

OBJECT ORIENTED REMOTING

**Foundations of Realtime,
Internet and Enterprise
Distribution Infrastructures**



Markus Völter
Michael Kircher
Uwe Zdun



WILEY SERIES IN
SOFTWARE DESIGN PATTERNS

2nd REVIEW DRAFT

25 June 2004

(c) 2003 Kircher, Völter, Zdun

FOR REVIEW ONLY
PLEASE DO NOT DISTRIBUTE
THE PDF ANY FURTHER

Table of Contents

1	Series Foreword	5
2	Foreword	7
3	Preface	11
3.1	How to Read this Book	11
3.2	Goals of the Book	13
3.3	About the Authors	14
3.4	Acknowledgements	16
3.5	Patterns and Pattern Languages	17
3.6	Our Pattern Form	22
3.7	Key to the Illustrations	25
4	Introduction To Distributed Systems	29
4.1	Distributed Systems: Reasons and Challenges	29
4.2	Communication Middleware	35
4.3	Remoting Styles	37
5	Pattern Language Overview	49
5.1	Overview of the Pattern Chapters	57
6	Basic Remoting Patterns	65
6.1	Interactions among the Patterns	97
7	Identification Patterns	103
7.1	Interactions among the Patterns	115
8	Lifecycle Management Patterns	117
8.1	Basic Lifecycle Patterns	118
8.2	General Resource Management Patterns	129
8.3	Interactions among the Patterns	142

9	Extension Patterns	157
9.1	Interactions among the Patterns	171
10	Extended Infrastructure Patterns	175
10.1	Interactions among the Patterns	193
11	Invocation Asynchrony Patterns	197
11.1	Interactions among the Patterns	216
11.2	Related Patterns: Invocation Asynchrony	229
12	Related Concepts, Technologies, and Patterns.....	231
12.1	Related Patterns	232
12.2	Distribution Infrastructures	236
12.3	Quality Attributes.....	246
12.4	Aspect-Orientation and Remoting	251
13	Technology Projections	253
14NET	
Remoting Technology Projection		255
14.1	Brief History of .NET Remoting	255
14.2	.NET Concepts - A Brief Introduction.....	256
14.3	Pattern Map	258
14.4	An Simple .NET Remoting Example.....	259
14.5	Remoting Boundaries.....	264
14.6	Basic Internals.....	266
14.7	Error Handling	267
14.8	Server-Activated Instances	268
14.9	Client-Dependent Instances and Leasing.....	276
14.10	More Advanced Lifecycle Management.....	283
14.11	Internals of .NET Remoting.....	285

14.12	Extensibility of .NET Remoting.....	289
14.13	Asynchronous Communication	296
14.14	Outlook to the Next Generation	302
15	Web Services Technology Projection.....	305
15.1	Brief History of Web Services	305
15.2	Pattern Map	310
15.3	SOAP Messages	310
15.4	Message Processing.....	320
15.5	Protocol Integration.....	328
15.6	Marshalling using SOAP XML Encoding.....	330
15.7	Lifecycle Management	333
15.8	Client-Side Asynchrony	335
15.9	Web Services and QoS	340
15.10	Web Service Security	342
15.11	Lookup of Web Services: UDDI.....	343
15.12	Other Web Service Frameworks	345
15.13	Consequences of the Pattern Variants Used in Web Services	354
16	CORBA Technology Projection	357
16.1	Brief History of CORBA.....	357
16.2	Pattern Map	359
16.3	Initial Example	360
16.4	CORBA Basics.....	362
16.5	Messaging.....	382
16.6	Real-Time CORBA	387
17	Appendix	395
17.1	Appendix A: Building Blocks for Component Infrastructures	395
17.2	Appendix B: Extending AOP Frameworks for Remoting...	399

18 **References..... 407**

2 Series Foreword

I

|

3

Foreword

Many of today's enterprise computing systems are powered by distributed object middleware. Such systems, which are common in industries such as telecommunications, finance, manufacturing, and government, often support applications that are critical to particular business operations. Because of this, distributed object middleware is often held to stringent performance, reliability, and availability requirements. Fortunately, modern approaches have no problem meeting or exceeding these requirements. Today, successful distributed object systems are essentially taken for granted.

There was a time, however, when making such claims about the possibilities of distributed objects would have met with disbelief and derision. In their early days, distributed object approaches were often viewed as mere academic fluff with no practical utility. Fortunately, the creators of visionary distributed object systems such as Eden, Argus, Emerald, COMANDOS, and others were undeterred by such opinion. Despite the fact that the experimental distributed object systems of the 1980s were generally impractical — too big, too slow, or based on features available only from particular specialized platforms or programming languages — the exploration and experimentation required to put them together collectively paved the way for the practical distributed objects systems that followed.

The 1990s saw the rise of several commercially successful and popular distributed object approaches, notably the Common Object Request Broker Architecture (CORBA) promoted by the Object Management Group (OMG) and Microsoft's Common Object Model (COM). CORBA was specifically designed to address the inherent heterogeneity of business computing networks, where mixtures of machine types, operating systems, programming languages, and application styles are the norm and must coexist and cooperate. COM, on the other hand, was built specifically to support component-oriented applications running on the Microsoft Windows operating system. Today, COM has been

largely subsumed by its successor, .NET, while CORBA remains in wide use as a well-proven architecture for building and deploying significant enterprise-scale heterogeneous systems, as well as real-time and embedded systems.

As this book so lucidly explains, despite the fact that CORBA and COM were designed for fundamentally different purposes, they share a number of similarities. These similarities range from basic notions, including remote objects, client and server applications, proxies, marshallers, synchronous and asynchronous communications, and interface descriptions, to more advanced areas, including object identification and lookup, infrastructure extension, and lifecycle management. Not surprisingly, though, these similarities do not end at CORBA and COM. They can also be found in newer technologies and approaches, including .NET, the Java 2 Enterprise Edition (J2EE), and even in Web Services (which, strictly speaking, is not a pure distributed objects technology, but still has inherited many of its characteristics).

Such similarities are of course better known as patterns. Patterns are generally not so much created as they are discovered, much like a miner finds a diamond or a gold nugget buried in the earth. Successful patterns result from the study of successful systems, and the remoting patterns presented here are no exception. Our authors, Markus, Michael, and Uwe, who are each well versed in both the theory and practice of distributed objects, have worked extensively with each of the technologies I've mentioned. Applying their pattern mining talents and efforts, they have captured for the rest of us the critical essence of a number of successful solutions and approaches found in a number of similar distributed objects technologies.

Given my own long history with CORBA, I am not surprised to find that several of the patterns that Markus, Michael, and Uwe document here are among my personal favorites. For example, topping my list is the Invocation Interceptor pattern, which I have found to be critical for creating distributed objects middleware that provides extensibility and modularity without sacrificing performance. Another favorite of mine is the Leasing pattern, which can be extremely effective for managing object lifecycles.

This book does not simply describe a few remoting patterns, however. While many patterns books comprise only a loose collection of

patterns, this book also provides a series of technology projections that tie the patterns directly back to the technologies that employ them. These projections clearly show how the patterns are used within .NET, CORBA, and Web Services, effectively recreating these architectures from the patterns mined from within them. With technology projections like these, it has never been easier to see the relationships and roles of different patterns with respect to each other within an entire architecture. These technology projections clearly link the patterns, which are already invaluable by themselves, together into a comprehensive, harmonious, and rich distributed objects pattern language. In doing so, they conspicuously reveal the similarities among these different distributed object technologies. Indeed, we might have avoided the overzealous and tiresome “CORBA vs. COM” arguments of the mid-1990s had we had these technology projections and patterns at the time.

Distributed objects technologies continue to evolve and grow. These patterns have essentially provided the building blocks for the experimental systems of the 1980s, for the continued commercial success and wide deployment of distributed objects that began in the 1990s, and for today’s Web Services integration approaches. Due to the never-ending march of technology, you can be sure that before too long, new technologies will come along to displace Web Services. You can also be sure that the remoting patterns that Markus, Michael, and Uwe have so expertly provide for us here will be at the heart of those new technologies as well.

Steve Vinoski
Chief Engineer
IONA Technologies
March 2004

4 Preface

Today distributed object middleware belongs to the basic elements in the toolbox of software developers, designers, and architects developing distributed systems. Popular examples of such distributed object middleware systems are CORBA, Web Services, DCOM, Java RMI, and .NET Remoting. There are many other books that explain how a particular distributed object middleware works. If you just want to use one particular distributed object middleware, many of these books are highly valuable. However, as a professional software developer, designer, or architect working with distributed systems, you will also experience situations in which just understanding how to use one particular middleware is not enough. To do this, you are required to gain a deeper understanding of the inner workings of the middleware in order to customize or extend the middleware to your needs. Or, as a consequence of business requirements, you are forced to migrate your system to a new kind of middleware. Or, you need to integrate two systems which use two different middleware products.

This book is intended to help you in these and similar situations: it explains the inner workings of successful approaches and technologies in the field of distributed object middleware in a practical manner. To achieve this we use a pattern language describing the essential building blocks of distributed object middleware based on a number of compact, Alexandrian-style [AIS+77] patterns. We supplement the pattern language with three technology projections that explain how the patterns are realized in different real world examples of distributed object middleware systems: .NET Remoting, Web Services, and CORBA.

How to Read this Book

This book primarily aims at software developers, designers, and architects having at least a basic understanding of software development and design concepts.

For readers who are new to patterns, we introduce patterns and pattern languages to some extent in this chapter. Readers familiar with patterns might want to skip that section. We also briefly explain the pattern form and the diagrams used in this book at the end of this chapter. Readers might at least want to scan this information and use it as a reference, when reading the later chapters of the book.

In the pattern chapters and the later technology projections we assume some knowledge of distributed system development. We give a short introduction to the basic terminology and concepts used in this book in the next chapter. Readers who are not familiar with the field or who want to refresh their knowledge should read this chapter. Readers who are experts in the field might want to skip the *Introduction to Distributed Systems* chapter. If you are completely new to this field, you might want to read a more detailed introduction like Tanenbaum and van Steen's *Distributed Systems: Principles and Paradigms* [TS02].

For all readers, we recommend reading the pattern language chapters as a whole. This should give you a fairly good picture of how distributed object middleware systems work. When working with the pattern language, you usually can directly go to particular patterns of interest and use the pattern relationships described in the pattern descriptions to find related patterns.

Details on the interactions among the patterns can be found at the end of each pattern chapter depicted in a number of sequence diagrams. We have not included these interactions in the individual pattern descriptions for two reasons: Firstly, it would make the pattern chapters less readable as a whole. Secondly, the patterns in each chapter have strong interactions - so it makes sense to illustrate them with integrated examples instead of scattering the examples across the individual pattern descriptions.

Especially, if you want to implement your own distributed object middleware system or extend an existing one, we recommend to look closely at the sequence diagram examples. This will give you some more insight into how the pattern language can be implemented. As a next step, you may want to read the technology projections to see a couple of well established real-world examples of how the pattern language got implemented by vendors.

If you want to understand the commonalities and differences between some of the mainstream distributed object middleware systems, you should read the technology projections in any order you like. They are completely independent of each other.

Goals of the Book

Numerous projects use, extend, integrate, customize, and build distributed object middleware. The major goal of the pattern language in this book is to provide knowledge about the general, recurring architecture of successful distributed object middleware as well as more concrete design and implementation strategies. As a reader you might benefit from reading and understanding this pattern language in several ways:

- If you want to *use* a distributed object middleware, you will benefit from better understanding the concepts of your middleware implementation and this, in turn, helps you to make better use of the middleware. If you know how to use one middleware system and need to switch to another, then understanding the patterns of distributed object middleware helps you to see the commonalities in spite of different remoting abstractions, terminologies, implementation language concepts, and so forth.
- Sometimes you need to *extend* the middleware with additional functionality. For instance, consider you are developing a Web Service application. Because Web Services are relatively new, your chosen Web Service framework might not implement certain security or transaction features that you really need for your application. Then you need to implement these features on your own. Here, the patterns help you to find the best hooks for extending the Web Service framework. The patterns show you the alternatives of successful implementations of such extensions. The book also let you find similar solutions in other middleware implementations so that you can get some ideas - to avoid reinventing the wheel. Another typical extension is the introduction of “new” remoting styles, implemented on top of existing middleware. Consider server-side component architectures, such as CORBA Components, COM+, or Enterprise Java Beans (EJB): they use distributed object middleware implementations as a foundation for remote communication [VSW02]. They extend the middleware

with new concepts. Again, as a developer of a component architecture, you have to understand the patterns of the distributed object middleware, for instance, to integrate the lifecycle models of the components and remote objects.

