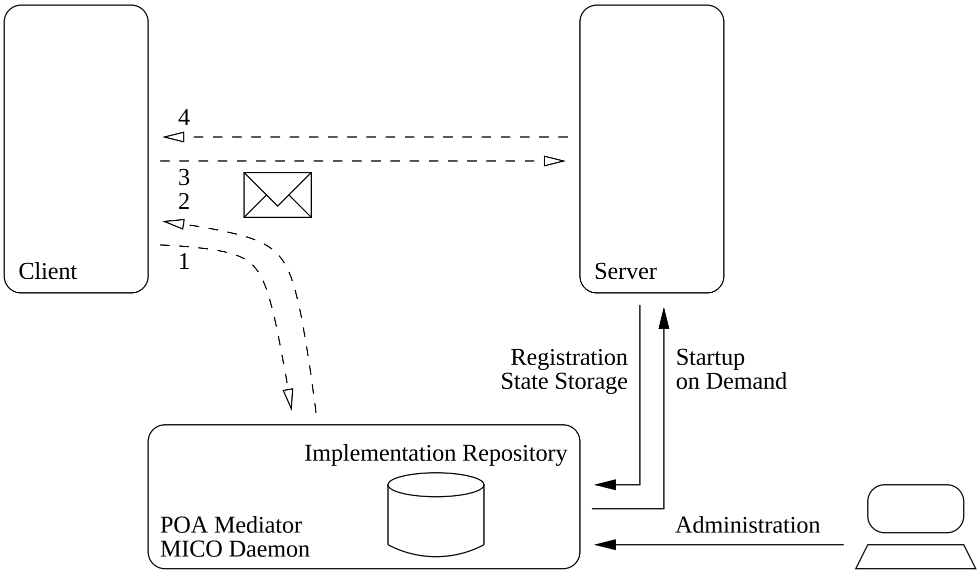

Figure 5.6: All client requests are routed through the POA Mediator

returns a forward message to the client, which – transparently to the user code – resends the request to the new location. Further requests are sent directly to the server and do not need handling by the mediator, until the server is shut down. This is noticed by the client, which receives a GIOP Close Connection message and then falls back to contacting the mediator again. The communication overhead induced by the mediator is kept minimal.

The POA Mediator is also a singleton object, integrated in the "BOA Daemon." True to the idea of a microkernel ORB, it is plugged into the ORB as an object adapter: instead of delivering requests to a local implementation, they are forwarded to the currently running server. Figure 5.6 shows the architecture of the POA Mediator. The numbered arrows indicate a request's path to the server:

1. The request is sent to the POA Mediator.
2. The POA Mediator starts the server and sends a GIOP location forward message back to the client.
3. The client forwards the request to the server, and uses the new address for all further requests until the connection is broken.
4. The server processes the request and sends the reply directly to the client.

Information about available servers is kept in the Implementation Repository, which has been extended with the new, according to BOA terms, *poa* activation policy. The Implementation Repository is now shared, and the two mediators must agree on the "ownership" of entries: the POA Mediator only handles servers with the *poa* activation policy, and a new rule has been added to the OAMediator to ignore such entries.

The POA Mediator must arrange to intercept all client-server communication. All object references to persistent objects must point to the mediator, so that the server can exit without invalidating any addresses. Rather than asking the mediator each time, a server and the POA

```
interface POAMediator {
  string create_impl (in string svid, in string ior);
  void activate_impl (in string svid);
  void deactivate_impl (in string svid);

  void save_state (in string svid, in StateInf state);
  StateInf restore_state (in string svid);

  // admin interface
  boolean force_activation (in ImplementationDef impl);
};
```

Figure 5.7: IDL for the POA Mediator

Mediator *exchange* their object reference templates (see section 5.3) upon startup, when the server calls the POA Mediator's `create_impl` method. When creating an object reference in a POA with the persistent lifespan policy, the POA does not use the local ORB's template, but the POA Mediator's: requests will then be sent to the mediator first.

The mediator then uses the server's template to construct a forwarding address for incoming requests, by constructing a new object reference from the server's template and the request's object key.

But before that can be done, the correct implementation must be found. The POA Mediator may be managing many implementations simultaneously and needs a means of directing the request to the right one. The trick is to require that a server's Implementation Name must match the name of the corresponding entry in the Implementation Repository. When a server is started on demand by the POA Mediator, the necessary `-POAImplName` command line switch is added automatically. Because of the way an object key is constructed, this results in the object key being prefixed by the Implementation Name. All the POA Mediator has to do is to read the object key from the request, and to use its prefix as an index into the Implementation Repository.

The only information kept by the POA Mediator is a single table, keeping the object reference templates of currently active servers. By puzzling with the pieces of an object reference, costly communication with the server is avoided. Where the OAMediator had to be informed of object activation by the BOA, keeping a table associating them with both an Implementation Repository entry and an active server, the POA Mediator simply does not need to know.

5.4.2 Synchronization

Still, some minor handshaking between a server and the POA Mediator is necessary. The four possible states an implementation can enter are shown in figure 5.8.

If no server is running, the implementation is *inactive*. If a new request is delivered to the mediator, it consults the Implementation Repository to find the name of an executable file for the implementation (as deduced from the object key), and uses POSIX process mechanisms to start the new server and POSIX signals to monitor them. The act of starting the server transitions the implementation to the *started* state, in which the mediator does not deliver

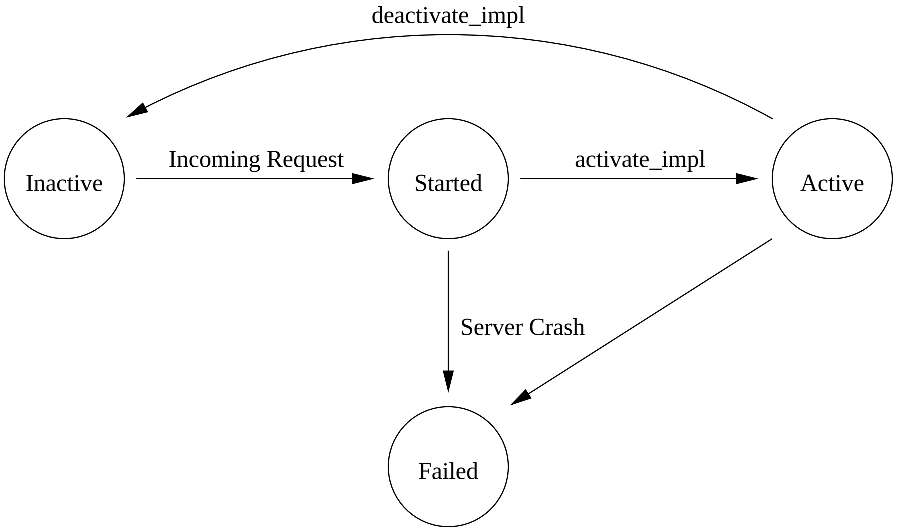

Figure 5.8: Implementation States according to the POA Mediator

any requests but keeps them in a queue and waits for the server to announce its readiness.

Within that state, the server can perform initialization unhindered by incoming requests. The Root POA calls the mediator to exchange object reference templates and to restore its state information (see next section), and then allows the user to activate or restore objects.

If the user is finished setting up the server, the Root POA's POA Manager is transitioned to the active state. At that time, the POA Mediator is informed that the server is ready, by invoking its `activate_impl` method.

Having waited for that callback, the POA Mediator sets the implementation's state to *active* and begins request delivery, first by draining its queue and then by forwarding requests as soon as they come in.

This continues until the server shuts itself down by calling the ORB's `shutdown` method, either as the result of a method invocation with the intended side-effect of server shutdown, or of a timeout within the server, which could decide to stop itself after a period of inactivity. Ultimately, this causes the Root POA to be destroyed, which calls the mediator's `deactivate_impl` method and returns the implementation to the inactive state.

A race condition exists upon shutdown. Since the POA Mediator and the server run in parallel, requests may be delivered during the short time between the initiation of server shutdown and its unregistration with the POA Mediator, when the ORB still accepts requests after the implementation has been deactivated. Also, the server could have stopped processing requests some time before shutdown by setting its POA Managers to the holding or inactive state.

Requests for transient objects are simply failed: the object they're referring to is ceasing to exist. However, this is not true for persistent objects, and the server faces a dilemma:

- The POA Mediator has forwarded and already forgotten about the request, the server keeps the only copy.

- The state of the POA Manager must be obeyed, so the request cannot be delivered to a servant.

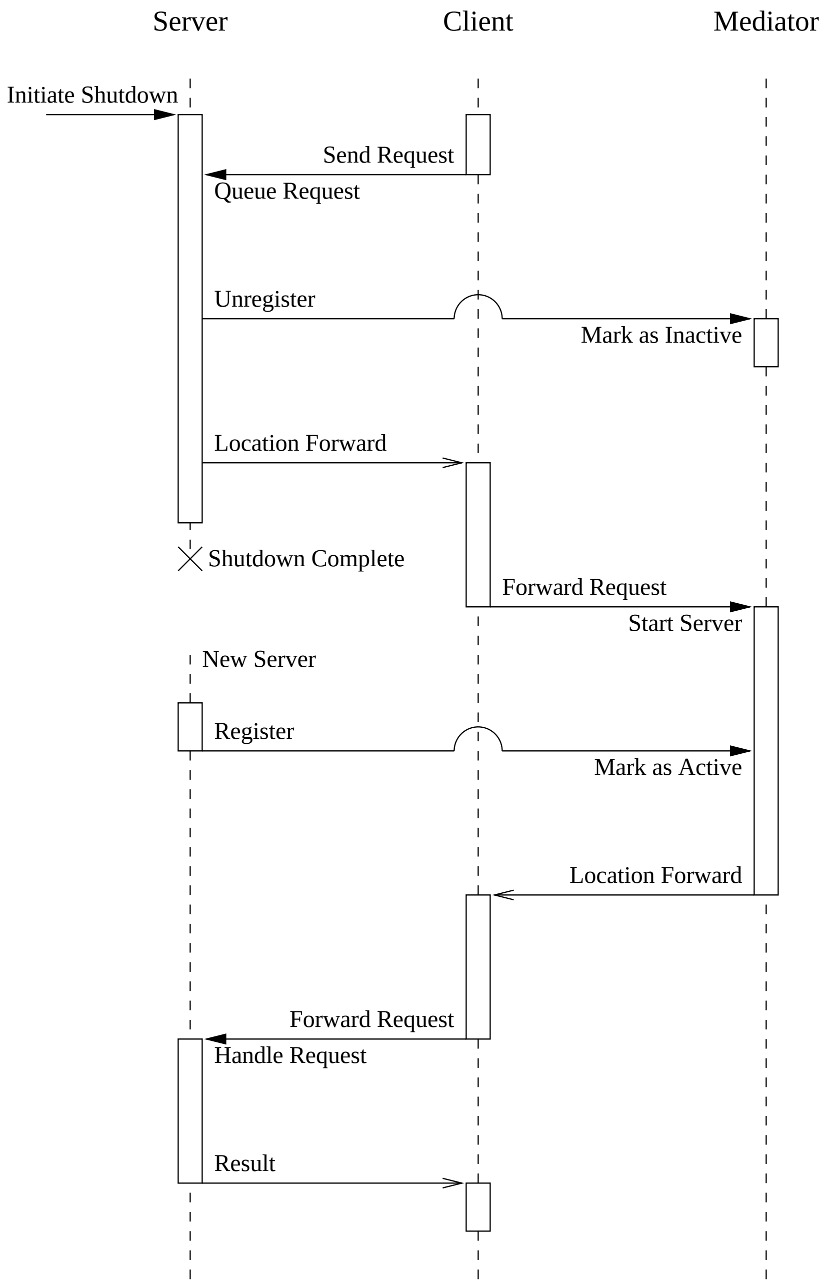

Figure 5.9: Handling of requests during server shutdown

- The server is about to exit, invalidating the request's addresses.