- While distributed object middleware is used to integrate heterogeneous systems, you might encounter situations in which you need to *integrate* the various middleware systems. Just consider the situation where your employer buys another company, which uses a different middleware product from the one used in your company. So you need to integrate the two middleware solutions in order to let the information systems of the two companies play in concert. Here, the patterns help you find integration spots and identify promising solutions.
- In some, more rare cases you need to *customize* a distributed object middleware, or even *build* it from scratch. Consider for instance an embedded system with tight constraints regarding memory consumption, performance, and real-time communication [Aut04]. If no suitable middleware product exists or all available products turn out to be inappropriate and/or have a footprint that is too large, the developers must develop their own solution. As an alternative, you might take a look at existing open-source solutions and try to customize them for your needs. Here the patterns help you to identify critical components of the middleware and assess the effort for customizing them. If customizing an existing middleware does not seem to be feasible, you can use the patterns to build a new distributed object middleware for your application.

The list above are just a few motivating examples. We hope these examples illustrate the broad variety of situations in which you might want to get a deeper understanding of distributed object middleware. And, as these situations are occurring over and over again, we hope these examples illustrate why we - as the authors of this book - think that the time is ready for a book that explains these issues in a way accessible for practitioners.

About the Authors

Markus Völter

Markus works as an independent consultant on software technology and engineering out of Heidenheim, Germany. His primary focus is on software architecture and patterns, middleware and model-driven software development. Markus has consulted and coached in many different domains such as banking, health care, e-business, telematics, astronomy and automotive embedded systems, in projects ranging from 5 to 150 developers.

Markus is also a regular speaker at international conferences on software technology and object orientation. Among others, he has given talks and tutorials at ECOOP, OOPSLA, OOP, OT, JAOO and GPCE. Markus has published patterns at various *PLOP conferences and is constantly writing articles in various magazines on topics he is interested in. He is also co-author of the book *Server Component Patterns* (also Wiley Pattern Series).

When not dealing with software, Markus enjoys cross-country flying in the skies over southern Germany in his sailplane.

Markus can be reached at voelter@acm.org or via www.voelter.de

Michael Kircher

Michael Kircher is currently working as Senior Software Engineer at Siemens AG Corporate Technology in Munich, Germany. His main fields of interest include distributed object computing, software architecture, patterns, agile methodologies, and management of knowledge workers in innovative environments. He has been involved as a consultant and developer in many projects, within various Siemens business areas, building software for distributed systems. Among those were the development of software for UMTS base stations, toll systems, postal automation systems, and operation and maintenance software for industry and telecommunication systems.

In recent years he has published at numerous conferences on topics such as patterns, software architecture for distributed systems, and eXtreme Programming and organized several workshops at conferences such as OOPSLA and EuroPLOP.

In his spare time Michael likes to combine family life with enjoying nature, doing sports or just watching wildlife.

Michael can be reached at michael@kircher-schwanninger.de or via www.kircher-schwanninger.de

Uwe Zdun

Uwe Zdun is currently working as an assistant professor in the department of information system at the Vienna University of Economics and Business Administration. He received his doctoral degree from the University of Essen in 2002, where he has also worked as research assistant from 1999 to 2002 in the software specification group. His research interests include software patterns, scripting, object-orientation, software architecture, and web engineering. Uwe has been involved as a consultant and developer in many software projects. He is author of a number of open-source software systems, including Extended Object Tcl (XOTcl), ActiWeb, Frag, and Leela, as well as many other open-source and industrial software systems.

In recent years he has published in numerous conferences and journals, and co-organized a number of workshops at conferences such as EuroPLoP, CHI, and OOPSLA.

He enjoys the outdoor sports of hiking, biking, and running, and the indoor sports of pool, guitar playing, and sports watching.

Uwe can be reached at zdun@acm.org or via <http://wi.wu-wien.ac.at/~uzdun>

Acknowledgements

A book like the one you have in your hands would not have been possible if not many other people would have supported us. For their support in discussing the contents of the book and for providing their feedback we express our gratitude.

First of all, we want to thank our shepherd, Steve Vinoski, and the pattern series editor, Frank Buschmann. They have read the book several times and provided in-depth comments on technical content as well as on the structure and coherence of the pattern language.

We also want to thank the following people who have provided comments on various versions of the manuscript as well as on extracted papers that have been workshopped at VikingPLoP 2002 and

EuroPLoP 2003: Mikio Aoyama, Steve Berczuk, Valter Cazzalo, Anniruddha Gokhale, Lars Grunske, Klaus Jank, Kevlin Henney, Wolfgang Herzner, Don Hinton, Klaus Marquardt, Jan Mendling, Roy Oberhauser, Joe Oberleitner, Juha Pärsinen, Michael Pont, Alexander Schmid, Kristijan Elof Sorenson (thanks for playing shepherd and proxy), Mark Strembeck, Oliver Vogel, Johnny Willemsen, and Eberhard Wolff.

Finally, we thank those that have been involved with the production of the book: our copy-editor Steve Rickaby and editor Gaynor Redvers-Mutton. It is a pleasure working with such proficient people.

Patterns and Pattern Languages

Over the past couple of years, patterns have become part of the mainstream of software development. They appear in different kinds and forms.

The most popular patterns are those for software design, pioneered by the Gang-of-Four (GoF) book [GHJV95] and continued by many other pattern authors. Design patterns can be applied very broadly, because they focus on everyday design problems. In addition to design patterns, the patterns community has created patterns for software architecture [BMR+96, SSRB00], analysis [Fow96] and even non-IT topics such as organizational or pedagogical patterns [Ped04, FV00]. There are many other kinds of patterns, some are specific for a particular domain.

What is a Pattern?

A pattern, according to the original definition of Alexander¹ [AIS+77], is:

...a three-part rule, which expresses a relation between a certain context, a problem, and a solution.

-
1. In his book, *A Pattern Language - Towns • Buildings • Construction* [AIS+77] Christopher Alexander presents a pattern language of 253 patterns about architecture. He describes patterns that guide the creation of space for people to live, including cities, houses, rooms and so forth. The notion of patterns in software builds on this early work by Alexander.

This is a very general definition of a pattern. It is probably a bad idea to cite Alexander in this way, because he explains this definition extensively. In particular, how can we distinguish a pattern from a simple recipe? Take a look at the following example:

Context	You are driving a car.
Problem	The traffic lights in front of you are red. You must not run over them. What should you do?
Solution	Brake.

Is this a pattern? Certainly not. It is just a simple, plain if-then rule. So, again, what is a pattern? Jim Coplien, on the Hillside web-site [Cop04], proposes another, slightly longer definition which summarizes the discussion in Alexander's book:

Each pattern is a three-part rule, which expresses a relation between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain software configuration which allows these forces to resolve themselves.

Coplien mentions *forces*. Forces are considerations that somehow constrain or influence the solution proposed by the pattern. The set of forces builds up *tension*, usually formulated concisely as a problem statement. A solution for the given problem has to balance the forces somehow, because usually the forces cannot all be resolved optimally – a compromise has to be found.

The pattern, in order to be understandable by the reader, should describe *how* the forces are balanced in the proposed solution, and *why* they have been balanced in the proposed way. In addition, the advantages and disadvantages of such a solution should be explained, to allow the reader to understand the *consequences* of using the pattern.

Patterns are solutions to recurring problems. They therefore need to be quite general so that they can be applied to several concrete problems. However, the solution should be concrete enough to be practically useful, and it should include a description of a specific software configuration. Such a configuration consists of the participants of the pattern,

their responsibilities, and their interactions. The level of detail of this description can vary, but after reading the pattern, the reader should know what he has to do to implement the pattern's solution. As the above discussion highlights, a pattern is not merely a set of UML diagrams or some code fragments.

Patterns are never "new ideas". Patterns are *proven* solutions to recurring problems. So *known uses* for a pattern must always exist. A good rule of thumb is that something that has not at least three known uses is hardly a pattern. In software patterns, this means that systems must exist that are implemented according to the pattern. The usual approach to writing patterns is not to invent them from scratch – instead they are discovered in, and then extracted from, real-life systems. These systems then serve as known uses for the pattern. To find patterns in software systems, the pattern author has to abstract the problem/solution pair from the concrete instances found in the systems at hand. Abstracting the pattern while preserving comprehensibility and practicality is the major challenge of pattern writing.

There is another aspect of what makes a good pattern, the *quality without a name* (QWAN) [AIS+77]. The quality without a name cannot easily be described, the best approximation is *universally-recognizable aesthetic beauty and order*. So, a pattern's solution must somehow appeal to the aesthetic sense of the pattern reader – in our case, to the software developer, designer, or architect. While there is no universal definition of beauty, there certainly are some guidelines as to what is a good solution and what is not. For example, a software system should be efficient, flexible and easily understandable while addressing a complex problem, and so on. The principle of beauty is an important – and often underestimated – guide for judging whether a technological design is good or bad. David Gelernter details this in his book *Machine Beauty* [Gel99].

Classifications of Patterns in this Book

The patterns in this book are *software patterns*. They can further be seen as *architectural* patterns or *design* patterns. It is not easy to draw the line between architecture and design, and often the distinction is depended on the situation you are in and the viewpoint that you take. For a rough

distinction, let's refer to the definition of software architecture from Bass, Clements, and Kazman [BCK98]:

The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally-visible properties of those components and the relationships among them.

What we can see here, is that it heavily depends on the viewpoint of the designer or architect whether a specific pattern is categorized as an architectural pattern or a design pattern. Consider for instance the *Interpreter* pattern [GHJV95]. The description in the Gang-of-Four book describes it as a concrete design guideline. Yet, according to the above software architecture definition, instances of the pattern are often seen as a central elements in the architecture of software systems because an *Interpreter* is a central component of the system that is externally visible.

Most of the patterns in this book can be seen as falling into both categories - design patterns and architectural patterns. From the viewpoint of the designer they provide concrete guidelines for the design of a part of the distributed object middleware. Yet, they also comprise larger, visible structures of the distributed object middleware and focus on the most important components and their relationships. Thus, according to the above definition they are architectural foundations of the distributed object middleware as well.

From Patterns to Pattern Languages

A single pattern describes one solution to a particular, recurring problem. However, 'real big problems' usually cannot be described in one pattern without compromising readability.

The pattern community has therefore come up with several ways to combine patterns to solve a more complex problem or a set of related problems:

- *Compound patterns* are patterns that are assembled from other, smaller patterns. These smaller patterns are usually already well known in the community. Often, for a number of related smaller patterns, known uses exist in which these patterns are always used together in the same software configuration. Such situations are good candidates for being described as a compound pattern. It is

essential that the compound pattern actually solves a distinct problem, and not just a combination of the problems of its contained patterns. A compound pattern also resolves its own set of forces. An example for a compound pattern is *Bureaucracy* by Dirk Riehle [Rie98] that combines *Composite*, *Mediator*, *Chain of Responsibility*, and *Observer* (all from the GoF book).

- A *family of patterns* is a collection of patterns that solves the same general problem. Each pattern either defines the problem more specifically, or resolves the common forces in a different way. For example, different solutions could focus on flexibility, performance or simplicity. Usually each of the patterns also has different consequences. A family therefore describes a problem and *several* proven solutions. The reader has to select the appropriate solution, taking into account how he wants to resolve the common forces in his particular context. A good example is James Noble's Basic Relationship Patterns [Nob97], that describe several alternatives of how logical relationships among objects can be realized in software.
- A *collection*, or *system of patterns* comprises several patterns from the same domain or problem area. Each pattern stands on its own, sometimes referring to other patterns in the collection in its implementation. The patterns form a system because they can be used by a developer working in a specific domain, each pattern resolving a distinct problem the developer might come across during his work. A good example is Pattern Oriented Software Architecture by Buschmann, Meunier, Rohnert, Sommerlad, and Stal (also known as POSA 1 – see [BMR+96]).

The most powerful form of combining patterns is a *pattern language*. There are several characteristics of pattern languages:

- A pattern language has a language-wide goal. The purpose of the language is to guide the user step by step to reach this goal. The patterns in a pattern language are not necessarily useful in isolation. The patterns in this book form a pattern language in the domain of distributed object middleware.
- A pattern language is *generative* in nature. Applying the pattern language generates a whole. This generated whole should be 'beautiful' in the sense of QWAN. The whole is "generated" by

applying the patterns in the pattern language one after another in an incremental process of refinement. The basic idea is that - with each refinement step - the whole gets more and more coherent.

- To generate the whole, the pattern language has to be applied in a specific order. This order is defined by one or more sequences. Depending on the context where the pattern language is applied or which aspects of the whole are actually required, there can be several sequences through a pattern language.
- Because the patterns must be applied in a specific sequence, each pattern must define its place in this sequence. To achieve this, each pattern has a section called its *context*, which mentions earlier patterns that have to be implemented before the current pattern can be implemented successfully. Each pattern can also feature a *resulting context* that describes how to continue. It contains references to the patterns that can be used to help in the implementation of the current pattern or explain how to proceed in the incremental refinement process of the whole.