As a solution, the server queues requests during shutdown, even while the POA Manager is in the inactive state. Only *after* the Root POA has called the mediator to halt request delivery, these requests are bounced back to the client using the GIOP forwarding mechanism (see figure 5.9), which forwards the request to the mediator.

Once the request is again received by the mediator, it determines the implementation to be in the inactive state and starts a new server. The old server may not have exited yet, so the old and the new server may run in parallel, but all requests are now directed to the new server.

5.4.3 State Information

Object state and their handling using servant managers has already been mentioned several times. However, not only persistent objects have state, but also persistent POAs themselves. If a POA has the "system id" id assignment policy, it must generate unique Object Ids upon each activation. This duty must be fulfilled beyond the server's lifetime, so that newly activated objects do not get assigned "old" Object Ids, which would result in a conflict when delivering requests.

For Object Id generation, a POA uses a `UniqueId` helper class, which is capable of encapsulating the current Id in a string. When a persistent POA is destroyed, it saves its state, the last-generated Object Id, in a global map. The Root POA, as part of server initialization or shutdown, stores and restores this table in the POA Manager, using the `save_state` and `restore_state` methods (see the IDL in figure 5.7).

An alternative would be to generate a unique prefix for Object Ids based, for example, on a timestamp. However, this is not as trivial as it seems, because a persistent POA can be created, destroyed and recreated within a process: a simple timestamp is not sufficient. With the side effect of being able to use simple Object Ids (a running number), this way of requiring one additional once-in-a-lifetime communication with the POA Mediator seemed a good compromise.

5.4.4 Usage

As with the extensions for the Interoperable Naming Service, a solution to the persistency question has been implemented that does not require any vendor-specific source code incompatibilities. The generic source code presented in [58] will work and produce the expected results without changes.

The only necessary efforts beyond the POA specification are administrative, to run the BOA Daemon and to add an entry to the Implementation Repository.

As a side note, the Implementation Repository also allows for coarse-grained mobility of servers. If the command to start the server is replaced with a remote shell invocation, the next instance of the server will be run remotely – and the client will not notice.

But again, we run into a bootstrapping problem, this time on the server side. In order to create object references and to export them, for example by entering them in the Naming Service, the first instance of the server has to be started manually, using one of three options:

- Forcing activation of a server using the Implementation Repository's administrative interface (`imr activate`).

- Starting the server on the command line, with the provision of the necessary command-line options (Implementation Name and the Implementation Repository's reference).

- Using the MICO Binder [50], which can also cause implementation activation.

If the second option is used, a server could also contact the Implementation Repository on its own – after all, its interface is specified in IDL – and add an entry for itself. This removes the need for adding an entry "by hand", but would introduce an incompatibility, since the layout of the Implementation Repository is MICO-specific.

It is also possible for a persistent server to run without contacting an Implementation Repository, but then clients will experience a "transient" exception when they try to send a request while the server is down. A third possibility is to create an entry in the Implementation Repository that leaves the implementation's executable undefined. The POAMediator is then unable to start a new server on demand, but will queue requests until a new server comes online by administrative action – the equivalent of the BOA's persistent activation policy.

It has been mentioned that the BOA Daemon should run permanently. MICO actually allows the daemon to be stopped and restarted. Upon shutdown, all connected servers are stopped, too, and their state information is written to a disk file, so that the system can be restored when the daemon is started again. However, clients that send a request while the daemon is down will also receive an exception indicating temporary failure.

5.5 Collocation

Location transparency is one of the corner stones of CORBA. The client is not aware of a servant's location, the ORB alone is responsible to read an object reference's profiles to contact the implementation. As described in section 3.3, auto-generated stub classes package their parameters in a CORBA request, which is then transported by the ORB and decoded by the skeleton on the server side.

This is fine for remote invocations, but suboptimal for local invocations, if the servant happens to be in the same process as the client, be it coincidence or by design. While performance optimizations might not be necessary for the random case, inefficient handling is annoying if a server knowingly manipulates local objects through CORBA calls.

An example is the Naming Service, which maintains a tree of "Naming Context" objects. These must be CORBA objects, since they must be accessible from remote. But if the Root context is asked to locate a node, it needs to traverse the tree, resulting in unsatisfactory performance if each step required a CORBA invocation to occur. In the local case, marshalling and unmarshalling parameters to and from a request seems overly complicated when the data could be used directly.

Effort to circumvent normal ORB request handling is well spent and allows for more scalability, because the user can easily migrate between efficient local invocations and ORB-mediated remote invocations.

Most ORBs, including MICO, implement collocation optimizations for BOA-based objects using the strategy shown in figure 5.10, which shows the inheritance for the stub and skeleton classes for interface "Account." By using the same inheritance from `CORBA::Object` and the same interface from the common base class `Account`, stub object and servant can be used interchangeably.

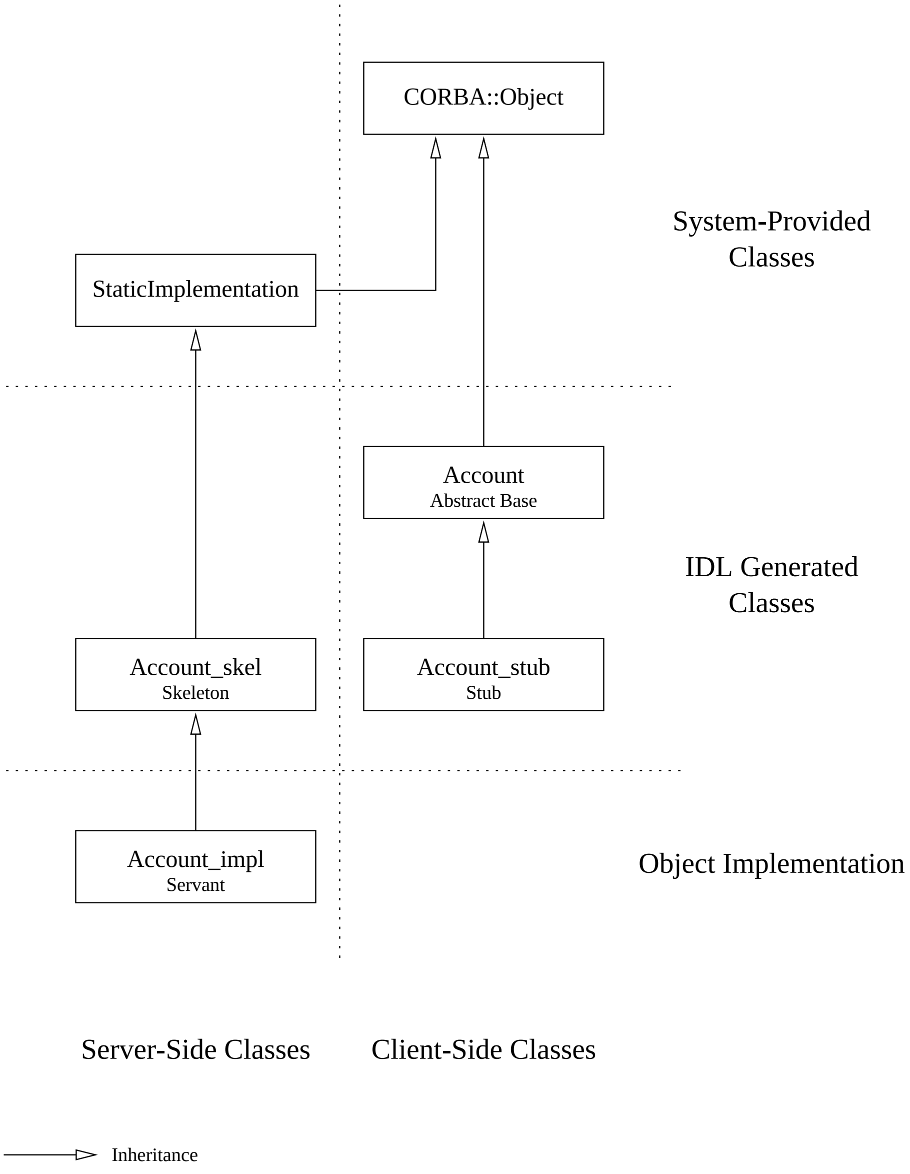

Figure 5.10: Common strategy for Collocation using the BOA

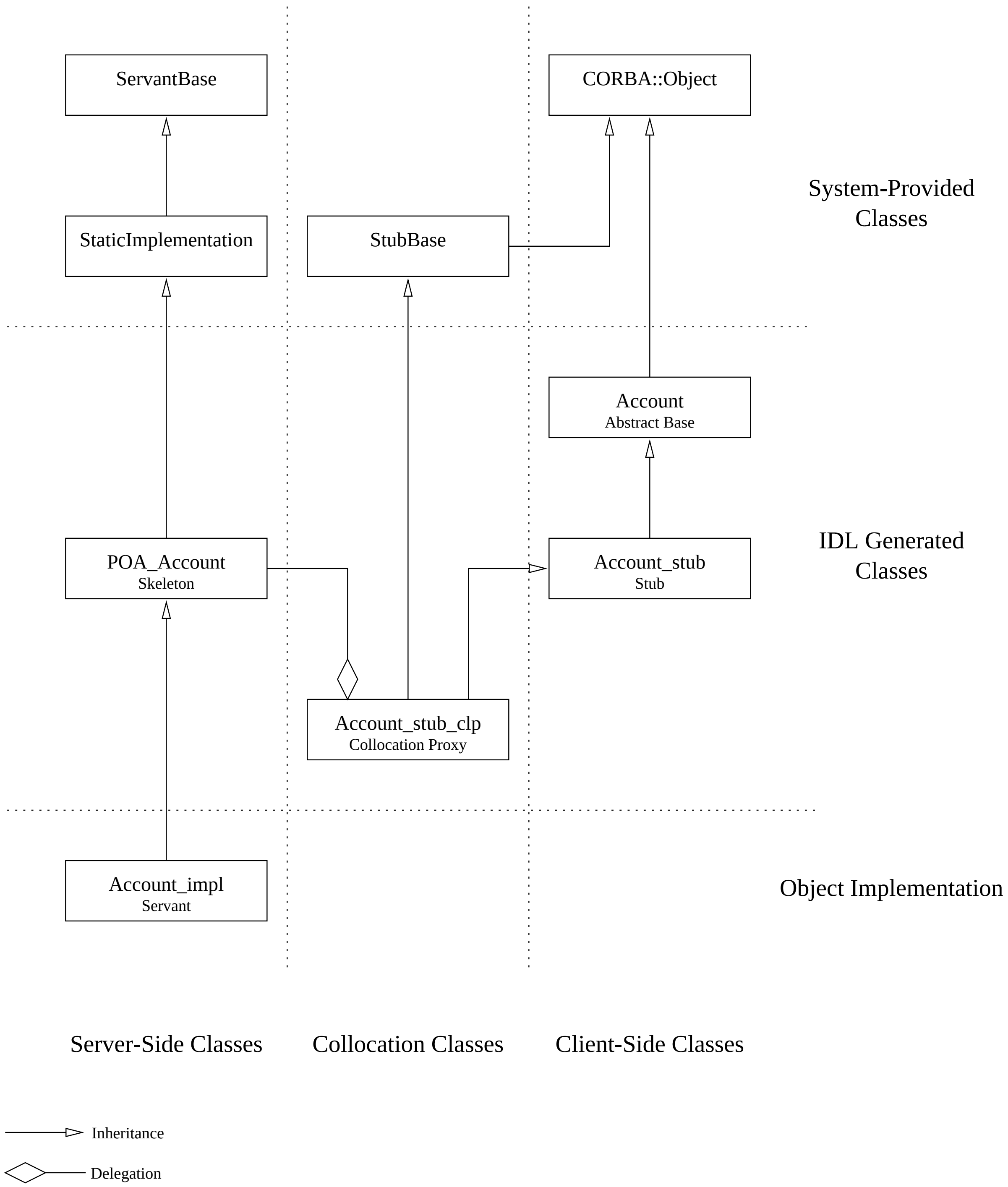

Figure 5.11: Collocation with the POA requires a Proxy

If a client receives an object reference, the ORB checks if the object is local, by comparing the reference's profiles with its own object reference template. If this comparison fails, the ORB falls back to generating a stub object. Otherwise, if the object is indeed local, it contacts the BOA to return the actual servant. The client cannot distinguish the servant from a stub: the interface matches, and the method signatures are the same.

The only difference is speed: if the client performs an invocation, the servant is called *directly*, thanks to C++'s abstract classes and virtual methods. The ORB is never involved, and no data is moved around; the location-transparent remote invocation decays to a local procedure call, without any overhead.

However, this inheritance-based approach is not possible with POA-based objects, for several reasons:

- The lifetimes of stubs and servants are independent of each other. A servant can be replaced, and stubs must reflect that change. This is not possible if "stubs" are actually pointers to the servant.
- Invocations must honor the POA's threading policy and the state of the POA Manager. It may be necessary to defer or to fail the request. With the above approach, invocations are always "delivered" immediately.
- The POA must be involved; for example, the `POACurrent` object must reflect the invocation, and the `_this` member must return the correct value.

Another obstacle raised in CORBA 2.3 is that valuetype parameters may need to be copied. As a solution, a proxy is introduced to control access to the servant [10]. For each interface, the IDL compiler creates a *collocation proxy* (`Account_stub_clp` in figure 5.11). This class is a stub that specializes the normal stub class in the case of a collocated servant. Whenever the client acquires an object reference as the result of a method invocation or ORB operation, the ORB produces a collocation proxy if the reference is local and if the servant is adapted through a POA.

For the purpose of collocation using the BOA, MICO already provided the `skeleton` method in the generic Object Adapter interface to produce the (BOA-based) servant. This part of the Object Adapter interface was first useless with the POA, but then found to be useful for collocation, as it made the process of acquiring a collocation proxy transparent to the client.

The collocation proxy uses delegation instead of inheritance to invoke the servant. Before an invocation is actually performed, the proxy has to cooperate with the servant's POA to find out if a direct procedure call can proceed:

1. Make sure the servant's POA that we refer to has not been destroyed.
2. Check that the POA is in the active state.
3. Ask the POA to retrieve the servant from its Active Object Map.
4. Verify that the servant is implemented through a static skeleton and not using the DSI.
5. Update the `POACurrent` settings to reflect the invocation in progress.

```
void
Account_stub_clp::deposit( CORBA::ULong amount )
{
  // The POA returns NULL here if either object or POA is not active
  PortableServer::Servant _serv = _poa->preinvoke (this);

  if (_serv) {
    // Object and POA are active, try narrowing to skeleton type
    POA_Account * _myserv = POA_Account::_narrow (_serv);
    if (_myserv) {
      // Perform local invocation
      _myserv->deposit(amount);
      _poa->_postinvoke ();
      return;
    }
    _poa->_postinvoke ();
  }

  // Object/POA Inactive or the servant is of unexpected type (DSI?)
  // Fall back to the normal stub and send a request.

  Account_stub::deposit(amount);
}
```

Figure 5.12: Abbreviated Sample code from a Collocation Proxy

Only after the checklist is complete, the proxy can delegate the invocation to the servant, by invoking the skeleton's virtual methods. If any of the above steps fails, the collection proxy falls back to the default stub's behavior by calling its parent class, where the parameters are marshalled into a CORBA request and fed to the ORB as usual, guaranteeing correct behavior in all circumstances. An example of the code generated by the IDL compiler is shown in figure 5.12, abbreviated by the removal failure handling for presentation purposes.

Performance measurements for a simple operation show that method invocation through the collocation proxy is about 40 times faster than an ORB-mediated invocation. This factor would actually be greater for operations with more complex parameters, because no marshalling needs to be done, and the complexity of the above steps is constant. However, if the computations performed in the operation become more complex and exceed the time saving from $200\mu s$ to $5\mu s$ (exemplary values on the author's system), the performance impact becomes less significant.

So far, the motivation for collocation has been performance, but collocation proxy classes are also convenient for locality-constrained objects, where the servant *must* be in the same process as the client.

Where not directed by the specification, MICO omits skeleton and stub classes when generating code for locality-constrained objects like the POA. There is only a single thread of inheritance, and the servant is called directly. This is not possible for servant managers and adapter activators, which the specification requires to follow the "usual" semantics, so that users can derive their object implementation from the skeletons. Yet no stubs and skeletons can be generated: the parameters handled by servant managers are native and cannot be marshalled.

In the first implementation, hand-written skeleton classes were used that could double as stubs, exploiting a special exception in the POA's activation mechanism, but bypassing POA mediation of method invocation. These were later replaced by code derived from an IDL-generated collocation proxy, with the inheritance from and delegation to the normal stub – which does not exist – removed.

Valuetypes as introduced by the Objects by Value specification in CORBA 2.3 [30] will present an additional challenge to the collocation mechanism. The major factor for increased performance in collocation is that a method's parameters do not need marshalling but can be used directly. This is not always possible with valuetypes, since they may be *shared*: the same valuetype can be referenced more than once, so that if the valuetype is changed using one reference (a C++ pointer), the other reference is affected, too. This is not in itself surprising but the same behavior as with normal, remote objects handled through object references.

But false behavior would result if a shared valuetype is passed as an inout parameter to a collocation proxy. If the shared valuetype was passed to the servant and there modified, as explicitly allowed in the C++ mapping [31], the other copy of the valuetype would, as an invalid *side effect*, change, too.

The collocation proxy must therefore check if valuetypes passed as an inout parameter are shared, and create a duplicate if so. Not only is the duplication of a valuetype complex in itself, – it frequently requires marshalling and unmarshalling of the valuetype – but introduces new problems, as valuetypes can also be shared across parameters. If the same valuetype is used twice as parameter in a method invocation, the servant must receive a shared parameter, too. An example is shown in figure 5.13, where the same valuetype is passed both as the first and the second parameter. If the second parameter needs to be copied in the collocation proxy, the sharing must be detected, and the copy must be passed as first parameter, too,

```
// IDL

valuetype Foo { ... };

interface Bar {
  Foo Op (in Foo f1, inout Foo f2);
};

// C++

  Foo_var f1, f2;
  ...

  f2 = f1;
  f1 = barRef->Op (f1.in(), f2.inout());
```

Figure 5.13: Valuetypes (Objects by Value) can be shared across Parameters.

when invoking the servant.

Parameter sharing is not always as obvious as in this example, as valuetypes can describe complex graphs, and sharing may occur between nodes of different graphs, between members of other types like structures or arrays, further masked by inheritance: a base valuetype can be shared with a derived valuetype.

Code has been added to the IDL generator to make the collocation proxy as smart as possible in detecting parameter sharing. If the IDL generator decides from a parameter's signature that either no or easily detectable parameter sharing can occur, extra code is produced to make copies of shared parameters as necessary. But if the possibilities become too overwhelming, the IDL compiler gives up and does not generate a collocation proxy at all.

Because of the many issues involved, the code involved both in the IDL generator and the generated code is complex and has not yet been fully verified, and maybe it would be sensible to keep the code small and to skip collocation optimizations altogether where valuetypes are involved.

5.6 Other Implementations

At the time of writing, more than a year after the release of the CORBA 2.2 specification, the Portable Object Adapter is still not widely available. IONA Technologies, the market leader according to number of sales, has just released the new major version 3.0 of their Orbix product, still without a POA. So far, the only two commercial ORBs that claim to come with a POA are ObjectBroker from BEA Systems and DAIS from PeerLogic. Both are not freely available, and information about them is scarce.

However, two other Open Source ORBs, ORBit from Redhat Labs and TAO from Washington University, have evolved a POA implementation in parallel to the one in Mico. Both can be downloaded and distributed in source under liberal licenses, so their implementation of the POA could be examined.

A third candidate for comparison is ORBacus from Object Oriented Concepts, a commercial ORB that can be downloaded in source for private use.

The goal of this comparison was not to search and expose minor bugs, but rather to make a structural comparison of the different design strategies and to see how other ORBs handle the topics of object key handling, persistence and collocation.

The development of all three ORBs is not finished. While ORBit has not changed over the last three months, new versions of TAO appear almost weekly, and ORBacus is still in the alpha stage. The most recent versions as of mid-June 1999 have been downloaded and checked, but the information presented here could be outdated soon.

5.6.1 ORBit

This ORB from Redhat Labs [43] was available in version 0.4, and it is interesting to look at its history. ORBit was developed as a basis for the distribution of components in the GNOME desktop project [12], using some of the same basic ideas as in the KDE project [19]. The KDE project has used MICO from the beginning, and so did GNOME. However, their developers have been unimpressed with the performance of MICO and so decided to start writing an ORB of their own.

Being very concerned with performance, the GNOME project does not use C++ but plain C, and consequently, ORBit provides a C language mapping only.[5]

ORBit is still a work in progress, and while it has reached maturity in that it can be used to implement clients and servers, many advanced features are not yet implemented. In the POA implementation, some of the identity mapping operations are missing, as is support for virtual objects, adapter activators or the `POACurrent` interface.