Pattern languages—in contrast to the other forms of combining patterns discussed above—do not only specify solution to specific problems, but also describe a way to create the whole, the overall goal of the pattern language. Note that one particular pattern can play a role in different pattern languages. For instance, we integrate some patterns from other source in our pattern language, such as the *Broker* pattern [BMR+96] which we use to motivate the overall problem and context of the pattern language in this book.

Our Pattern Form

In this section, we provide a brief introduction to the pattern form used in this book. The form of our patterns is similar to the Alexandrian [AIS+77] format, but omits examples in the pattern description and photographs visualizing the pattern. This book does contain many examples of the patterns; but instead of presenting an example for each individual pattern, the examples appear separate in form of UML sequence diagrams, at the end of each pattern chapter, and as *Technology Projections*.

The individual pattern descriptions are structured as follows: Each pattern starts with a name. The name is an important part of a pattern in a pattern language because it is used throughout the language to refer to the particular pattern. When referencing pattern names from our pattern language, we always write them in SMALLCAPS font. External patterns from the literature are highlighted in *Italics*.

The next section is the context of the pattern in the pattern language. The context is separated by three stars from the main body of the pattern. The main body starts with a problem section written in bold face font. The problem, addressed by the pattern, is then illustrated in more detail in plain font. Especially the system of forces leading to the pattern's solution is discussed here. The section is written in two different styles:

- In structural patterns this section contains problem details and a short discussion of forces.
- In behavioral patterns, found in the *Lifecycle Patterns* and *Client Asynchrony Patterns* chapters, this section contains an example application scenario that motivates the need for the pattern's solution. We feel that such an example illustrates the forces much better for those patterns than an abstract discussion.

Following the problem discussion, the solution of the pattern is again provided using bold face. It is illustrated by a figure. Three stars separate the main body of the pattern from the solution details that follow. The solution details discuss variants of the pattern, related patterns (in the pattern language and from literature), as well as the consequences of applying the pattern.

Examples and known uses are provided in the *Technology Projections* chapter, which projects the patterns on a certain technology. We present technology projections for .Net Remoting, Web Services, and CORBA. Other known uses of the patterns are discussed in the Chapter *Related Concepts, Technologies, and Patterns*. Below, the pattern format

layout is illustrated by an example with some side-headings for explanation.

Pattern Name	Invocation Interceptor
Context	Applications need to transparently integrate add-on services.
Problem	<p style="text-align: center;">* * *</p> <p>In addition to hosting remote objects, the client and server application often have to provide a number of add-on services, such as transactions, logging, or security. The clients and remote objects themselves should be independent of those services.</p>
Problem Detail and Forces - or - Motivating Example	<p>Consider the typical concern of security in distributed object middleware: remote objects need to be protected from unauthorized access. The remote objects themselves should not have to worry about authorization; they should actually be accessible with or without the security concern “authorization” enabled. While the server application needs to enforce security, the client needs to add credentials, such as user name and password, to the request.</p> <p>Therefore:</p> <p>Provide hooks in the invocation path, for example in the INVOKER and REQUESTOR, to plug in INVOCATION INTERCEPTORS. INVOCATION INTERCEPTORS are invoked before and after request and response messages pass the hook. Provide the interceptor with all the necessary information to allow it to provide meaningful add-on services such as operation name, parameters, OBJECT ID, and, if used, the INVOCATION CONTEXT.</p>
Solution	
Scenarion Illustration and Ex-planation	<div><p>The INVOKER receives an invocation and checks whether INVOCATION INTERCEPTORS are registered for that remote object. If so, the interceptor is invoked before and after the actual invocation.</p></div>
Solution Details and Related Patterns	<p style="text-align: center;">* * *</p> <p>INVOCATION INTERCEPTORS allow for the transparent integration of additional orthogonal services, and concerns. Those additional services are independent of the individual clients or remote objects.</p>
Consequences	<p>If more than one INVOCATION INTERCEPTOR is applicable for an invocation, the interceptors are often interdependent. Therefore, in most cases, they are arranged in a <i>Chain of Responsibility</i> [GHJV95]. Each INVOCATION INTERCEPTOR forwards the invocation to the next interceptor in the chain. Alternatively, the components that provide the hooks, such as REQUESTOR and INVOKER, can manage passing information between elements of the chain.</p>
References to Patterns from the Literature	<p>The INVOCATION INTERCEPTOR is a specialization of <i>Interceptor</i> [SSRB00] to distributed object middleware.</p>

Structure of the Pattern Chapters

The pattern chapters have a specific structure that we want to outline quickly. Some of the chapters provide additional sections, but the following three sections are common to each chapter:

- Each chapter starts with an introduction of the general topic of the chapter. Each pattern in the chapter is introduced with one or two sentences. An overview diagram that shows the relationship between the different patterns is provided. This diagram also connects the patterns to patterns in other chapters - the patterns from other chapters are rendered in grey in these figures.
- The second part of each pattern chapter then introduces the patterns themselves. Each pattern - using the structure shown above - covers a couple of pages.
- A third section called *Interactions among the Patterns* shows how the patterns interact. This is done mainly using UML sequence diagrams, illustrating certain example usage scenarios. Additional information on how the patterns collaborate are provided in the text. Note that these diagrams are just scenarios that might omit certain parts or show certain details in a different way than other possible scenarios. They only show a relevant aspect that should be highlighted by the particular example, and abstract from other parts of the solution. Otherwise, the diagrams would become way too complex. The *Technology Projection* chapters provide more detailed examples focussing on the pattern language as a whole.

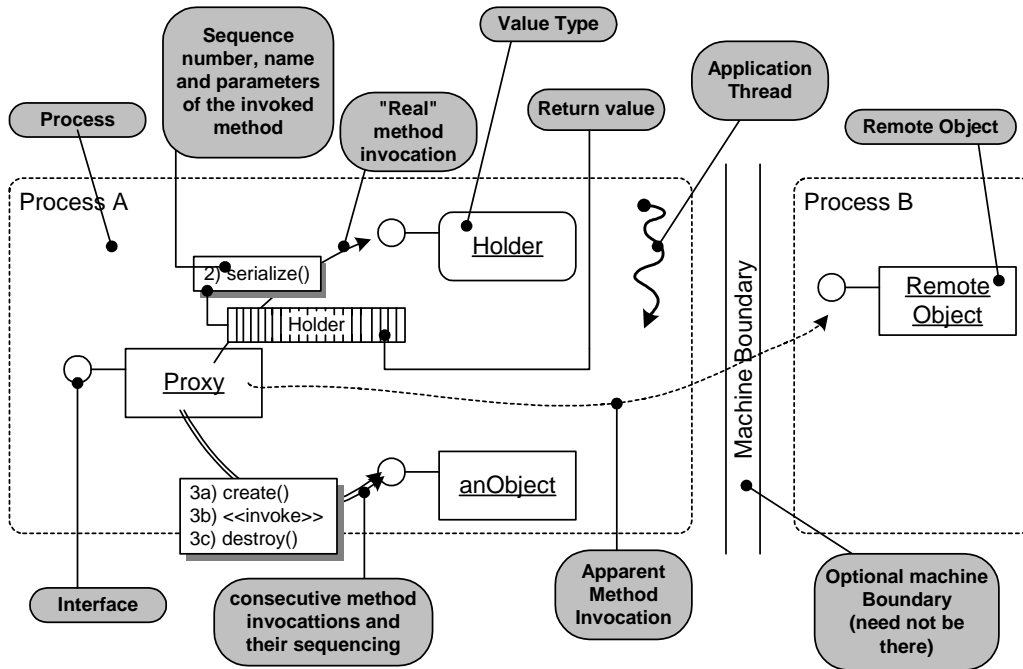
Key to the Illustrations

Most of the illustrations in this book use the UML with some slight variations; we explain the UML subset we use in the following sections. Only the collaboration diagrams used in the pattern descriptions cannot really be called UML.

Collaborations

Each pattern is illustrated with a “collaboration diagram”. In addition to the dynamic collaboration semantics as known from the respective UML diagrams, we also display containment structures in these diagrams. Thus they are not formally UML diagrams. Anyway, we

hope they are useful. The following figure provides an example illustration, annotated with comments to serve as a key.



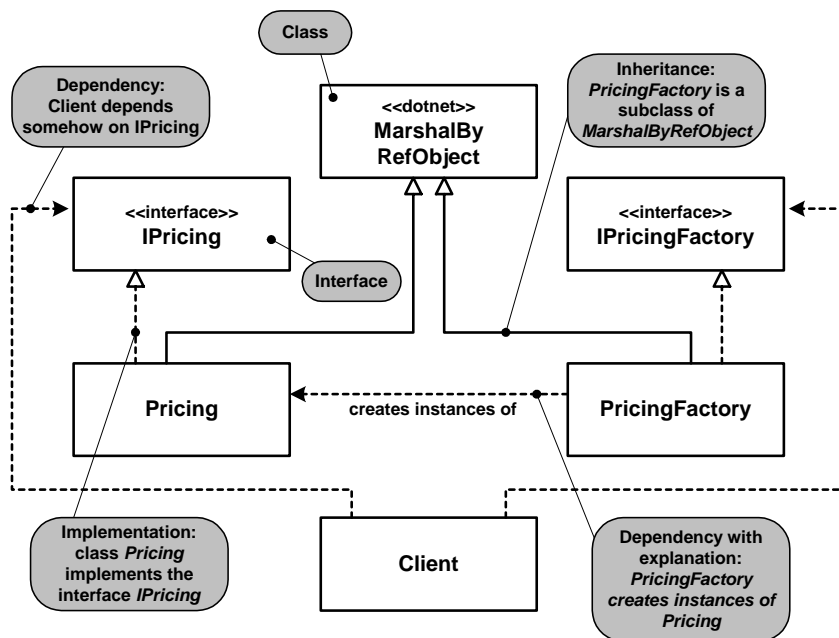
Some more detail on the notation:

- Although remoting is typically used to communicate from one machine to another, that's not necessarily so. It can also be used just to communication between two processes on the same machine. This is the reason, why the *Machine Boundary* is optional.
- An *apparent method invocation* means that the denoted operation is logically executed directly by the invoking entity, but in reality, there might be other intermediate steps involved (such as a proxy).
- The `<<stereotype>>` notation used in some methods denotes the fact that we don't actually invoke an operation, but influence the target conceptually. For example `<<create>>` does not mean that the `create()` operation is called, but instead that the client *creates* the target object.
- The double-arrow does not mean that there are exactly two operations invoked, but instead, that a sequence of operations – the ones

mentioned in the attached box – are executed directly after one another.

Class diagrams

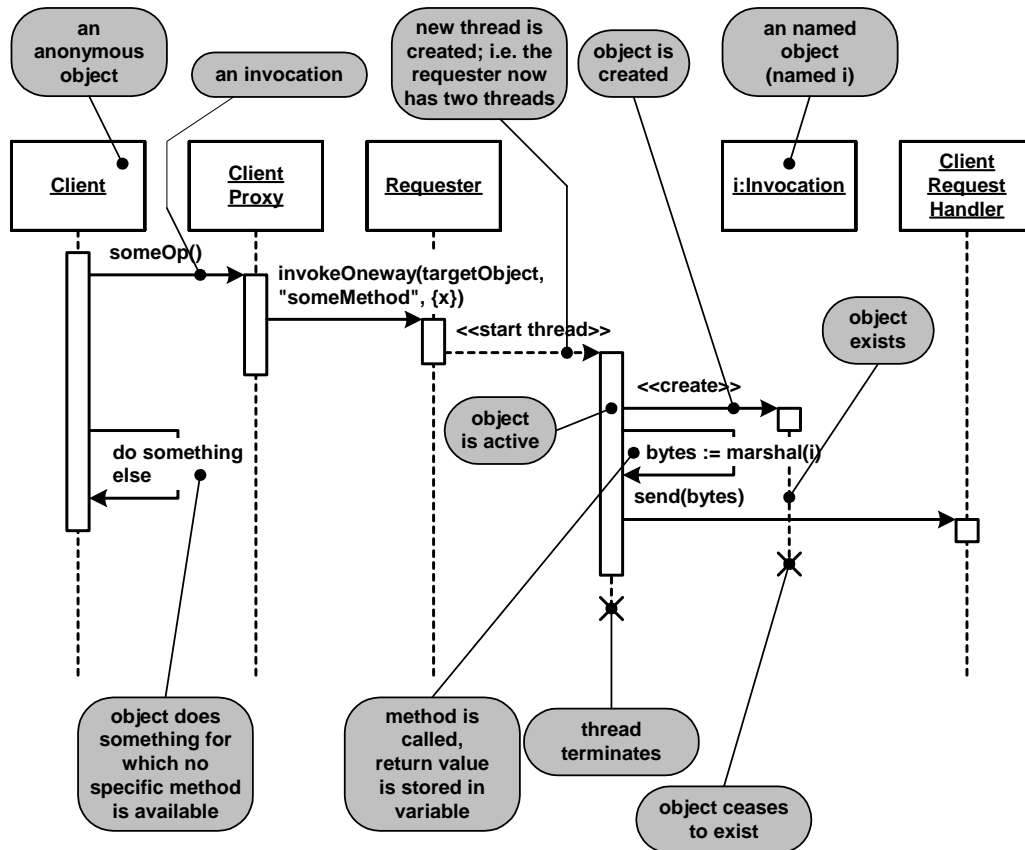
Class diagrams are standard UML, except for the dependency arrow, which we draw as a dashed arrow instead of a lined arrow. This is purely for aesthetic reasons.