The ORBit POA does not encode the full POA name into the object key, but rather a numerical identifier. A global map then translates this POA Id back to the POA instance. While making for shorter Object Ids, this will require ORBit to keep state information for old POAs – once adapter activators are implemented. In the unlikely situation that an implementation routinely creates and deletes POAs, the internal tables in ORBit will grow, while MICO keeps all information in the clients, should they decide to keep an object reference.

The lifespan policy is wholly unimplemented, and the way object keys are generated (again using a numerical identifier that points into an array of Object Ids) does not satisfy either persistency or transiency.

On the other hand, ORBit does include collocation optimizations, albeit using the BOA-based approach as shown in figure 5.10. Invocations on a collocated object bypass both ORB and POA.

A striking feature of ORBit is its size of just 10.000 lines of code[6] for the ORB core and its components, including the POA. For users who can live with the C language mapping and without advanced CORBA features, ORBit is a lean and efficient ORB.

5.6.2 TAO

The tested version of The ACE ORB [61] from Washington University was 0.3.23 from May 22, 1999 and comes with a full-featured POA.

[5] A C++ language mapping is, according to the ORBit home page, in very early stages of development.
[6] Comparable numbers are 50.000 for MICO and 60.000 for both TAO and ORBacus.

TAO already supports persistent objects, but does not yet allow for on-demand startup of new servers. The distribution includes a design paper for an Implementation Repository and some preliminary code.

A problem of relying on ORB extensions has already been mentioned in section 5.3, where a new methods was introduced in order to support short and legible object keys for the Interoperable Naming Service.

The focus of TAO development is its applicability in real-time systems, and TAO's developers are spending much thought on algorithms to provide runtime guarantees [51]. The POA does have to maintain its own data structures like the Active Object Map, which needs to be traversed upon invocation. As mentioned in section 5.2, the Mico POA uses an STL map to provide an average case behavior of $O(\log n)$, which may not be enough for hard real-time applications.

The idea used by TAO is that servers are commonly static, i.e. the number of POAs and active servants is known beforehand, at design time. TAO uses this knowledge to compute a *perfect hashing* function on the expected set of Object Ids to guarantee constant-time lookup in the Active Object Map. In the skeleton code, again perfect hashing is used in the dispatching of a request to an object implementation's method.

Like ORBit, TAO keeps the names of active POA instances in a global map and stores an index into that table in the object key, but avoids ORBit's pitfall with adapter activators by additionally adding the full POA name, in case the POA is inactive.

TAO also implements collocation optimizations using a collocation proxy. However, unlike in Mico, TAO's collocation proxies are separate from the normal stub and only forward the method invocation to the servant [51], and do not determine the servant's readiness to receive requests. Again, the ORB and the POA are bypassed. In order to provide correct behavior, the `-ORBcollocation` command-line switch can be used to disable collocation optimization altogether – but TAO defaults to potentially unsafe direct calls.

Unfortunately, TAO uses an alternative C++ mapping, because it still supports old C++ compilers without exception support.[7] In an object implementation, methods take an additional parameter to carry an exception. While client code could be exchanged between Mico and TAO, the first interoperability test with server-side source code failed.

With the necessary changes made, – adding a default parameter to each method in the object implementation – server code could indeed be ported and ran on the other ORB with the expected results.

5.6.3 ORBacus

In the evolution of commercial software, some design decisions cannot be reversed for the sake of backwards compatibility. Over time, it becomes ever more complicated to add new features that would have been straightforward to implement if old designs could be replaced.

For similar reasons, ORBacus is being largely rewritten to get rid of historic restrictions like an alternative mapping for C++ compilers without namespaces and the tight integration of the BOA into the ORB. The latest available version was the 4.0a2 alpha release; the final version 4.0 is promised to comply to CORBA 2.3.

Already, the POA support seems reasonably complete, no omissions could be found. Missing is an Implementation Repository; while persistent objects are supported and properly

[7] See "Alternative Mappings for C++ Dialects" in the CORBA 2.2 C++ language mapping.

POA Feature	MICO	ORBit	TAO	ORBacus
Basic Features	yes	(yes)	yes	yes
INS Support	(yes)	no	(yes)	no
Persistence	yes	no	yes	yes
Implementation Repository	yes	no	(yes)	no
Collocation Optimization	yes	(yes)	(yes)	(yes)

Table 5.1: ORB Comparison

implemented, their servers must be started manually.

Like ORBit and TAO, ORBacus bypasses ORB and POA in its implementation of collocation optimizations, with the slight improvement that at least the `POACurrent` interface reflects the invocation in progress.

No provisions for legible object keys for use with the Interoperable Naming Service exist.

Unlike TAO, ORBacus uses the normal C++ mapping, and so it was possible to exchange both client and server source code between MICO and ORBacus. The one item that still needed changing in both client and server source code were the names of IDL-generated include files, which are not prescribed by the specification and can therefore vary between ORBs.

5.6.4 Comparison

The comparison of ORBit, TAO and ORBacus demonstrates that despite the apparent standardization of the CORBA platform, each ORB is individual with its own ideas and intentions.

Table 5.1 shows the POA-related features highlighted in this chapter and whether they are realized in the other ORBs. A (yes) in parenthesis means that the feature exists but has not been fully implemented or requires vendor-specific extensions.

For fairness, it should be added that the table does not list other ORB features like real-time support or multithreading which are available in other ORBs but not in MICO – but such features are beyond the scope of this thesis.

5.7 Evaluation

The reference implementation of the POA in MICO has in most areas verified the validity of the POA specification. The description of the POA's methods and their behavior is precise with little margin for error or misinterpretation, and in this respect, server-side interoperability is achieved.

Porting server-side code between ORB implementations is still not fully possible, though. Some ORBs like TAO continue to support old environments by using an alternate C++ mapping, but even if the same mapping is used, as in the case of ORBacus, the names of IDL-generated files differ, so that the include directives at the beginning of each source file need modification. If the names of generated files could be agreed on, client-side and server-side source code could indeed be transported between ORBs without any modification.

However, the required source file modifications are trivial compared with the porting of BOA-based code. The POA not only provides the programmer with a consistent interface, but also with consistent design, whereas the proprietary extensions that existed for the BOA as often as not required a complete redesign and rewriting of server code from scratch when being ported to a different ORB.

It is interesting and somewhat ironic that the one missing piece of the POA specification, the Implementation Repository, was removed with the BOA specification, so vendor-specific extensions are again required to support persistency.

But a distinction must be made between vendor-specific source-level extensions like additional methods to the ORB or POA, and vendor-specific administrative action. The latter can be deemed acceptable – each vendor should be able to define a custom set of commands and options, as long as the source code is portable.

It has been demonstrated that persistency and the synchronization of active servers with the Implementation Repository is possible without any source-level changes, but with administrative action only: The Mico daemon must be run, the implementation must be registered with the IMR, and the `-POAImplName` option must be added to a persistent server's command line.

The Interoperable Naming Service and its requirement of short and legible object keys is bound to cause more trouble. While the Mico POA takes care to use acceptable object keys by default, again without any ORB modifications as in TAO, the Mico solution relies on cooperation with the developer, who must choose the "right" POA names and Object Ids. The developer must rely on the Mico POA to construct the object key in a foreseeable manner. In this respect, the solution is just as unportable as TAO's added ORB method.

It is to be expected that each vendor's solution for constructing object keys will be different, so that service implementations that wish to publish an URI-style IOR are again non-portable.

Since both solutions, the generation of object keys or a new ORB method to keep an object key mapping table, are unlikely to be standardized, one idea would be the introduction of a new standard "trivial" object adapter. It could use Object Ids directly as the object key, with the sole purpose of forwarding incoming requests to the proper POA-based object.

Chapter 6

Conclusions

The goal of a reference POA implementation has been achieved, proving the sensibility of the specification. The MICO POA is, in terms of features and correctness, equivalent or superior to the few other current POA implementations that have evolved in parallel in other ORBs.

The extensibility of the microkernel ORB has also been demonstrated satisfactorily, as the addition of the POA did not require any modification of the ORB and could be put to work just by implementing the generic Object Adapter interface. Modification of the IDL compiler was necessary, but this was hardly unexpected with the different layout of POA skeleton classes.

The microkernel architecture and its support for multiple object adapters is a unique feature of MICO, as the comparison with other ORBs shows. The development of ORBit started after CORBA 2.2, and it never had a Basic Object Adapter. Version 0.0.4 of TAO did have a BOA, but it was abandoned and completely replaced with the POA. ORBacus has had its BOA removed as part of its transistion process to the new major version. MICO is the only ORB where BOA and POA – among other, less user-visible, object adapters – coexist. As verified in their source code, none of the other monolithic ORBs allows more than one object adapter.

The POA was the missing piece for MICO to achieve CORBA 2.2 compliance. Its source code was contributed to the main distribution in September 1998 for the release of MICO 2.2.0 and was included in the second version of the MICO book. Some of the more advanced features have been added in later revisions, notably persistence in 2.2.3 and collocation in 2.2.4. Since then, it has been widely distributed and put to use; some minor bugs were reported on the mailing list and quickly fixed. Some enhancements like a more efficient handling of the Active Object Map are also the result of open discussion.

As a real-life test, the Naming Service, originally contributed by Kai-Uwe Sattler, was rewritten to use the Portable Object Adapter without any noticable difference of behavior. A new experimental language mapping for the Tcl scripting language also makes heavy use of the POA. In fact, it provides scripts with the full POA interface and was used in the testing of the POA implementation. The K Office Suite, the "killer application" for the K Desktop Environment [19], already exploits the POA's advanced features like servant managers and request forwarding.

A lot of BOA-based code remains in MICO, but this is more a feature than a problem. Since BOA and POA coexist, there is no need to spend effort in replacing the old BOA-based code.

While the decision whether to use the POA or BOA is easy in terms of features, it can be answered differently in terms of literature. Existing books are still mostly focused on the BOA, and the documentation that comes with the Mico distribution is no exception. A chapter about POA programming has been contributed, but is placed less prominently behind the BOA part. Otherwise, the only documentation for using the POA has been the specification itself. Only one recent book [16] deals with the POA.

The source code for the POA is complete, though coming CORBA revisions will most likely bring minor changes, as CORBA 2.3 does. As this thesis has shown, the POA might endure longer than the Basic Object Adapter did, which survived for seven years despite its shortcomings. On the other hand, CORBA now grows faster than it did ever before, and new applications of the CORBA architecture will probably need new object adapters, too.

The Portable Object Adapter is successful in its own field, for the implementation of servers in a Client/Server system. Real-time systems, data streaming services, parallel or fault-tolerant systems all have different requirements, and it is possible that new object adapters can fulfill their needs better than the POA does.

In the meantime, the POA is central to many new CORBA features. Objects by Value and Messaging both make extensive use of the POA, Objects by Value for supporting interfaces, and Messaging for callback objects.

CORBA has in the past often suffered from weak and incorrect ORB implementations. The test suite from the Open Group has only been passed by three ORBs so far, including Mico. However, the suite covered CORBA 2.1 only, and so the POA has not been verified. It would be interesting to have a CORBA 2.2 test suite, so that remaining bugs could be detected and expelled, and to ensure that the POA implementations of different ORBs are indeed compatible.

Appendix A

Glossary

A.1 Abbreviations

AMI	Asynchronous Method Invocation
ANSI	American National Standards Institute
API	Application Program Interface
ATM	Asynchronous Transfer Mode
BOA	Basic Object Adapter
BSD	Berkeley System Distribution
CDR	Common Data Representation
CORBA	Common Object Request Broker Architecture
DCE	Distributed Computing Environment
DCOM	Distributed Component Object Model
DII	Dynamic Invocation Interface
DSI	Dynamic Skeleton Interface
GIOP	General Inter-ORB Protocol
GUID	Globally Unique Identifier
IANA	Internet Assigned Numbers Authority
IDL	Interface Definition Language
IEEE	Institute of Electrical and Electronics Engineers
IFR	Interface Repository
IIOP	Internet Inter-ORB Protocol
IMR	Implementation Repository
INS	Interoperable Naming Service
IOR	Interoperable Object Reference
IP	Internet Protocol
ITU	International Telecommunication Union

MICO	MICO is CORBA
NFS	Network File System
NNTP	Network News Transport Protocol
OBV	Objects by Value
OMA	Object Management Architecture
OMG	Object Management Group
ORB	Object Request Broker
OSI	Open System Interconnection
PID	Process Id
POA	Portable Object Adapter
POSIX	Portable Operating System Interface
PVM	Parallel Virtual Machine
RFC	Request For Comments
RFP	Request For Proposal
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SII	Static Invocation Interface
SSI	Static Skeleton Interface
SSL	Secure Socket Layer
STL	Standard Template Library
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
X11	X Window System, Version 11
XDR	External Data Representation

A.2 Terminology

Note: For more POA terminology, see chapter 4.2.

Activation The act of associating a type, servant and a user-supplied or generated Object Id in order to create an object. The servant may be left empty in order to create a new virtual object.

Basic Object Adapter (BOA) The only object adapter specified by CORBA versions up to and including 2.1. Removed from the specification in CORBA 2.2.

Client The program that acquires client objects and performs CORBA invocations.

Collocation Proxy A special type of stub used if the servant is in the same process as the client. The collocation proxy sets up the environment and performs a direct (local) procedure call into the servant if possible, else it falls back to normal stub behavior of sending a request through the ORB.

Common Data Representation (CDR) Standardized "on-the-wire" representation for IDL data types, taking padding and endianness issues into account. Used by the General Inter-ORB Protocol.

Deactivation Removes an object from the active object map. See etherealization. Deactivated objects become virtual and may be reincarnated at a later point in time.

Dynamic Invocation Interface (DII) A standardized client-side interface for the marshalling of parameters into a request and invoking methods using run-time type information. Unlike the Static Invocation Interface, the DII allows for asynchronous method invocations by first sending a request and then waiting or polling for the result.

Dynamic Skeleton The skeleton class that an object implementation must be derived from if it is to use the Dynamic Skeleton Interface.

Dynamic Skeleton Interface (DSI) A standardized server-side interface for the unmarshalling of parameters from a request and retrieval of other request-specific data using run-time type information.

Etherealization The act of disassociating a servant from an object, so that only a virtual object remains. Etherealization is triggered by an object's deactivation. Both terms are somewhat synonymous: an object is *deactivated*, and the servant is *etherealized*.

General Inter-ORB Protocol (GIOP) The standardized request-response protocol used to send *requests* and to exchange control messages between ORBs. Uses Common Data Representation for the exchange of data and Interoperable Object References for addressing.

IDL Compiler A program that reads descriptions from Interface Definition Language files and produces type declarations, stubs and skeletons for a target programming language according to the language mapping.

Internet Inter-ORB Protocol (IIOP) The application of the General Inter-ORB Protocol to TCP/IP.

Implementation In a POSIX environment, an implementation is an executable program. When executed, an implementation becomes a server. The term is also frequently used to denote the user code that, when activated, becomes a servant. This document uses the term *object implementation* instead to emphasize the difference.

Implementation Name A unique name that must be assigned to an implementation if it contains persistent objects. The Implementation Name is set using the `-POAImplName` option on the command line. (MICO-specific)

Implementation Repository (IMR) A database containing information about available implementations and the objects they implement.

Incarnation The act of associating a virtual object with a servant, usually in a servant activator.

Interface Definition Language (IDL) The language used to describe an object's interface. This description is translated by the IDL compiler into a programming language's type declarations, stubs and skeletons according to the language mapping.

Interface Repository (IFR) A database containing type information for interfaces, their data structures, operations and parameters, filled in from IDL files, usually by the IDL compiler. Can be used by clients or servers to retrieve type information at runtime.

Interoperable Object Reference (IOR) A standardized format for object references.

Language Mapping A language mapping describes how the types and declarations from the IDL language are expressed and how clients and servants can be implemented in a particular programming language.

Marshalling The packaging of a method's parameters into a request.

Object A CORBA object is an abstract entity with a public interface and internal state. An object "exists" on the server side; the difference to the term "servant" is that the same servant can be used to implement many objects at once. Logically, an object is a three-tuple consisting of a type, a unique Object Id and a servant. Clients do not handle objects, they handle object references.

Object Adapter The layer that exists on the server side to mediate between the ORB's request processing and the servants' readiness to receive requests.

Object Id The part of an object key that identifies a particular servant within its object adapter. Can be either user-selected or assigned by the object adapter itself.

Object Implementation The user-provided implementation code realizing all methods of a specific interface. In the C++ language mapping, an object implementation is a C++ class that inherits either from an IDL-generated skeleton or from the Dynamic Skeleton base class. A servant is an instance of an object implementation.

Object Key The part of an object reference (or more precisely, of a profile) that identifies a particular object adapter within the server and a servant within that object adapter, using the object Id.

Object Reference The "address" of a specific servant. An object reference contains a type and one or more profiles. To the user, an object reference is opaque data and cannot be examined or constructed.

Object Request Broker (ORB) The ORB is basically only the entity that takes requests from a client and delivers them to a potentially remote servant. Sometimes called the "ORB core," to distinguish this rudimentary task from the other user services also provided by the ORB, such as the stringification of object references. The term "ORB" is also frequently used to denote the sum of all non-user CORBA components, including the object adapters, dynamic invocation interface etc.

Persistent Object A persistent object has been activated in a POA with the "persistent" lifespan policy. Persistent objects can outlive the server they were originally created in, so the server can be stopped and restarted transparently, usually by an Implementation Repository.

Portable Object Adapter (POA) A powerful object adapter introduced in CORBA 2.2.

Profile The part of an object reference that identifies a particular servant, consisting of location information (in the case of an IIOP, an Internet host name and TCP port number) and the object key.

Request A request is sent from the client to the server as part of method invocation, encapsulating the object's identity (object reference), the method name and the parameters. The request is dispatched to a servant, and the result is then returned in a reply message.

Servant A servant is an instance of an object implementation. Servants must be activated with an Object Adapter. After activation, object references can be created for that servant.

Server A server contains one or more servants. In a POSIX environment, a server corresponds to a separate process.

Skeleton The server-side programming-language code (C++ class) generated for a particular interface by the IDL compiler. Skeletons are abstract, the programmer must derive from the skeleton and add the methods' implementation. Instances of a class that derives from skeleton are servants.

Static Invocation Interface A vendor-specific interface for the marshalling of parameters, employed by IDL-generated stubs. Proprietary, but usually much faster than the Dynamic Invocation Interface.

Static Skeleton Interface A vendor-specific interface for the unmarshalling of parameters, employed by IDL-generated skeletons. Proprietary, but usually much faster than the Dynamic Skeleton Interface.

Stub The client-side programming-language code (concrete C++ class) generated for a particular interface by the IDL compiler. Stub objects are instances of a stub.

Stub Object Encapsulates an object reference and incarnates a specific most-derived interface. To the programming language, it presents the same interface as declared in the

IDL file. The code for stub objects is generated by the IDL compiler. A stub objects' methods use the Static Invocation Interface to package its parameters into a CORBA request and causes a remote invocation on the servant.

Transient Object A transient object is an object that has been activated in a POA with the "transient" lifespan policy. Its lifespan is bounded by the POA it was activated in, and ceases to exist if its POA instance is destroyed, for example as part of server shutdown. *Note: this definition has changed in CORBA 2.3; according to CORBA 2.2, a transient object's lifespan is bounded by the lifespan of its server process.*

Virtual Object A virtual object has not been incarnated yet. The POA allows to create object references to virtual objects. The server could incarnate the object at a later point in time, or on demand by using a servant manager.

Appendix B

Examples

The following section is an excerpt of the text contributed to the MICO manual, containing source code examples of using the POA. Some passages with basic explanations (policies, terminology) have been omitted to not repeat the text of the previous chapters. More examples can be found in [16] or [59].

B.1 POA

The Basic Object Adapter provides a bare minimum of functionality to server applications. As a consequence, many ORBs added custom extensions to the BOA to support more complex demands upon an object adapter, making server implementations incompatible among different ORB vendors. In CORBA 2.2, the new *Portable Object Adapter* was introduced. It provides a much-extended interface that addresses many needs that were wished for, but not available with the original BOA specification. POA features include:

- Support for transparent activation of objects. Servers can export object references for not-yet-active servants that will be incarnated on demand.
- Allow a single servant to support many object identities.
- Allow many POAs in a single server, each governed by its own set of *policies.*
- Delegate requests for non-existent servants either to a default servant, or ask a servant manager for an appropriate servant.

These features, make the POA much more powerful than the BOA and should fulfill most server applications' needs. As an example, object references for some million entries in a database can be generated, which are all implemented by a single default servant.

B.1.1 Example

As an example, let's write a simple POA-based server. You can find the full code in the `demo/poa/hello-1` directory in the MICO distribution. Imagine a simple IDL description in the file "hello.idl":

```
interface HelloWorld {
  void hello ();
};
```

The first step is to invoke the IDL to C++ compiler in a way to produce skeleton classes that use the POA:

```
idl --poa --no-boa hello.idl
```

The first option, `--poa`, turns on code generation for POA-based skeletons. The second option, `--no-boa` optionally turns off code generation for the old BOA-based skeletons. Next, we rewrite the server.

```
 1: // file server.cc
 2:
 3: #include "hello.h"
 4:
 5: class HelloWorld_impl : virtual public POA_HelloWorld
 6: {
 7:   public:
 8:     void hello() { printf ("Hello World!\n"); };
 9: };
10:
11:
12: int main( int argc, char *argv[] )
13: {
14:   CORBA::ORB_var orb = CORBA::ORB_init (argc, argv, "mico-local-orb");
15:   CORBA::Object_var poaobj = orb->resolve_initial_references ("RootPOA");
16:   PortableServer::POA_var poa = PortableServer::POA::_narrow (poaobj);
17:   PortableServer::POAManager_var mgr = poa->the_POAManager();
18:
19:   HelloWorld_impl * servant = new HelloWorld_impl;
20:
21:   PortableServer::ObjectId_var oid = poa->activate_object (servant);
22:
23:   mgr->activate ();
24:   orb->run();
25:
26:   poa->destroy (TRUE, TRUE);
27:   delete servant;
28:   return 0;
29: }
```

The object implementation does not change much with respect to a BOA-based one, the only difference is that `HelloWorld_impl` does not inherit from the BOA-based skeleton `HelloWorld_skel` any more, but from the POA-based skeleton `POA_HelloWorld`.