Interactions

At the end of each pattern chapter we illustrate the patterns, their use, and their collaborations with a set of sequence diagrams. These are mostly standard UML and thus require no specific explanation. However, we use some non-standard but intuitive notations, which the following example illustrates. For example, a multi-threaded object has

two lifelines; and we use a stereotype <<create>> to denote the creation of other objects.



5

Introduction To Distributed Systems

I In this chapter we briefly introduce distributed systems. In particular, we motivate the basic challenges addressed by the patterns in this book and introduce important terminologies and concepts. This chapter, however, cannot provide a complete introduction to the broad field of distributed systems. More extensive introductions to these topics can be found in other books such as Tanenbaum and van Steen's *Distributed Systems: Principles and Paradigms* [TS02].

Distributed Systems: Reasons and Challenges

This section provides an introduction to distributed systems, the reasons why they exist, and the challenges in their design.

Application Areas for Distributed Systems

The application areas for distributed systems are diverse and broad. Many major large and complex systems in use today are distributed systems. In the following we will give a few examples to better illustrate this.

The *Internet* can be seen as a large distributed system with a huge number of servers providing services via a set of communication protocols, such as HTTP, FTP, Telnet, and SMTP, to an even larger number of clients. The protocols used in the Internet are simple, rugged, and proven.

Telecommunication networks use digital switches that run software to provide the communication facilities. That is, telecommunications networks heavily depend on distributed communication. As the development cycles are getting shorter in this domain, the use of low-level programming and communication means is replaced by object-oriented programming languages and corresponding distributed object middleware technologies. Those allow for higher programming

efficiency, though often at the cost of memory overhead and somewhat reduced execution performance.

Business-to-business (B2B) collaboration systems, conducting electronic commerce among businesses, are also an important application area for distributed systems. Today Electronic Data Interchange (EDI) [Ans04] is heavily used for electronic commerce, but web-based technologies such as XML-based Web Services will be most probably used in the future in this area (see for instance BEPL4WS [ACD+03]).

International financial transactions are executed via a distributed system provided by SWIFT (Society of Worldwide Interbank Financial Telecommunication). SWIFT is an industry-owned cooperation supplying secure, standardized messaging services and interface software to financial institutions. In 2002 there have been more than 7,500 financial institutions in 200 countries connected to SWIFT processing about 7 million messages daily [Swi02]. The system was originally established to simplify the execution of international payments. Today there have been several financial and infrastructure services added to the portfolio.

Embedded systems, such as in-vehicle software, elevator control systems, household appliances, mobile devices, and many others, have crucial requirements for distributed communication. For example, modern passenger cars contain a complex network of so-called electronic control units (ECU). These ECUs run software that controls a particular aspect of the car. The features of a car are typically interrelated: for example, you should not be able to switch critical ECUs into diagnostic mode while the car is driving. Thus the ECUs need to communicate with each other. In this example - similar to other embedded systems - strict timing requirements have to be obeyed. More and more embedded systems become network-aware, using for example wireless technologies, and therefore requiring efficient programmability of distributed communication. The upcoming AUTOSAR middleware standard [Aut04] addresses these concerns in the context of in-vehicle software.

Many *scientific applications*, such as DNS analysis, extraterrestrial signal interpretation, or cancer research, need massive amounts of computational power and thus cannot easily be run on a single machine.

Clusters, GRIDs, or distributed peer-to-peer systems are commonly used to collaboratively solve these kinds of problems.

These were only a few examples of distributed systems. Many other fields use distributed systems as well. We hope this short and incomplete list of examples gives you as a reader an impression of the *diversity* and *complexity* of distributed systems.

Reasons for Using Distributed Systems

Why do we use distributed systems? There are many reasons why distributed systems are used today. In general, we can distinguish *problem-related* reason and *property-related* reasons, often occurring together.

Problem-related reasons for distributed systems occur when we face problems that are inherently distributed. For instance, if an user in Europe wants to read a Web page that is located on a server in the US, then the Web page has to be transported to the remote user – there is no way around that, it is simply the purpose of the system.

There are also property-related reasons for using distributed systems - that is, reasons that are motivated by a particular system property that should be improved by distributing the system. Examples of such system properties are:

- *Performance and Scalability:* If a system has to cope with such heavy loads that a single machine cannot cost-effectively solve the problem, the problem is divided and assigned to several machines, now sharing the load. To allow them to efficiently handle the load, the machines have to be coordinated in some way (this process is called “load balancing”), resulting in a distributed system. For example, most Web sites with very high hit rates are served by more than one machine. Another example is the area of super-computing: GRIDs of smaller machines are assembled to provide a performance unattainable by any single machine. We discuss availability and scalability patterns in Chapter *Related Concepts, Technologies, and Patterns*.
- *Fault Tolerance:* Another reason for using distributed systems is fault tolerance. All hardware devices have some Mean Time Between Failure (MTBF), which in reality is smaller than infinity.

Software can also fail: a program might have a bug or some non-deterministic behavior that results in a failure or in “strange behavior” that may happen only occasionally. As a consequence, every machine, or a part of it, will fail at some time. If the overall system should continue working in such cases, it must be made sure that the system does not fail completely when a partial fault occurs. One approach to do this is to distribute the software system over many machines, and make sure clients do not notice when one of the serving machines fails. Again, coordination among different software components running on these machines results in a distributed system. Note that there are also other ways for providing fault tolerance than hardware redundancy. For more details, refer to the Chapter *Related Concepts, Technologies, and Patterns*.

- *Service and Client Location Independence*: In many systems the location of clients and services are not known in advance. In contrast to classical mainframe or client/server systems, services can be transparently executed on remote hosts, for instance, equipped with more storage or computing power. For example, in a peer-to-peer system, new - distributed or local - peers, providing additional services, might join and leave the system from time to time.
- *Maintainability and Deployment*: Deployment of software to a large number of machines is a maintenance nightmare increasing the Total Cost of Ownership (TCO) of a system. An alternative solution is to provide thin clients that only accept user inputs and present outputs. The business logic is remotely located on central servers. Changes to the business logic do not affect clients. Enterprise systems use this *thin client* approach, keeping the “actual functionality” and the data on one or more central machines.
- *Security*: Another reason for using a distributed system is security. Security information, such as user credentials, might be consistently kept at a central site and accessed from many remote places. On the other hand, for security reasons, some information or functionality might not be kept at a single site only, but instead distributed over a number of nodes, perhaps administered by different people and organizations. Also, some machines that store critical data might be located in specially secured computing centers: for example, a restricted area that requires a special key

for access and is monitored with security cameras. Often, these secured machines need to be accessed by other nodes of the system. Again, to coordinate the nodes, remote communication is necessary.

- *Business Integration:* In the past, business was mainly performed by persons that interacted with each other, either directly, per mail, or per phone. When e-mail started to appear, some business was done via mails, but still orders were typed in by persons, sitting at their personal computer. In the last years, the term “Enterprise Application Integration” (EAI) appeared. EAI is about integrating the different systems in an enterprises into one coherent, collaborating „system of systems“. This integration involves communication among the various systems using different remoting styles as well as data mapping and conversions.

Challenges in Distributed Systems

Compared to traditional, non-distributed systems there are additional challenges to be taken into account when engineering distributed systems. Important challenges are:

- *Network Latency:* A remote invocation in a distributed system takes considerably more time than an invocation in a non-distributed system. In case a certain performance level is required from the distributed system or when strict deadlines have to be obeyed, this additional time delay has to be taken into account.
- *Predictability:* The time that it takes to invoke an operation differs from invocation to invocation because it depends on the network load and other parameters, such as the actual location of the remote process, how fast the invocation travels across the network and is processed by the remote process. The network can even fail. Lacking end-to-end predictability of remote invocations is especially problematic for systems with hard real-time requirements which demand that specified deadlines must not be missed – otherwise it constitutes a system failure. Guaranteeing real-time requirements in distributed systems is a major challenge in many distributed, real-time embedded systems, such as aircraft flight control. Note that predictability cannot be assumed in non-distributed systems either. An operation invocation time is influenced by

many factors, such as processor load or the status of a garbage collector. However, for many systems, these times can be assumed to be very small and constant - and thus can be neglected in practice. Such assumptions cannot be made in a distributed system. Predictable transmission times in distributed systems can be provided only by using time-triggered communication [Kop97], which is increasingly used in safety-critical real-time applications.

- *Concurrency*: Other problems arise from the fact that there is real concurrency in a distributed system. In contrast to multi-threaded or multi-process systems running on a single processor machine, in distributed systems several processing steps can *really* happen at the same time. Coordinating such systems is far from simple. Even basic coordination, such as providing a common time, requires sophisticated protocols. Concurrency can result in a number of problems, such as non-determinism, deadlocks, and race conditions.
- *Scalability*: Since different parts of a system, such as clients and servers, are distributed and therefore more or less independent, it is not always possible to know in advance how many of these different systems are actually available and how high the communication load is going to be at a certain time. For example, it is hard to determine the number of clients for a Web server: at certain peak times, many more clients than expected might want to access the content of a Web site. The software, the hardware, and the network must be able to scale up to handle this additional load at any time - or at least fail gracefully.
- *Partial Failure*: In systems that run on a single machine, ideally in a single process, a failure, such as a hardware problem or a fatal software bug, usually has a simple consequence: the program stops running - completely. It is not possible that only parts of the program stop, which is what partial failure is all about. In distributed systems, this is different: it is very well possible that only a part of the overall system fails. Designers might have considered this behavior - see the failover discussion above. However, in many systems it can become very difficult to detect what part of a system has actually failed, and how to recover. For example, when a client wants to communicate with a server and does not receive a reply, this can have many reasons: The server process may have

crashed. Or, the server machine may have gone down. Or, the process has not crashed, but only overloaded and it may just take more time to answer the request. Finally, there might be no problem with the server at all, the network might be broken, overloaded, or unreliable. Depending on how the system failed, different ways of recovery are necessary. Therefore, the real problem is to determine the actual cause of the failure. It is sometimes impossible for clients to detect failures. There are many algorithms to detect such problems (see [TS02]), but none of them is simple, and all imply an additional performance overhead.

As a bottom line, you should only distribute your system if distribution is really needed. Distribution adds complexity, concurrency, inefficiency, and other potential problems to your application that would not occur in a non-distributed context. Note that we do not advocate to avoid the use of distributed systems; it is just important to carefully evaluate when to distribute and what to distribute - check also Fowler's "First law of distributed object design" in [Fow03]: "Don't distribute your objects!"

Communication Middleware

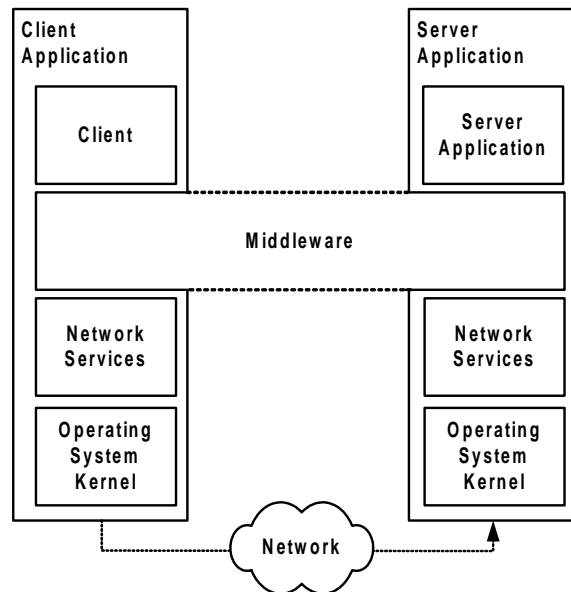
Distributed systems can be built directly on top of low-level network protocols. For instance, communication can be based on TCP/IP sockets which allow distributed processes to pass each other messages using *send* and *receive* operations directly [Ste98]. But this yields a number of problems because developers have to deal with low-level networking details. In particular, such systems:

- are usually not easy to *scale*,
- are typically rather *cumbersome* and *error prone* to use for developers,
- are hard to *maintain and change*, and
- do not provide *transparency* of the distributed communication.

A common solution to these problems is to add an additional software layer, called *Communication Middleware*, or simply *Middleware*, that hides the heterogeneity of the underlying platforms and provides transparency of distributed communication for application developers. In this context, transparency aims at making remote invocations as

similar as possible to local invocations. However, since remote invocations introduce new kinds of errors, latency, and so forth, complete transparency is not possible. This aspect will be addressed in more detail later in this book.

The figure below shows that the middleware hides the network services and other details of remote communication from clients and server applications.



The middleware is an additional software layer that sits between the network services, offered by the operating system, and the application layer hosting the client and server application. Due to the additional layer the middleware hides the heterogeneity of the underlying platforms from application developers. Clients and server applications are usually not allowed to bypass the middleware layer in order to access low-level network services directly.

As outlined in the next section, different middleware systems are based on different remoting styles, such as remote procedure calls or message passing. Whatever remoting styles is used today in an application, the application developer should be shielded from the details of how the respective middleware implementation really communicates over the

network. The middleware takes care of the specifics of realizing the remoting styles using available lower-level services. Middleware provides a simple, high-level programming model to the developers, hiding all the nitty-gritty details as good as possible. For example, a middleware might take care of connection handling, message passing, and serialization, but does not try to hide the specific errors conditions that are introduced by the fact that an invocation target is located remote. Middleware specifications and products are available for all popular remoting styles. The technology projections introduced later in the book show some examples of distributed objects middleware.

Remoting Styles

There are a number of different remoting styles used in today's middleware systems. In this section, we introduce some popular remoting styles and explain a few prominent example of middleware systems for each of the styles.

Historically, distributed computing is derived from simple *file transfers*. File transfer was the original means of program-to-program communication. And it is still the basis of many mainframe systems today.

The principle of file transfers is that each program works on a physical file. The programs accept files as its input and create new files, or modify existing files, after that the files are exchanged with the next program. While file transfer has proven effective for solving many problems, the issues of latency and resource contention have made it unattractive for today's high-speed, high-volume systems.

In the following sections, we take a brief look at the following basic remoting styles that are used instead of file transfer today. There are systems that

- use the metaphor of a *remote procedure call* (RPC),
- use the metaphor of posting and receiving *messages*,
- utilize a *shared repository*, or
- use continuous *streams* of data.

This book mainly focusses on object-oriented variants of the RPC style. However, all of the mentioned remoting styles can be used to implement the others. Also, there are interrelations between the styles, such

as a shared repository accessed with RPC or asynchronous RPC invocations sent as messages.

Because of these interrelations among the remoting styles, it is important to understand the diversity in remoting styles in order to properly understand the pattern language contained in this book. Also, it is important to understand that there are many other remoting styles that are based on top of the basic remoting styles. Examples of such more advanced remoting styles are code mobility, peer-to-peer (P2P), remote evaluation, GRID computing, publish/subscribe systems, transaction processing, and many others. Note that these more high-level remoting styles are often implemented using the basic styles, or as variants of them. However, the users of these styles are not confronted with their internal realization. For example, the user of a P2P system, which is based on RPC, does not have to deal with the internal RPC mechanisms, nor with the naming service used for ad hoc lookup of peer services.

In the chapter *Related Concepts, Technologies, and Patterns* we discuss how some of these more advanced remoting styles can be realized using the pattern language presented in this book. When new remoting styles emerge, usually no complete, industry-level implementation is available. Thus, parts of the system have to be built by hand using low-level remoting styles. Consider a small, custom publish/subscribe system built on top of an OO-RPC middleware: in order to build such a system, the internals of the OO-RPC middleware, as well as the possible ways to extend, customize, and integrate it, have to be well understood. The pattern language in this book provides a foundation for understanding distributed object middleware systems in such a way.

Remote Procedure Calls

Remote procedure calls extend the well-known procedure call abstraction to distributed systems. They aim at letting a remote procedure invocation behave as if it were a local invocation.

Use of low-level network protocols requires developers to directly invoke the *send* and *receive* operations of the respective network protocol implementations. Remote Procedure Call (RPC) systems [BN84] introduce a simple but powerful concept to avoid this problem:

programs are allowed to invoke procedures (or operations) in a different process and/or on a remote machine.

In RPC, two different roles are distinguished: *clients* and *servers*. Clients invoke operations. Servers accept operations. When a client process invokes an operation in a server process, the invoking client process is suspended, and the execution of the invocation in the server takes place. After, the server sends the result back to the client and the result is received, the client resumes its work.

A server provides a well defined set of operations which the client can invoke. To the client developer these operations look almost exactly like local operations: they typically have an operation name, parameters, and a return type, as well as a way to signal errors. One major difference to ordinary operations is that additional errors might occur during a remote invocation, for instance, because the network fails or the requested operation is not implemented at the server. These errors can be signaled to the client, for instance, as specific kinds of exceptions.

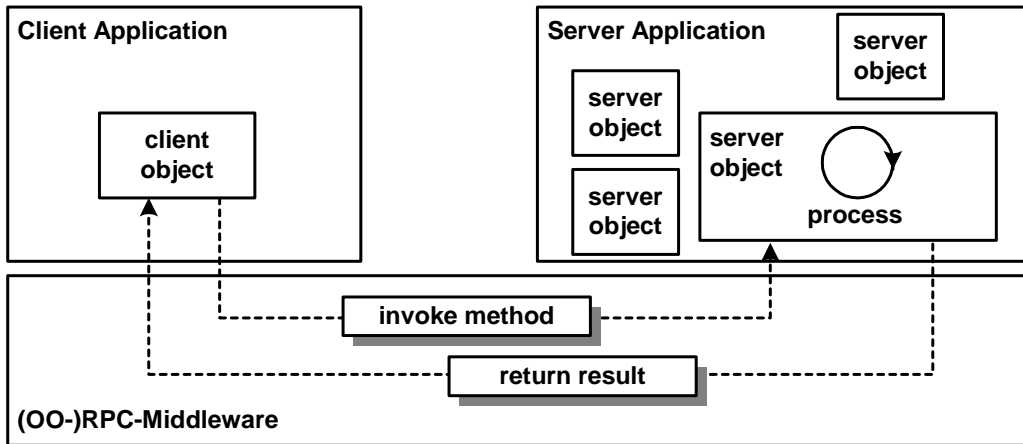
As an extension to the RPC style, there are also many asynchronous RPC variants. That is, a client immediately resumes its work after it has invoked a remote operation, without blocking until the result becomes available.

There are two different, popular flavours of remote procedure call systems:

- Those that really use the procedure-based approach, where a server application provides only the specified operations to clients, and
- object-oriented remote procedure call systems (OO-RPC), where a server application hosts a set of objects that provide the operations to clients as part of their public interface.

The internal mechanisms are in both variants similar; however, in OO-RPC systems there is the notion of object identity – meaning that clients can address objects separately. Also, a distributed object can have its own state whereas RPC-based systems often provide stateless services. In addition, the OO-RPC approach maps more naturally into today's object-oriented way of software development and is therefore used almost exclusively in new systems.

The figure below shows an example of an OO-RPC system: the OO-RPC middleware allows clients to transparently access objects within a server process.



Popular procedure-based systems are the Distributed Computing Environment (DCE) [Ope91] and Sun RPC [Sun88]. DCE, for instance, was developed by the Open Software Foundation. Originally designed as a RPC middleware for Unix platforms, it has later been ported to all major operating systems, including Windows variants and VMS.

DCE introduces a typical and straightforward way of performing RPC, which allows to automatically locate the server hosting the operations of interest (during a process called *binding*): first, the server registers a procedure as an endpoint with the DCE daemon running within the server's machine, then it registers the service in a directory server that might run on a different machine. When the client wants to access the remote procedure, it first looks up the server in the directory server. The server is then asked for the endpoint. This endpoint is then used to invoke the remote procedure.

Even though DCE is a procedure-based middleware, it also provides extensions for supporting remote objects. Nowadays there are many popular OO-RPC middleware systems specifically designed for distributed object communication. Examples of such systems are:

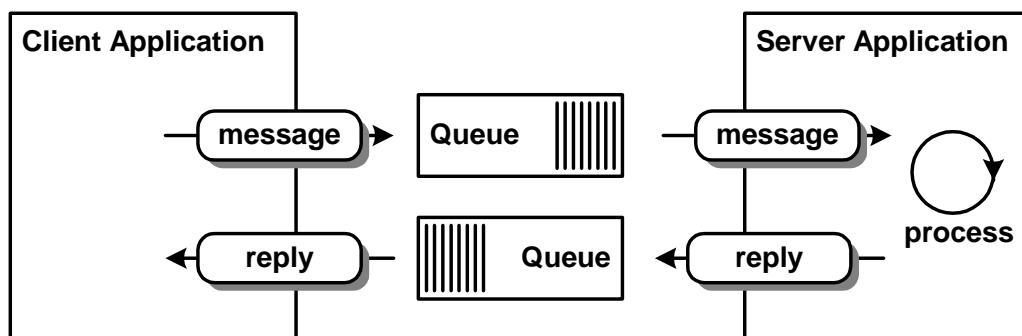
- Common Object Request Broker Architecture (CORBA) (see *CORBA Technology Projection* Chapter),

- Microsoft's .NET Remoting (see *.NET Remoting Technology Projection* Chapter).
- Web Services (see *Web Services Technology Projection* Chapter),
- Microsoft's DCOM [Gri97], and
- Sun's Java RMI [Gro01]

These middleware systems and their concepts are the primary focus of this book.

Message Passing

While the (OO-)RPC middleware uses the metaphor of a procedure (or operation) invocation, the *message passing* metaphor is different. Messages are exchanged between peers, and the peers are free to consume the messages as and when they like. Message passing is inherently asynchronous.



Messages are not passed from client to server application directly, but through intermediate message queues that store and forward the messages. This has a number of consequences: Senders and receivers of messages are decoupled; they do not need to know each other's identity. A sender just puts messages into a certain queue and does not necessarily know who consumes the messages. For instance, a message might be consumed by more than one receiver. Receivers consume messages by "listening" to queues.

The two peers of a remote communication are timely decoupled: a sender can put a message into the message queue while there is no receiver running or connected. Receivers can later get the messages from the queue. Thus, the messaging style is inherently asynchronous.

Replies as part of the middleware are not foreseen. If applications require replies, they have to be modelled and sent as separate messages.

Many messaging protocols provide a reliable means of message delivery, such as an acknowledgement scheme. Further, the ordering of messages is ensured, so that receivers process messages in the same order as they were sent, even if messages had to be retransmitted in between. Thus, messaging provides a way to tolerate temporal failures of the peers. Also, many message passing systems guarantee that once the message is put into the queue it will be delivered exactly once and only once to the receiver. Consequently, messaging systems can provide a high level of reliability to their users.

The message passing style allows for many variations in the implementations. Some systems distinguish between queues where there is one receiver, or several. In the latter case, each message is delivered to each receiver. This is often referred to as publish/subscribe systems.

Typically, messages are delivered asynchronously, as explained above. However, some systems allow for synchronous delivery, blocking the sender until the message has been delivered successfully. Similar to RPC invocations, the message is sent to the receiver, and then a response is awaited. This mode is provided by messaging systems, for instance, to support typical client/server transaction characteristics. As a side effect, this technique can be used to coordinate concurrent systems.

Messages can be persistently buffered in the message queue in case the receiver is not accessible. A message can then be guaranteed to be delivered exactly once.

Several messages can be delivered to one or more clients atomically in the context of a transaction. However, there are some limitations to transaction processing, when using the message passing style. For instance, sending a message and receiving the same message in one distributed transaction is not possible. This would violate the desired transaction property “isolation” because entering a message into a destination and removing a message from a destination are two distinct operations that must be executed in two different transactions.

As discussed above, the message passing style is conceptually different from a remote procedure call for the reasons given above. Technically however, it is easily possible to implement RPC semantics with a message passing system, or implement message passing using RPC. We will discuss the latter variant briefly in the *Invocation Asynchrony Patterns* Chapter.

There are a number of simple message passing systems, including Berkeley Sockets [Lem97] and the Message-Passing Interface (MPI) [Qui03], allowing mainly for asynchronous communication, and a number of message queueing systems allowing also for persistent messaging.

Message-Oriented Middleware (MOM) extends the concept of such simple message passing systems with support for inter-process messaging and persistent message queueing. That is, reliable communication is enabled: the MOM stores messages intermediately, without having the sender or receiver actively participate in the network transfer. Once a message is put into a queue it remains there until it is removed. This model allows for loosely-coupled communication: a receiver does not have to directly consume a message, when it arrives. This way, for instance, a crash of the server can be tolerated. Popular MOM products are IBM's WebSphere MQ (formerly MQ Series) [Ibm04], JMS [Sun04c], Microsoft's MSMQ [Mic04a], and Tibco [Tib04].