In `main()`, we first initialize the ORB, then we obtain a reference to the Root POA (lines 15–16) and to its POA Manager (line 17).

Then, we create an instance of our server object. In line 21, the servant is activated. Since the Root POA has the `SYSTEM_ID` policy, a unique Object Id is generated automatically and returned. At this point, clients can use the MICO binder to connect to the HelloWorld object.

However, client invocations upon the HelloWorld object are not yet processed. The Root POA's POA Manager is created in the holding state, so in line 23, we transition the POA Manager, and therefore the Root POA, to the active state. We then enter the ORB's event loop in 24.

In this example, `run()` never returns, because we don't provide a means to shut down the ORB. If that ever happened, lines 26–27 would first destroy the Root POA. Since that deactivates our active HelloWorld object, we can then safely delete the servant.

Since the Root POA has the `IMPLICIT_ACTIVATION` policy, we can also use several other methods to activate the servant instead of `activate_object()`. We could, for example, use `servant_to_reference()`, which first implicitly activates the inactive servant and then returns an object reference pointing to the servant. Or, we could invoke the servant's inherited `_this()` method, which also implicitly activates the servant and returns an object reference.

B.1.2 Using a Servant Manager

While the previous example did introduce the POA, it did not demonstrate any of its abilities – the example would have been just as simple using the BOA.

As a more complex example, we want to show a server that generates "virtual" object references that point to non-existent objects. We then provide the POA with a servant manager that incarnates the objects on demand.

We continue our series of "Account" examples. We provide the implementation for a Bank object with a single "create" operation that opens a new account. However, the Account object is not put into existence at that point, we just return a reference that will cause activation of an Account object when it is first accessed. This text will only show some code fragments; find the full code in the `demo/poa/account-2` directory.

The implementation of the Account object does not differ from before. More interesting is the implementation of the Bank's `create` operation:

```
Account_ptr
Bank_impl::create ()
{
  CORBA::Object_var obj = mypoa->create_reference ("IDL:Account:1.0");
  Account_ptr aref = Account::_narrow (obj);
  assert (!CORBA::is_nil (aref));
  return aref;
}
```

The `create_reference()` operation on the POA does not cause an activation to take place. It only creates a new object reference encapsulating information about the supported interface and a unique (system-generated) Object Id. This reference is then returned to the client.

Now, when the client invokes an operation on the returned reference, the POA will first search its Active Object Map, but will find no servant to serve the request. We therefore implement a servant manager, which will be asked to find an appropriate implementation.

There are two types of servant managers: a *Servant Activator* activates a new servant, which will be retained in the POA's Active Object Map to serve further requests on the same object. A *Servant Locator* is used to locate a servant for a single invocation only; the servant will not be retained for future use. The type of servant manager depends on the POA's Servant Retention policy.

In our case, we use a servant activator, which will incarnate and activate a new servant whenever the account is used *first*. Further operations on the same object reference will use the already active servant. Since the `create_reference()` operation uses a unique Object Id each time it is called, one new servant will be incarnated for each Account.

A servant activator provides two operations, `incarnate` and `etherealize`. The former one is called when a new servant needs to be incarnated to serve a previously unknown Object

Id. etherealize is called when the servant is deactivated (for example in POA shutdown) and allows the servant manager to clean up associated data.

```
class AccountManager : public virtual POA_PortableServer::ServantActivator
{ /* declarations */ };

PortableServer::Servant
AccountManager::incarnate (/* params */)
{
  return new Account_impl;
}

void
AccountManager::etherealize (PortableServer::Servant serv,
                             /* many more params */)
{
  delete serv;
}
```

Our servant activator implements the POA_PortableServer::ServantActivator interface. Since servant managers are servants themselves, they must be activated like any other servant (see below).

The incarnate operation has nothing to do but to create a new Account servant. incarnate receives the current POA and the requested Object Id as parameters, so it would be possible to perform special initialization based on the Object Id that is to be served.

etherealize is just as simple, and deletes the servant. In "real life", the servant manager would have to make sure that the servant is not in use anywhere else before deleting it. Here, this is guaranteed by our program logic.

The main() code is a little more extensive than before. Because the Root POA has the USE_ACTIVE_OBJECT_MAP_ONLY policy and does not allow a servant manager, we must create our own POA with the USE_SERVANT_MANAGER policy.

```
CORBA::ORB_var orb = CORBA::ORB_init (argc, argv, "mico-local-orb");
CORBA::Object_var poaobj = orb->resolve_initial_references ("RootPOA");
PortableServer::POA_var poa = PortableServer::POA::_narrow (poaobj);
PortableServer::POAManager_var mgr = poa->the_POAManager();

CORBA::PolicyList pl;
pl.length(1);
pl[0] = poa->
  create_request_processing_policy (PortableServer::USE_SERVANT_MANAGER);
PortableServer::POA_var mypoa = poa->create_POA ("MyPOA", mgr, pl);
```

Note that we use the Root POA's POA Manager when creating the new POA. This means that the POA Manager has now control over both POAs, and changing its state affects both POAs. If we passed NULL as the second parameter to create_POA(), a new POA Manager would have been created, and we would have to change both POA's states separately.

We can now register the servant manager.

```
AccountManager * am = new AccountManager;
PortableServer::ServantManager_var amref = am->_this ();
mypoa->set_servant_manager (amref);
```

After creating an instance of our servant manager, we obtain an object reference using the inherited `_this()` method. This also implicitly activates the servant manager in the Root POA.

```
Bank_impl * micocash = new Bank_impl (mypoa);
PortableServer::ObjectId_var oid = poa->activate_object (micocash);
mgr->activate ();
orb->run();
```

Now the only thing left to do is to activate a Bank object, to change both POAs to the active state, and to enter the ORB's event loop.

B.1.3 Persistent Objects

Our previous examples used "transient" objects which cannot outlive the server process they were created in. If you write a server that activates a servant and export its object reference, and then stop and re-start the server, clients will receive an exception that their object reference has become invalid.

In many cases it is desirable to have persistent objects. A persistent object has an infinite lifetime, not bound by the process that implements the object. You can kill and restart the server process, for example to save resources while it is not needed, or to update the implementation, and the client objects will not notice as long as the server is running whenever an invocation is performed.

An object is persistent if the servant that implements them is activated in a POA that has the `PERSISTENT` lifespan policy.

As an example, we will expand our Bank to create persistent accounts. When the server goes down, we want to write the account balances to a disk file, and when the server is restarted, the balances are read back in. To accomplish this, we use a persistent POA to create our accounts in. Using a servant manager provides us with the necessary hooks to save and restore the state: when etherealizing an account, the balance is written to disk, and when incarnating an account, we check if an appropriately named file with a balance exists.

We also make the Bank itself persistent, but use a different POA to activate the Bank in. Of course, we could use the Accounts' POA for the Bank, too, but then, our servant manager would have to discriminate whether it is etherealizing an Account or a Bank: using a different POA comes more cheaply.

The implementation of the Account object is the same as in the previous examples. The Bank is basically the same, too. One change is that the `create` operation has been extended to activate accounts with a specific Object Id – we will use an Account's Object Id as the name for the balance file on disk.

We also add a `shutdown` operation to the Bank interface, which is supposed to terminate the server process. This is accomplished simply by calling the ORB's shutdown method:

```
void
Bank_impl::shutdown (void)
{
  orb->shutdown (TRUE);
}
```

Invoking `shutdown()` on the ORB first of all causes the destruction of all object adapters. Destruction of the Account's POA next causes all active objects – our accounts – to be etherealized by invoking the servant manager. Consequently, the servant manager is all we need to save and restore our state.

One problem is that the servant manager's `etherealize()` method receives a `PortableServer::Servant` value. However, we need access to the implementation's type, `Account_impl*`, to query the current balance. Since CORBA does not provide narrowing for servant types, we have to find a solution on our own. Here, we use an STL map mapping the one to the other:

```
class Account_impl;
typedef map<PortableServer::Servant,
  Account_impl *,
  less<PortableServer::Servant> > ServantMap;
ServantMap svmap;
```

When incarnating an account, we populate this map; when etherealizing the account, we can retrieve the implementation's pointer.

```
PortableServer::Servant
AccountManager::incarnate (/* params */)
{
  Account_impl * account = new Account_impl;
  CORBA::Long amount = ...  // retrieve balance from disk
  account->deposit (amount);

  svmap[account] = account; // populate map
  return account;
}

void
AccountManager::etherealize (PortableServer::Servant serv,
                             /* many more params */)
{
  ServantMap::iterator it = svmap.find (serv);
  Account_impl * impl = (*it).second;
  ... // save balance to disk
  svmap.erase (it);
  delete serv;
}
```

Please find the full source code in the `demo/poa/account-3` directory.

One little bit of magic is left to do. Persistent POAs need a key, a unique "implementation name" to identify their objects with. This name must be given using the `-POAImplName` command line option:[1]

```
./server -POAImplName Bank
```

Now we have persistent objects, but still have to start up the server by hand. It would be much more convenient if the server was started automatically. This can be achieved using the MICO Daemon (`micod`) (see section 5.4).

[1] If you omit this option, you will receive an "Invalid Policy" exception when trying to create a persistent POA.

For POA-based persistent servers, the implementation repository entry must use the "`poa`" activation mode, for example

```
imr create Bank poa ./server IDL:Bank:1.0
```

The second parameter to imr, `Bank`, is the same implementation name as above; it must be unique within the implementation repository. If a persistent POA is in contact with the MICO Daemon, object references to a persistent object, when exported from the server process, will not point directly to the server but to the MICO Daemon. Whenever a request is received by `micod`, it checks if your server is running. If it is, the request is simply forwarded, else a new server is started.

Usually, the first instance of your server must be started manually for bootstrapping, so that you have a chance to export object references to your persistent objects. An alternative is to use the MICO Binder: the `IDL:Bank:1.0` in the command line above tells `micod` that `bind()` requests for this repository id can be forwarded to this server – after starting it.

With POA-based persistent objects, you can also take advantage of the "iioploc:" addressing scheme that is introduced by the Interoperable Naming Service. Instead of using a stringified object reference, you can use a much simpler, URL-like scheme. The format for an iioploc address is

```
iioploc://<host>:<port>/<object-key>
```

`host` and `port` are as given with the `-ORBIIOPAddr` command-line option, and the object key is composed of the implementation name, the POA name and the Object Id, separated by slashes. So, if you start a server using

```
./server -ORBIIOPAddr inet:thishost:1234 -POAImplName MyService
```

create a persistent POA with the name "MyPOA", and then activate an object using the "MyObject" Object Id, you could refer to that object using the IOR

```
iioploc://thishost:1234/MyService/MyPOA/MyObject
```

These "iioploc" addresses are understood and translated by the `string_to_object()` method and can therefore be used wherever a stringified object reference can be used.

For added convenience, if the implementation name, the POA name and the Object Id are the same, they are collapsed into a single string. An example for this is the NameService implementation, which uses the "NameService" implementation name. The root naming context is then activated in the "NameService" POA using the "NameService" ObjectId. Consequently, the NameService can be addressed using

```
iioploc://<host>:<port>/NameService
```

Please see the Interoperable Naming Service specification for more details.

B.1.4 Reference Counting

With the POA, implementations do not inherit from `CORBA::Object`. Consequently, memory management for servants is the user's responsibility. Eventually, a servant must be deleted with C++'s `delete` operator, and a user must know when a servant is safe to be deleted – deleting a servant that is still known to a POA leads to undesired results.

CORBA 2.3 addresses this problem and introduces reference counting for servants. However, to maintain compatibility, this feature is optional and must be explicitly activated by the user. This is done by adding `POA_PortableServer::RefCountServantBase` as a base class of your implementation:

```
class HelloWorld_impl :
  virtual public POA_HelloWorld
  virtual public PortableServer::RefCountServantBase
{
  ...
}
```

This activates two new operations for your implementation, `_add_ref()` and `_remove_ref()`. A newly constructed servant has a reference count of 1, and it is deleted automatically once its reference count drops to zero. This way, you can, for example, forget about your servant just after it has been created and activated:

```
  HelloWorld_impl * hw = new HelloWorld_impl;
  HelloWorld_var ref = hw->_this(); // implicit activation
  hw->_remove_ref ();
```

During activation, the POA has increased the reference count for the servant, so you can remove your reference immediately afterwards. The servant will be deleted automatically once the object is deactivated or the POA is destroyed. Note, however, that once you introduce reference counting, you must keep track of the references yourself: All POA operations that return a servant (i.e. `id_to_servant()` will increase the servants' reference count. The `PortableServer::ServantBase_var` class is provided for automated reference counting, acting the same as `CORBA::Object_var` does for Objects.

Appendix C

POA IDL

```
// -*- c++ -*-

#include <mico/policy.idl>
#include <mico/current.idl>

#pragma prefix "omg.org"

module PortableServer
{
  // forward reference

  interface POA;

  native Servant;
  typedef sequence<octet> ObjectId;

  exception ForwardRequest {
    Object forward_reference;
  };

  /*
   * ***************************************************
   *
   * Policy Interfaces
   *
   * ***************************************************
   */

  const CORBA::PolicyType THREAD_POLICY_ID = 16;
  const CORBA::PolicyType LIFESPAN_POLICY_ID = 17;
  const CORBA::PolicyType ID_UNIQUENESS_POLICY_ID = 18;
  const CORBA::PolicyType ID_ASSIGNMENT_POLICY_ID = 19;
  const CORBA::PolicyType IMPLICIT_ACTIVATION_POLICY_ID = 20;
  const CORBA::PolicyType SERVANT_RETENTION_POLICY_ID = 21;
  const CORBA::PolicyType REQUEST_PROCESSING_POLICY_ID = 22;

  enum ThreadPolicyValue {
    ORB_CTRL_MODEL,
    SINGLE_THREAD_MODEL
  };

  interface ThreadPolicy : CORBA::Policy {
    readonly attribute ThreadPolicyValue value;
  };

  enum LifespanPolicyValue {
    TRANSIENT,
    PERSISTENT
  };

  interface LifespanPolicy : CORBA::Policy {
    readonly attribute LifespanPolicyValue value;
  };

  enum IdUniquenessPolicyValue {
    UNIQUE_ID,
    MULTIPLE_ID
  };

  interface IdUniquenessPolicy : CORBA::Policy {
    readonly attribute IdUniquenessPolicyValue value;
  };

  enum IdAssignmentPolicyValue {
    USER_ID,
    SYSTEM_ID
  };

interface IdAssignmentPolicy : CORBA::Policy {
  readonly attribute IdAssignmentPolicyValue value;
};

enum ImplicitActivationPolicyValue {
  IMPLICIT_ACTIVATION,
  NO_IMPLICIT_ACTIVATION
};

interface ImplicitActivationPolicy : CORBA::Policy {
  readonly attribute ImplicitActivationPolicyValue value;
};

enum ServantRetentionPolicyValue {
  RETAIN,
  NON_RETAIN
};

interface ServantRetentionPolicy : CORBA::Policy {
  readonly attribute ServantRetentionPolicyValue value;
};

enum RequestProcessingPolicyValue {
  USE_ACTIVE_OBJECT_MAP_ONLY,
  USE_DEFAULT_SERVANT,
  USE_SERVANT_MANAGER
};

interface RequestProcessingPolicy : CORBA::Policy {
  readonly attribute RequestProcessingPolicyValue value;
};

/*
 * ***************************************************
 *
 * POAManager interface
 *
 * ***************************************************
 */

interface POAManager {
  exception AdapterInactive {};

  enum State { HOLDING, ACTIVE, DISCARDING, INACTIVE };

  void activate ()
    raises (AdapterInactive);

  void hold_requests (in boolean wait_for_completion)
    raises (AdapterInactive);

  void discard_requests (in boolean wait_for_completion)
    raises (AdapterInactive);

  void deactivate (in boolean etherealize_objects,
                   in boolean wait_for_completion)
    raises (AdapterInactive);

  State get_state ();

  // begin-mico-extension
  void add_managed_poa (in POA managed);
  void del_managed_poa (in POA managed);
  // end-mico-extension
};
```

```
/*
 * ***************************************************
 *
 * AdapterActivator interface
 *
 * ***************************************************
 */

interface AdapterActivator {
  boolean unknown_adapter (in POA parent, in string name);
};

/*
 * ***************************************************
 *
 * ServantManager interface
 *
 * ***************************************************
 */

interface ServantManager {};

interface ServantActivator : ServantManager {
  Servant incarnate (in ObjectId oid,
                     in POA adapter)
    raises (ForwardRequest);

  void etherealize (in ObjectId oid,
                    in POA      adapter,
                    in Servant  serv,
                    in boolean  cleanup_in_progress,
                    in boolean  remaining_activations);
};

interface ServantLocator : ServantManager {
  native Cookie;
  // typedef string Identifier;

  Servant preinvoke (in ObjectId oid,
                     in POA adapter,
                     in string operation,
                     out Cookie the_cookie)
    raises (ForwardRequest);

  void postinvoke (in ObjectId oid,
                   in POA adapter,
                   in string operation,
                   in Cookie the_cookie,
                   in Servant the_servant);
};

/*
 * ***************************************************
 *
 * POA interface
 *
 * ***************************************************
 */

interface POA {
  exception AdapterAlreadyExists {};
  exception AdapterInactive {};
  exception AdapterNonExistent {};
  exception InvalidPolicy { unsigned short index; };
  exception NoServant {};
  exception ObjectAlreadyActive {};
  exception ObjectNotActive {};
  exception ServantAlreadyActive {};
  exception ServantNotActive {};
  exception WrongAdapter {};
  exception WrongPolicy {};

  /*
   * ***************************************************
   *
   * POA creation and destruction
   *
   * ***************************************************
   */

  POA create_POA (in string adapter_name,
                  in POAManager a_POAManager,
                  in CORBA::PolicyList policies)
    raises (AdapterAlreadyExists, InvalidPolicy);

  POA find_POA (in string adapter_name, in boolean activate_it)
    raises (AdapterNonExistent);

  void destroy (in boolean etherealize_objects,
                in boolean wait_for_completion);

  /*
   * ***************************************************
   *
   * Factories for Policy objects
   *
   * ***************************************************
   */

  ThreadPolicy create_thread_policy
    (in ThreadPolicyValue value);
  LifespanPolicy create_lifespan_policy
    (in LifespanPolicyValue value);
  IdUniquenessPolicy create_id_uniqueness_policy
    (in IdUniquenessPolicyValue value);
  IdAssignmentPolicy create_id_assignment_policy
    (in IdAssignmentPolicyValue value);
  ImplicitActivationPolicy create_implicit_activation_policy
    (in ImplicitActivationPolicyValue value);
  ServantRetentionPolicy create_servant_retention_policy
    (in ServantRetentionPolicyValue value);
  RequestProcessingPolicy create_request_processing_policy
    (in RequestProcessingPolicyValue value);


  /*
   * ***************************************************
   *
   * POA attributes
   *
   * ***************************************************
   */

  readonly attribute string the_name;
  readonly attribute POA the_parent;
  readonly attribute POAManager the_POAManager;
  attribute AdapterActivator the_activator;


  /*
   * ***************************************************
   *
   * ServantManager registration
   *
   * ***************************************************
   */

  ServantManager get_servant_manager ()
    raises (WrongPolicy);

  void set_servant_manager (in ServantManager imgr)
    raises (WrongPolicy);


  /*
   * ***************************************************
   *
   * operations for the USE_DEFAULT_SERVANT policy
   *
   * ***************************************************
   */

  Servant get_servant ()
    raises (NoServant, WrongPolicy);

  void set_servant (in Servant p_servant)
    raises (WrongPolicy);


  /*
   * ***************************************************
   *
   * object activation and deactivation
   *
   * ***************************************************
   */

  ObjectId activate_object (in Servant p_servant)
    raises (ServantAlreadyActive, WrongPolicy);

  void activate_object_with_id (in ObjectId id,
                                in Servant p_servant)
    raises (ServantAlreadyActive, ObjectAlreadyActive,
            WrongPolicy);

  void deactivate_object (in ObjectId oid)
    raises (ObjectNotActive, WrongPolicy);
```

```
    /*
     * *************************************************
     *
     * reference creation operations
     *
     * *************************************************
     */

    // typedef string RepositoryId;

    Object create_reference (in string intf)
      raises (WrongPolicy);

    Object create_reference_with_id (in ObjectId oid,
                                     in string intf)
      raises (WrongPolicy);

    /*
     * *************************************************
     *
     * Identity mapping operations
     *
     * *************************************************
     */

    ObjectId servant_to_id (in Servant p_servant)
      raises (ServantNotActive, WrongPolicy);

    Object servant_to_reference (in Servant p_servant)
      raises (ServantNotActive, WrongPolicy);

    Servant reference_to_servant (in Object reference)
      raises (ObjectNotActive, WrongAdapter, WrongPolicy);

    ObjectId reference_to_id (in Object reference)
      raises (WrongAdapter, WrongPolicy);

    Servant id_to_servant (in ObjectId oid)
      raises (ObjectNotActive, WrongPolicy);

    Object id_to_reference (in ObjectId oid)
      raises (ObjectNotActive, WrongPolicy);

    // begin-mico-extension
    Object activate_for_this (in Servant serv);
    void poa_manager_callback (in POAManager::State newstate,
                               in boolean etherealize_objects,
                               in boolean wait_for_completion);
    Servant preinvoke (in Object for_obj);
    void postinvoke ();
    // end-mico-extension
  };

  /*
   * *************************************************
   *
   * Current interface
   *
   * *************************************************
   */

  interface Current : CORBA::Current {
    exception NoContext {};

    POA get_POA ()
      raises (NoContext);

    ObjectId get_object_id ()
      raises (NoContext);

    // begin-mico-extension
    Object make_ref ();
    boolean iscurrent ();
    Servant get_serv ();
    // end-mico-extension
  };
};
```

Bibliography

[1] Werner Almesberger, *Linux ATM API. Draft, version 0.4.* Lausanne, 1996.
`http://lrcwww.epfl.ch/linux-atm/`

[2] Joe Armstrong, Robert Virding, Claes Wikström and Mike Williams, *Concurrent Programming in Erlang.* Prentice Hall, 1996.