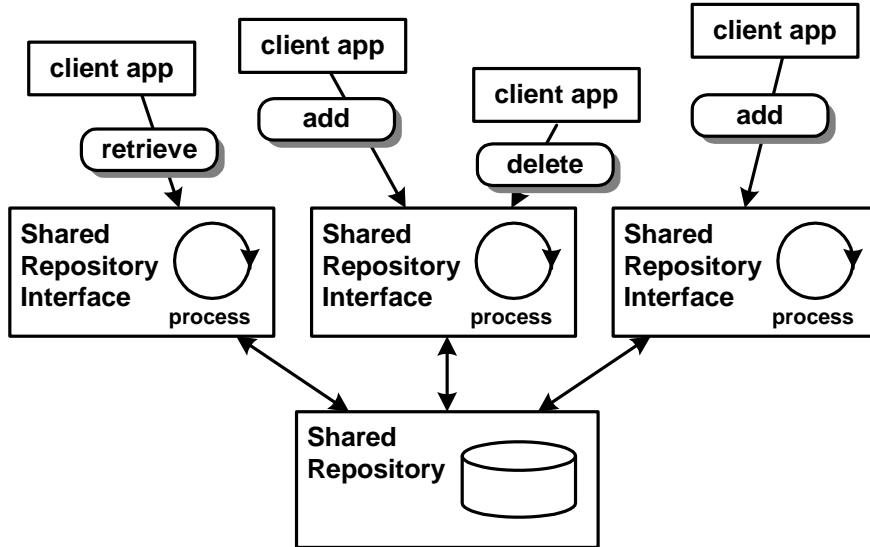
We discuss further details of messaging and how it relates to the pattern language presented in this book in Chapter *Related Concepts, Technologies, and Patterns*.

Shared Repository

A shared repository is based on the idea that there is some shared data space, and independent, remote clients interact by reading and writing to the shared data space.

There are two basic roles in the shared repository remoting style: the *shared repository*, which is also referred to as the *Blackboard* [BMR+96], and its *clients*. The shared repository offers a small interface consisting

of access primitives to the clients. Note that the data access in a shared repository is often implemented using RPC mechanisms.



Systems for shared repositories exist in many different flavours. The most simple variant is a shared memory or file space with which different processes interact. When memory and file space are shared among different machines, a distributed system is created. There are many more high-level shared repositories. In addition to simply sharing data, the solutions have to:

- handle problems of resource contention, for instance by locking accessed data,
- abstract from location of the shared data, the data storage, or the client,
- optimize performance and scalability, and
- provide additional services, such as security.

Some systems even introduce high-level access mechanisms, such as query languages or tuple spaces [GCC85].

Databases are one way to implement a shared repository. In a database, the database management system (DBMS) provides mechanisms for locking and unlocking the data, and also provides standard mechanisms for creating, deleting, searching, and updating information.

Most DBMSs provide query languages for high-level data access. Shared databases might utilize transaction processing monitors to support distributed transactions.

Some approaches, such as Linda [GCCC85] or PageSpace [CTV+98], are based on the concept of Virtual Shared Memory (VSM). VSM is a shared object repository that can be used to store shared data. It is virtual in the sense that no physically shared memory is required. The underlying data structure of Linda is a tuple space: processes do not communicate directly, but rather by adding and removing tuples (ordered collections of data) to and from the tuple space. The tuples are then stored in shared memory. Tuple space based systems typically provide only a very small set of operations on the shared tuple set. It has been demonstrated that many problems of distributed and concurrent applications can be solved very elegantly using this approach. Other examples of tuple-based systems include Sun's JavaSpaces [FHA99] and IBM's TSpaces [Ibm00].

Streaming

While the previously explained styles used discrete and complete units of data that are exchanged (invocation requests, messages, or data items); it is different for stream-based systems: here, information is exchanged as a continuously flowing stream of data.

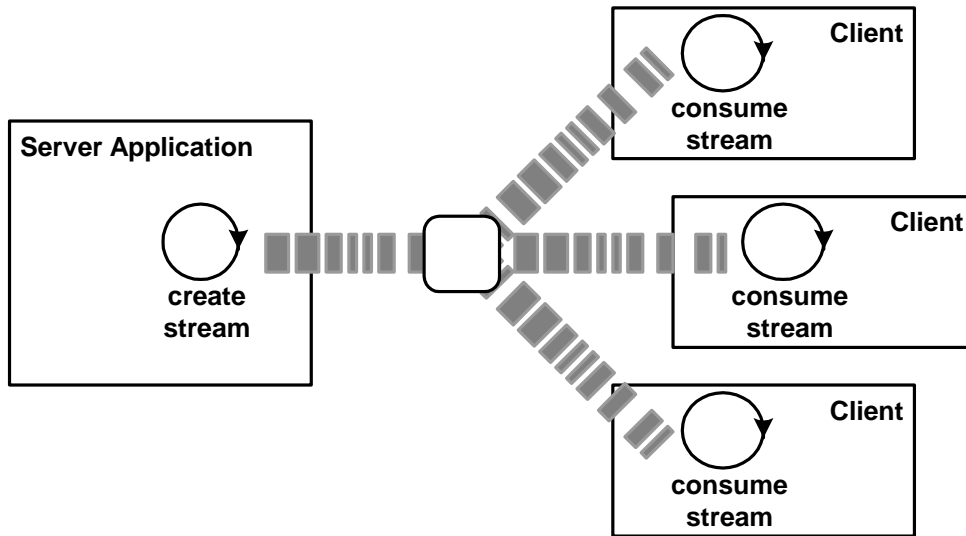
The two different roles to be distinguished are *streaming servers* and *clients* receiving streams. There are several different types of *streams* (see [TS02]):

- *Asynchronous* streams consist of data where the sequence of packets is important, but not the timing. An example could be a file transmitted over a TCP/IP socket. Whether the transfer rate is slow or fast does not matter. A slower rate will result in a longer transfer time of the file, but the file is nonetheless transferred correctly.
- *Synchronous* streaming is inherently time dependent. The data packets have to be transferred at a certain minimum rate, otherwise the data may be useless to the receiver. Thus there is maximum delay between two data packets. For example, subsequent measurements by a sensor might have to be transmitted at a

certain rate to achieve the necessary accuracy of a subsequent calculation.

- *Isochronous* streaming has even more rigid requirements regarding timing. Data items have to be transferred *on* time, not just *in* time. That is, there is a maximum delay between data packets as well as a minimum delay. Typical examples are audio or video streams.

Stream-based systems can follow a one-to-one or one-to-many streaming concept. In the latter case, depending on the streaming technology, the stream might still have to be transported separately to each receiver.



In the figure above an example configuration is shown: one stream server application provides continuous streams of data to multiple clients. Often, streams have to be received by several clients at a time. In a web-based video or audio broadcast, there is typically a single stream server and a relatively large number of clients. The necessary bandwidth must be ensured on the whole data path, from the server to all of the clients.

The primary challenge with stream based systems, especially isochronous ones, is to ensure the necessary timing requirements, typically referred to as quality of service (QoS).

Stream-based systems are not considered in this book.

6 Pattern Language Overview

This chapter provides an overview of the pattern language described in the next chapters. We start off by recasting the *Broker* pattern from POSA1 [BMR+96] as the entry point into the pattern language that follows. From our perspective, the *Broker* pattern is a compound pattern that is typically implemented using a number of patterns from our pattern language. Especially the patterns in the *Basic Remoting Patterns* chapter are used almost any Broker architecture.

Next, we introduce two important participants of *Broker*-based systems in detail: the remote object and the server application. This is important because these two participants are used in almost any distributed object application and thus are ever-present throughout our pattern language.

|

Broker

You are designing a distributed software system.



Distributed software system developers face many challenges that do not arise in single-process software. One main challenge is the communication across unreliable networks. Other challenges are the integration of heterogeneous components into coherent applications, as well as the efficient usage of networking resources. If developers of distributed systems must master all these challenges within their application code, they will likely lose their primary focus: to develop applications that resolve their domain-specific responsibilities well.

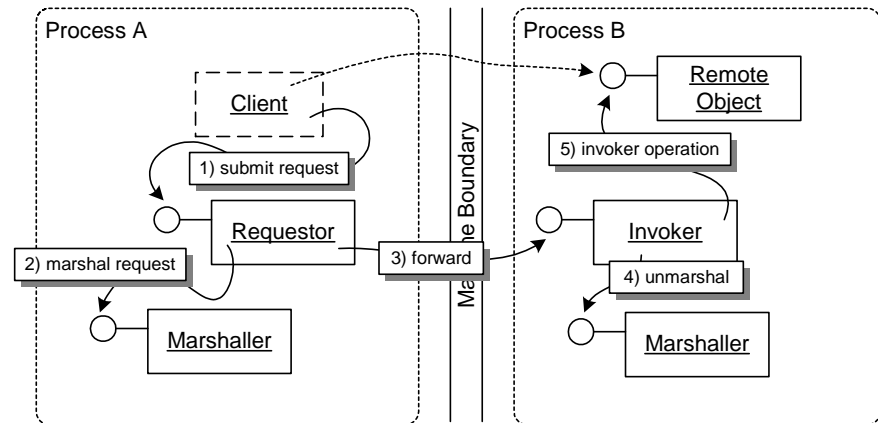
Communication across networks is more complex than local communication because connections need to be established, invocation parameters have to be transmitted, and a new set of possible errors has to be coped with. This requires handling invocations to local objects differently than invocations to remote objects. Additionally, tangling remote communication concerns with the overall application structure complicates the application logic. From an object-oriented programming perspective, it would be ideal, if the remote objects could be invoked as if they were local objects.

Also, the location of the remote objects should not be hard-wired into client applications. It should be possible to host remote objects on different machines without adaptation of the client's program code.

Therefore:

Separate the communication functionality of a distributed system from its application functionality by isolating all communication related concerns in a BROKER. The BROKER hides and mediates all communication between the objects or components of a system. A BROKER consists of a client-side REQUESTOR to construct and forward invocations, as well as a server-side INVOKER that is responsible for invoking the operations of the target remote object. A MARSHALLER on each side of the communications path handles the transformation

of requests and responses — from programming-language native data types into a byte array that can be sent over the wire.



The client submits a request to the REQUESTOR that marshalls the request and transmits it across the network. The server side INVOKER receives the invocation request and invokes the remote object.



Local BROKER components on client and server side enable the exchange of requests and responses between the client and the remote object. In addition to the core patterns consisting of REQUESTOR, INVOKER, and MARSHALLER, the BROKER typically relies on the following patterns:

- A CLIENT PROXY represents the remote object in the client process. This proxy has the same interface as the remote object it represents. The CLIENT PROXY transforms invocations of its operations into invocations of the REQUESTOR that, in turn, is responsible for constructing the request and for forwarding it to the target remote object. A CLIENT PROXY is thus used to let remote operation invocations look like local operation invocations from a client's perspective. An INTERFACE DESCRIPTION is used to make the remote object's interface known to the clients. The client makes use of the INTERFACE DESCRIPTION in the form of a CLIENT PROXY.
- LOOKUP allows clients to discover remote objects. This ensures that the location of remote objects does not need to be hard-wired into

the system. Remote objects can be moved to other hosts without compromising system integrity, and the location of remote objects is in control of the server developers.

- The CLIENT REQUEST HANDLER and SERVER REQUEST HANDLER handle efficient sending, receiving, and dispatching of requests. Their responsibility is to forward and receive request and response messages from and to the REQUESTOR and the INVOKER, respectively.
- REMOTING ERRORS are used to signal problems of remote communication to the client side. REMOTING ERRORS are caused either by technical failures in the network communication infrastructure or by problems within the server infrastructure. The REQUESTOR and INVOKER are responsible for forwarding REMOTING ERRORS to the client, if they cannot handle the REMOTING ERROR on their own.

In the following section we take a closer look at two important, additional participants of the BROKER-based applications: remote objects and the server application. Note that these two are not participants of the BROKER pattern itself. They are merely necessary in applications that use a BROKER as their communications backbone.

Remote Object

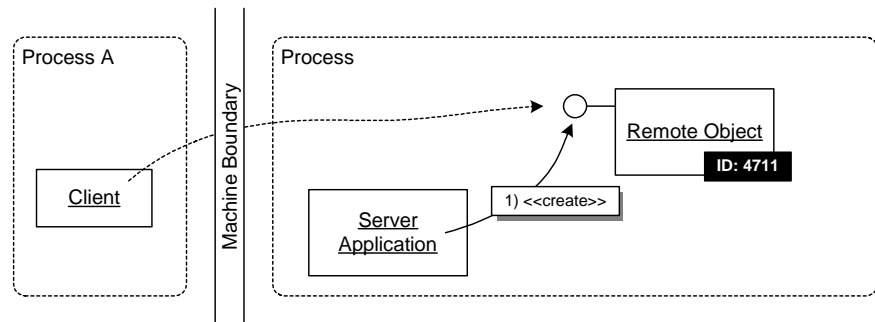
Consider you are using or building a distributed object middleware that is used to access objects remotely. Clients need to access functionality provided by remote application objects. In many respects, accessing an object over a network is different from accessing a local object. An invocation has to cross process and machine boundaries. Network latency, network unreliability, and many other distinctive properties of network environments play an important role and need to be “managed” for the developer. In particular:

- For a remote invocation, machine and process boundaries have to be crossed. An ordinary, local operation invocation is not sufficient, because the operation invocation has to be transferred from the local process to the remote process, running on the remote machine.
- Using memory addresses to define object identity will not work anymore. While unique in the address space of one process, they

are not necessarily unique across process boundaries and especially machine boundaries.

- Compared to local invocations, invocations across a network involve unpredictable latency. Because networks must be considered to be unreliable, clients must deal with new kinds of errors. Further, the communication path should be as short as possible, limiting the number of network hops.

The distributed object middleware provides solutions for these fundamental issues of accessing objects remotely using a BROKER architecture. The application logic, implemented by server developers is provided in form of *remote objects*. These remote objects are important conceptual building blocks for these distributed applications. Each remote object provides a well-defined interface to be accessed remotely; that is, the remote client can address the remote object across the network and invoke its operations. The BROKER transfers local invocations from the client side to a remote object, running within the server.



The server application instantiates a remote object. Then clients can access the remote object using the functionality provided by the distributed object middleware.

Remote objects have a unique OBJECT ID in their local address space, as well as means to construct an ABSOLUTE OBJECT REFERENCE. The ABSOLUTE OBJECT REFERENCE is used to reference and subsequently access a remote object across the network.

A remote object is in first place a logical entity in the scope of the distributed object middleware. It can be realized by different programming language entities. In an object-oriented language usually a

remote object is realized as an object of the programming language. However, that does not mean that each remote object is realized by exactly one programming language object or that all requests to one remote object are served by the same programming language object. For instance, the resource optimization pattern POOLING lets a pool of multiple objects handle requests for a remote object. In non-object-oriented programming languages other structures than objects are used to implement the remote object. To distinguish the runtime entity from the remote object as a logical entity, we call the entity that represents a remote object at runtime the *servant*. In cases, where there is exactly one servant for one remote object, we simply use the term remote object as a synonym for both, the remote object and its servant.

Clients need to know the remotely accessible interface of a remote object. A simple solution is to let remote clients access any operation of the remote object. But perhaps some local operations should be inaccessible for remote clients, such as operations used to access the distributed object middleware. Thus each remote object type defines or declares its remotely accessible interface.

Remote objects are used to extend the object-oriented paradigm across process and machine boundaries. However, accessing objects remotely always implies a set of inherent problems, such as network latency and network unreliability, that cannot be completely hidden from developers. Different distributed object middleware systems hide these issues to a different degree. When designing a distributed object middleware, there is always a trade-off between possible control and ease-of-use.

To a certain extent, these problems can be avoided by following the principle “Don’t distribute your objects” [Fow03]. However, this only works as long as the application problem is not inherently distributed in nature. If distribution is necessary, make sure you co-locate objects that require a lot of communication and design your interfaces in ways that allow for exchanging as much data as possible within one remote call - reducing the number of remote invocations to the bare minimum (see the *Data Transfer Object* and *Facade* patterns in [Fow03]). In case of errors that are related to the remote nature of a remote object, make sure there is a way to signal these REMOTING ERRORS to the client in a well defined manner.

Server Application

Remote objects need a *server application* that has to perform the following tasks:

- The constituents of the distributed object middleware have to be created and configured.
- The remote objects need to be instantiated and configured, if not done by the distributed object middleware constituents.
- Remote objects need to be advertised to interested clients.
- Individual remote objects need to get connected to distributed applications.
- If no longer needed, remote objects need to be destroyed or recycled.

To resolve these issues, use a server application as a building block that connects all parts of the distributed application. The server application instantiates and configures the distributed object middleware, as well as the remote objects on top of it. On shutdown, the server application has to ensure to free used resources by destroying remote objects, announcing their unavailability, and destroying the distributed object middleware instance.

Depending on how sophisticated the distributed object middleware is, the server application might delegate many of the remote object life-cycle management tasks to the distributed object middleware.



The server application creates an instance of a remote object, hands it over to the lifecycle manager of the distribut-

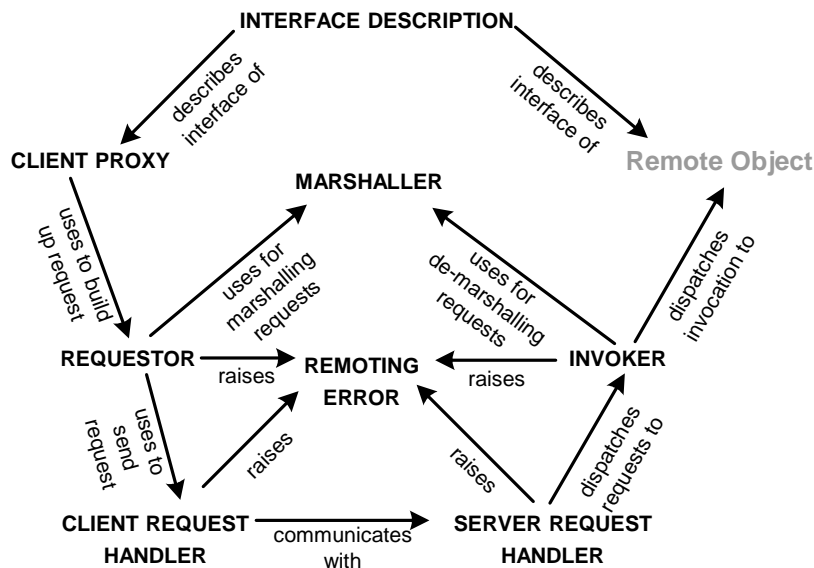
ed object middleware, and finally announces the availability of the remote object by registering it with a lookup service.

The server application bundles remote objects that conceptually belong together, and handles all object management tasks with respect to the distributed object middleware.

I

Overview of the Pattern Chapters

The patterns mentioned so far detail the **BROKER** pattern and will be described in the *Basic Remoting Patterns* chapter of this book. The following illustration shows the typical interactions of patterns.

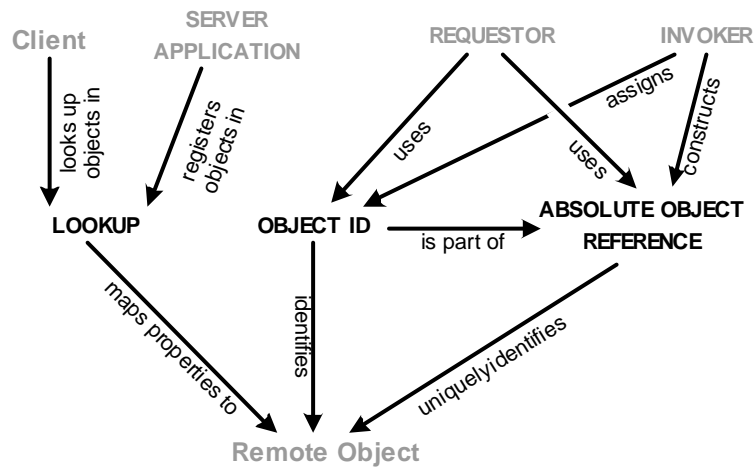


The chapters following the *Basic Remoting Patterns* chapter present patterns that extend the elementary patterns; the remainder of this section will provide an brief overview of them.

The patterns in the *Identification Patterns* chapter deal with issues regarding identification, addressing, and lookup of remote objects. It is important for clients to find the correct remote object within the server application. This is done by the assignment of logical **OBJECT IDS** for each remote object instance. The client embeds these **OBJECT IDS** in invocations so that the **INVOKER** can find the correct remote object. However, this assumes that we are able to deliver the message to the correct server application - because in two different server applications two different objects with the same **OBJECT ID** might exist. An **ABSOLUTE OBJECT REFERENCE** extends the concept of **OBJECT IDS** with location information. An **ABSOLUTE OBJECT REFERENCE** contains, for example the hostname, the port, and the **OBJECT ID** of a remote object.

LOOKUP is used to associate remote objects with human readable names and other properties. The server application typically associates properties with the registered remote object. The client must only know the ABSOLUTE OBJECT REFERENCE of the lookup service instead of the potentially huge number of ABSOLUTE OBJECT REFERENCES of the remote objects it wants to communicate with. The LOOKUP pattern simplifies the management and configuration of distributed systems as clients can easily find remote objects, while avoiding tight coupling between them.

The interactions of the patterns are visualized in the following dependency diagram.



The *Lifecycle Management Patterns* chapter deals with the management of the lifecycle of remote objects: While some remote objects need to exist all the time, others need to be available only for a limited period of time. Further, the activation and deactivation of remote objects might be coupled with additional tasks.

Lifecycle management strategies are used to adapt to specifics of the lifecycle of remote objects and their usage. The strategies have a strong influence on the overall resource consumption of the distributed application. The *Lifecycle Management Patterns* describe some of the most common strategies used in today's distributed object middleware.

The three basic lifecycle strategy patterns have the following focus:

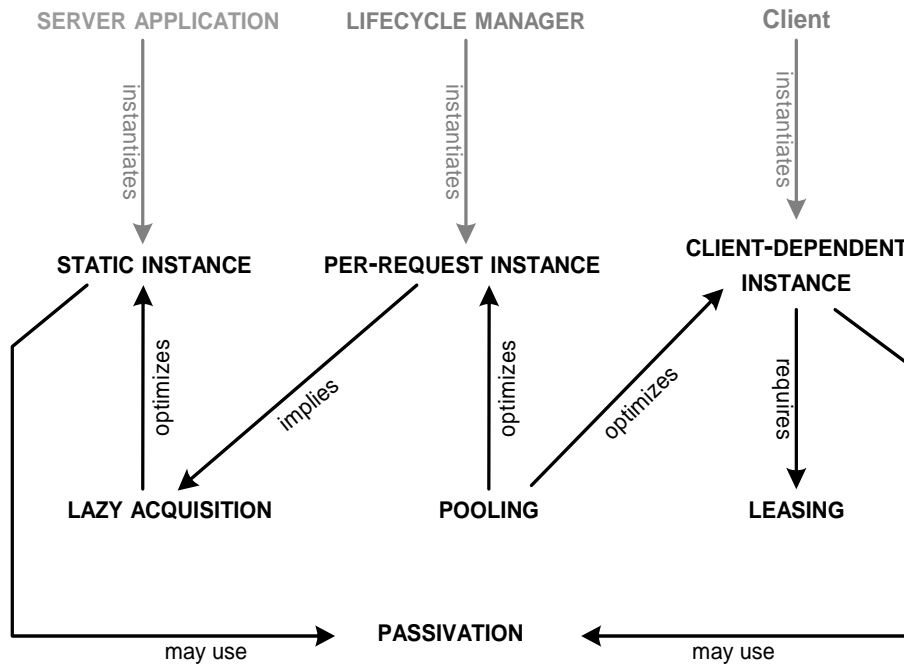
- **STATIC INSTANCES** are used to represent “fixed” functionality in the system. Their lifetime is typically identical to the lifetime of their server application.
- **PER-REQUEST INSTANCES** are used for highly concurrent environments. They are created for each new request and destroyed after the request.
- **CLIENT-DEPENDENT INSTANCES** are used to represent client state in the server. They rely on the client to explicitly instantiate them.

The lifecycle strategies patterns internally make use of a set of specific resource management patterns, which are:

- **LEASING** is used to properly release **CLIENT-DEPENDENT INSTANCES** when they are no longer used.
- **LAZY ACQUISITION** describes how to activate remote objects on demand.
- **POOLING** manages unused remote object instances in a pool to optimize reuse.

For state management, **PASSIVATION** takes care of temporarily removing unused instances from memory and storing them in a persistent storage. Upon request, the instances are restored again.

The patterns and their relationships are shown in the following figure.