[3] The ATM Forum, *ATM User-Network Interface Specification, Version 3.0.* Prentice Hall, 1993.

[4] Christian Becker and Kurt Geihs, *QoS as a Competitive Advantage for Distributed Object Systems.* Proceedings of the Second International Enterprise Distributed Object Computing Workshop, 1998.

[5] Tim Berners-Lee, *Universal Resource Identifiers in WWW.* RFC 1630, June 1994.

[6] A. D. Birrell and B. J. Nelson, *Implementing Remote Procedure Calls.* ACM Transactions on Computer Systems. Vol. 2, No. 1, February 1984.

[7] Grady Booch, *Object-oriented Analysis and Design.* Benjamin Cummings Publishing Inc., 1994.

[8] B. Callaghan, B. Pawlowski and P. Staubach, *NFS Version 3 Protocol Specification.* RFC 1813, June 1995.

[9] L. Cardelli and P. Wegener, *On Understanding Types, Data Abstraction, and Polymorphism.* ACM Computing Surveys, Vol. 17, No. 4, December 1985.

[10] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[11] Al Geist et al, *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

[12] GNOME Project Home Page, June 1999.
`http://www.gnome.org/`

[13] Seif Haridi, Peter van Roy, Per Brand and Christian Schulte, *Programming Languages for Distributed Applications.* New Generation Computing, Vol. 16, No. 3, Tokyo, 1998.

[14] Elliotte R. Harold, *Java Network Programming.* O'Reilly & Associates, Inc., 1997

[15] Michi Henning, *Binding, Migration, and Scalability in CORBA*. Communications of the ACM, Vol. 41, No 10, October 1998.

[16] Michi Henning and Steve Vinoski, *Advanced CORBA Programming with C++*. Addison-Wesley, 1999.

[17] A. Josey, *Go Solo 2: The Authorized Guide to Version 2 of the Single UNIX Specification*. Prentice Hall, 1997.

[18] Brian Kantor and Phil Lapsley, *Network News Transfer Protocol*. RFC 977. February 1986.

[19] KDE Project Home Page, June 1999.
`http://www.kde.org/`

[20] S. Maffeis, *Adding Group Communication and Fault-Tolerance to CORBA*. Proceedings of the USENIX Conference on Object-Oriented Technologies, June 1995.

[21] MICO Web site, `http://www.mico.org/`

[22] Steffen Nowacki, *An Alternative IDL-C++-Mapping*. Proceedings of the European Conference on Object-Oriented Programming. 1997.

[23] Object Management Group, *The History of CORBA*. October 1998.
`http://www.omg.org/corba/corbahistory.html`

[24] Object Management Group, *Interoperable Naming Service*. Preliminary Specification, October 1998.
`ftp://ftp.omg.org/pub/docs/orbos/98-10-11.ps`

[25] Object Management Group, *ORB Portability Enhancement RFP*. June 1995.
`ftp://ftp.omg.org/pub/docs/1995/95-06-26.ps`

[26] Object Management Group, *ORB Portability Joint Submission*. May 1997.
`ftp://ftp.omg.org/pub/docs/orbos/1997/97-05-15.ps`

[27] Object Management Group, *CORBA 2.0*. July 1996.
`ftp://ftp.omg.org/pub/docs/ptc/96-03-04.ps`

[28] Object Management Group, *CORBA 2.2*. February 1998.
`ftp://ftp.omg.org/pub/docs/formal/98-07-01.ps`

[29] Object Management Group, *CORBA 2.2 - Mapping of OMG IDL to Java*. February 1998.
`ftp://ftp.omg.org/pub/docs/formal/98-02-29.ps`

[30] Object Management Group, *CORBA 2.3 Core Final Revision*. December 1998.
`ftp://ftp.omg.org/pub/docs/ptc/98-12-04.pdf`

[31] Object Management Group, *CORBA 2.3 - Mapping of OMG IDL to C++*. Preliminary Specification, November 1998.
`ftp://ftp.omg.org/pub/docs/ptc/98-09-03.ps`

[32] Object Management Group, *CORBA 2.3 - Mapping of OMG IDL to Java*. Preliminary Specification, May 1999.
`ftp://ftp.omg.org/pub/orbrev/drafts/idljava_2_4v2.3.pdf`

[33] Object Management Group, *Issues for Object Request Broker 2.3a Revision Task Force*. May 1999.
`http://www.omg.org/issues/orb_revision.html`

[34] Object Management Group, *Issues for C++ Revision Task Force*. May 1999.
`http://www.omg.org/issues/cxx_revision.html`

[35] Object Management Group, *CORBA Messaging Joint Revised Submission*. May 1998.
`ftp://ftp.omg.org/pub/docs/orbos/98-05-05.ps`

[36] Object Management Group, *CORBA Security Service*. December 1998.
`ftp://ftp.omg.org/pub/docs/formal/98-12-17.ps`

[37] Object Management Group, *Real-Time CORBA Joint Revised Submission*. May 1999.
`ftp://ftp.omg.org/pub/docs/ptc/99-05-03.ps`

[38] Object Management Group, *Persistent Object State Service 2.0 RFP*. June 1997.
`ftp://ftp.omg.org/pub/docs/orbos/1997/97-06-07.ps`

[39] Object Management Group, *Persistent Object State Service*. December 1997.
`ftp://ftp.omg.org/pub/docs/formal/97-12-12.ps`

[40] Object Management Group, *Policies and Procedures of the OMG Technical Process*. Version 1.3, June 1997
`ftp://ftp.omg.org/pub/docs/pp/97-06-01.ps`

[41] Object Oriented Concepts, *ORBacus*. June 1999.
`http://www.ooc.com/ob/`

[42] Bernd Oestereich, *Objektorientierte Softwareentwicklung mit der Unified Modeling Language*. Oldenbourg, 1997.

[43] RedHat Labs, *ORBit*. June 1999.
`http://www.labs.redhat.com/orbit/`

[44] Robert Orfali, *The Essential Distributed Objects Survival Guide*. John Wiley & Sons, Inc, 1996.

[45] Open Group, MICO 2.2.7 *Open Brand Certificate*. May 1999.
`http://www.opengroup.org/registration/certificates/thinkone541.pdf`

[46] Bruce Perens, *The Open Source Definition*. Version 1.4, June 1997.
`http://www.opensource.org/osd.html`

[47] Jonathan B. Postel, *User Datagram Protocol*. RFC 768. August 1980.

[48] Jonathan B. Postel, *Internet Protocol*. RFC 791. September 1981.

[49] Jonathan B. Postel, *Transmission Control Protocol*. RFC 793. September 1981.

[50] Arno Puder and Kay Römer, MICO - MICO *is CORBA*. Morgan Kaufmann Publishers, 1998.

[51] Irfan Pyarali, Carlos O'Ryan, Douglas C. Schmidt, Nanbor Wang, Vishal Kachroo and Aniruddha Gokhale, *Design Considerations and Performance Optimizations for Real-time ORBs.* 5^{th} USENIX Conference on OO Technologies and Systems, San Diego, CA, May 1999.

[52] Eric S. Raymond, *The Intercal Resource Page*. May 1999.
`http://www.tuxedo.org/~esr/intercal/`

[53] Joyce K. Reynolds and Jon Postel, *RFC 1700: Assigned Numbers*. October 1994.

[54] Dale Rogerson, *Inside DCOM*. Microsoft Press, 1997.

[55] Kay Römer, MICO - MICO *is CORBA. Eine erweiterbare CORBA-Implementierung für Forschung und Ausbildung.* Diplomarbeit am Fachbereich Informatik. Frankfurt am Main, Germany, February 1998.

[56] Douglas C. Schmidt and Steve Vinoski, *Distributed Callbacks and Decoupled Communication in CORBA*. SIGS, Vol 8, No 9, October 1996.

[57] Douglas C. Schmidt and Steve Vinoski, *Object Interconnections: Object Adapters: Concepts and Terminology.* SIGS, Vol 9, No 11, November/December 1997.

[58] Douglas C. Schmidt and Steve Vinoski, *Object Interconnections: Using the Portable Object Adapter for Transient and Persistent CORBA Objects.* SIGS, Vol. 10, No 4, April 1998.

[59] Douglas C. Schmidt and Steve Vinoski, *Object Interconnections: C++ Servant Classes for the POA*. SIGS, Vol. 10, No 6, June 1998.

[60] Douglas C. Schmidt and Steve Vinoski, *Object Interconnections: Programming Asynchronous Method Invocations with CORBA Messaging.* C++ Report, SIGS, Vol. 11, No 2, February 1999.

[61] Douglas C. Schmidt et al, *TAO - The ACE ORB.*
`http://www.cs.wustl.edu/~schmidt/TAO.html`

[62] Krishnan Seetharaman, *The CORBA Connection.* Communications of the ACM, Vol. 41, No 10, October 1998.

[63] Jon Siegel, *CORBA Fundamentals and Programming.* John Wiley & Sons, Inc., 1996.

[64] Jon Siegel, *OMG Overview: CORBA and the OMA in Enterprise Computing.* Communications of the ACM, Vol. 41, No 10, October 1998.

[65] R. Srinivasan, *RPC: Remote Procedure Call Protocol Specification Version 2.* RFC 1831. August 1995.

[66] R. Srinivasan, *XDR: External Data Representation Standard.* RFC 1832. August 1995.

[67] Richard Stevens, *Unix Network Programming, Second Edition, Volume 1. Networking APIs: Sockets and XTI.* Prentice Hall, 1998.

[68] Richard Stevens, *Unix Network Programming, Second Edition, Volume 2. Interprocess Communication.* Prentice Hall, 1999.

[69] Sun Microsystems, Inc., *Java-Based Distributed Computing. RMI and IIOP in Java.* Press Release, June 1997.
`http://java.sun.com/pr/1997/june/statement970626-01.html`

[70] Andrew S. Tanenbaum, *Computer Networks.* Prentice Hall, 1989.

[71] Steve Vinoski, *New Features for CORBA 3.0.* Communications of the ACM, Vol. 41, No 10, October 1998.

[72] Torben Weis, *Dynamische Konfiguration von Anwendungen.* Diplomarbeit am Fachbereich Informatik. Frankfurt am Main, Germany, February 1999.

[73] J. E. White, *A High-Level Framework for Network-Based Resource Sharing.* RFC 707, December 1975.