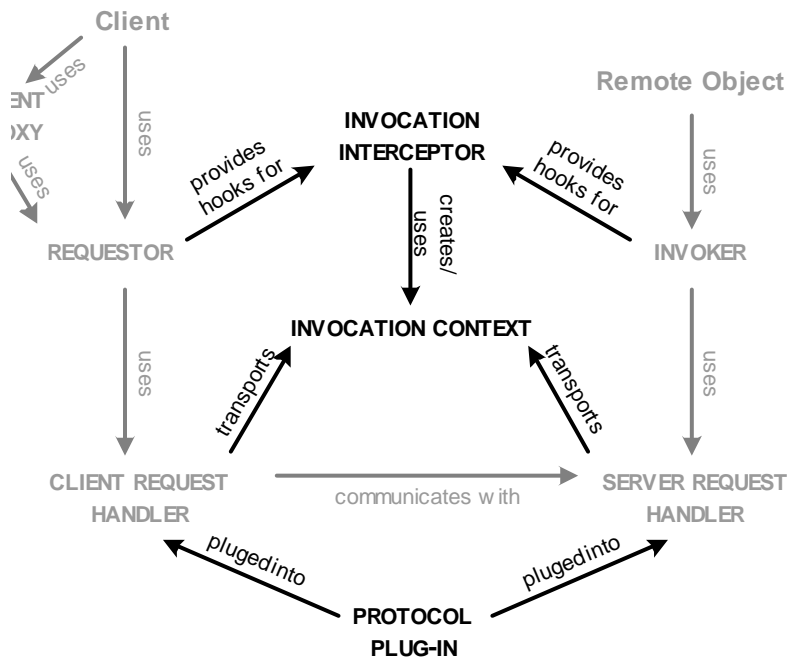
The *Extension Patterns* chapter deals with patterns that are used to extend a distributed object middleware. Examples of such extensions are the support for security, transactions, or even the exchange of communication protocols.

In order to handle aspects such as security or transactions, remote invocations need to contain more information than just the operation name and its parameters: some kind of transaction ID or security credentials need to be transported between client and server. For that purpose **INVOCATION CONTEXTS** are used: they are typically transparently added to the invocation on client side and read out on server side by the **REQUESTOR**, **CLIENT/SERVER REQUEST HANDLERS**, and **INVOKER**, respectively.

When the invocation process needs to be extended with behavior, for instance, when evaluating security credentials in the client and the server, **INVOCATION INTERCEPTORS** can be used. For passing information between clients and servers, **INVOCATION INTERCEPTORS** use the already mentioned **INVOCATION CONTEXTS**.

Another important extension is the introduction and exchange of different communication protocols. Consider again the security example: a secure protocol might be needed that encrypts the invocation data before it is sent and received using the CLIENT or SERVER REQUEST HANDLER, respectively. While simple REQUEST HANDLERS use a fixed communication protocol, PROTOCOL PLUG-INS make the REQUEST HANDLERS extensible to support multiple communication protocols.

The relationships of the patterns are illustrated in the following figure.



The *Extended Infrastructure Patterns* chapter deals with specific implementation aspects of the server-side BROKER architecture.

The **LIFECYCLE MANAGER** is responsible for managing activation and deactivation of remote objects - by implementing the lifecycle management strategies described in the *Lifecycle Management* chapter, typically as part of the **INVOKER**.

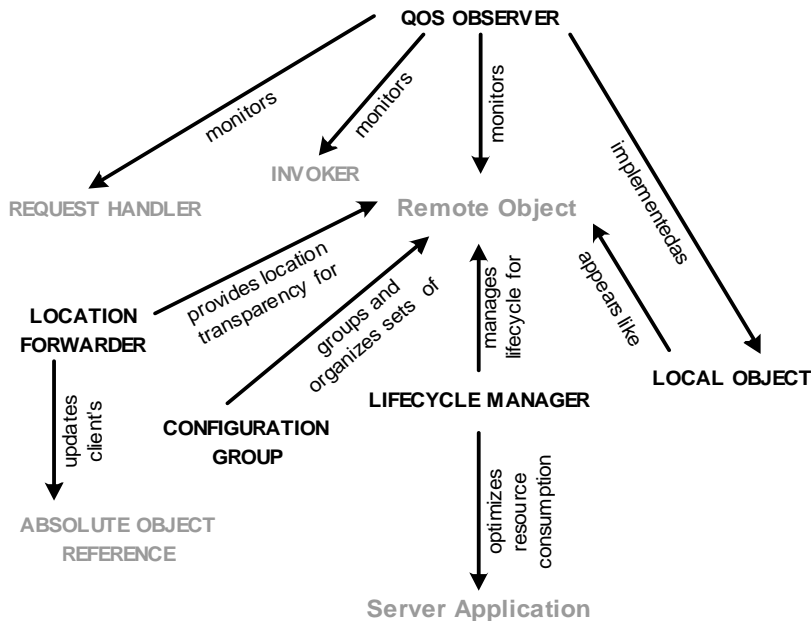
To be able to configure groups of remote objects—instead of configuring each object separately—with regards to lifecycle, extensions, and other options, **CONFIGURATION GROUPS** are used.

In order to be able to monitor the performance of various parts of the system, such as the INVOKER, the SERVER REQUEST HANDLER, or even the remote objects themselves, QOS OBSERVERS can be used. They help ensuring specific quality of service constraints of the system.

To make infrastructure objects of the distributed object middleware follow the same programming conventions as remote objects, while making them inaccessible from remote sites, LOCAL OBJECTS can be used. Typical LOCAL OBJECTS are the REQUESTOR, the LIFECYCLE MANAGER, the QOS OBSERVERS, and so forth.

ABSOLUTE OBJECT REFERENCES identify a remote object in a server. If the remote object instances are to be decoupled from the ABSOLUTE OBJECT REFERENCE, an additional level of indirection is needed. LOCATION FORWARDERS allow this, they can forward invocations between different server applications. This way load balancing, fault tolerance, and remote object migration can be implemented.

The following figure shows the Extended Infrastructure Patterns and their relationships.

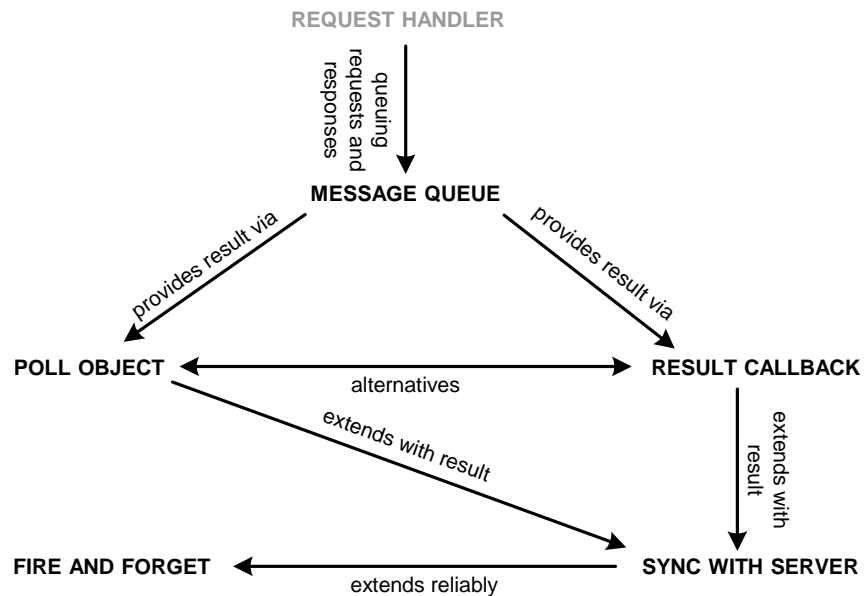


The last patterns chapter in this book describes *Invocation Asynchrony Patterns*. This chapter deals with how to handle asynchronous invocations. It presents four alternative patterns that extend ordinary synchronous invocations:

- FIRE AND FORGET describes best-effort delivery semantics for asynchronous operations that have void return types.
- SYNC WITH SERVER sends—in addition to the semantics of FIRE AND FORGET—an acknowledgement back to the client once the operation has arrived on the server-side.
- POLL OBJECTS allow clients to query the distributed object middleware for responses to asynchronous requests.
- RESULT CALLBACK actively notifies the requesting client of a asynchronously arriving responses.

MESSAGE QUEUES allow for queueing messages in the CLIENT REQUEST HANDLER and SERVER REQUEST HANDLER. They are the foundation for message-oriented communication.

The following figure illustrates the patterns and their interactions.



I

7

Basic Remoting Patterns

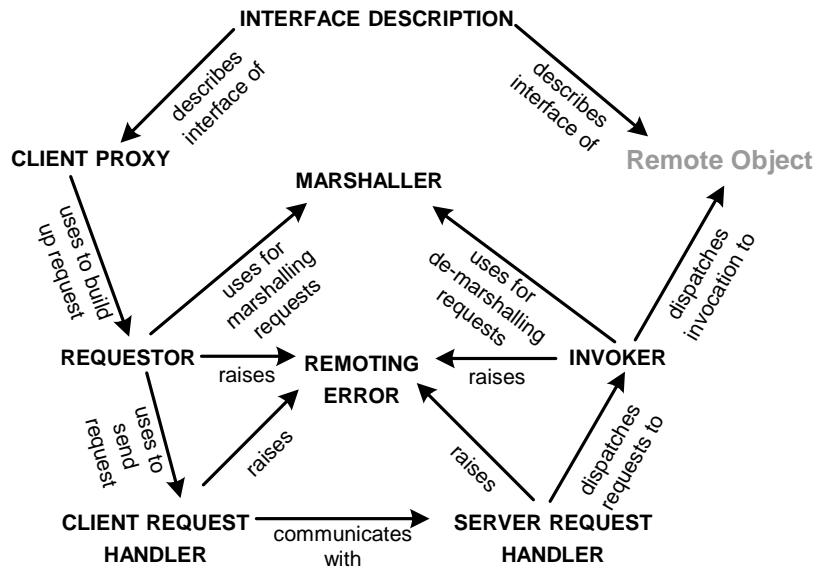
This chapter presents patterns that constitute the basic building blocks of a distributed object middleware. These patterns are used in almost every middleware implementation. Before presenting the patterns, we give an overview of the patterns and their dependencies. At the end of this chapter, we show how the participants of the patterns interact with each other.

Distributed object middleware provides an infrastructure for clients to communicate with remote objects on a remote server. Remote objects are implemented as ordinary objects on the server side. The client invokes an operation of a local object and expects it to be forwarded to the remote object.

To make this happen, the invocation has to cross the machine boundary. A REQUESTOR constructs a remote invocation on client side from parameters such as remote object location, remote object type, operation name, and arguments. A client can either use a REQUESTOR directly or use a CLIENT PROXY. The CLIENT PROXY is a local object within the client process that offers the same interface as the remote object. This interface is defined using an INTERFACE DESCRIPTION. Internally, the CLIENT PROXY uses the REQUESTOR to construct remote invocations.

The REQUESTOR on the client side uses a CLIENT REQUEST HANDLER to handle network communication. On the server side, the remote invocations are received by a SERVER REQUEST HANDLER. It handles message reception in a efficient and scalable way, and subsequently forwards invocations to the INVOKER, once the message has been received completely. The INVOKER dispatches remote invocations to the responsible remote object using the received invocation information.

The parameters passed between client and server are serialized and de-serialized using a MARSHALLER. If the client experiences technical problems in communicating with the server, or if the server has internal technical problems, this is returned to the CLIENT REQUEST HANDLER and signalled to the client using a REMOTING ERROR.



Requestor

A client needs to access one or more remote objects on a remote server.



Invocation of remote objects requires that operation parameters are collected and marshaled to a byte stream, since networks only allow to send byte streams. Further, a connection needs to be established and the request information must be sent to the target remote object. These tasks have to be performed for every remote object that the client accesses and can therefore become tedious for client developers.

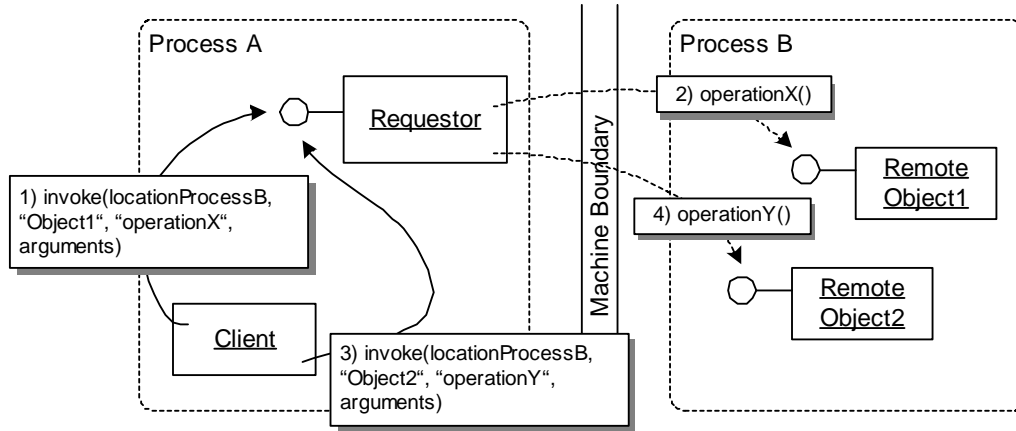
Clients have to perform many recurring steps for every single invocation. These tasks include marshalling of invocation information, network connection management, the actual transmission of remote invocations across the network, handling the result of an invocation, as well as error handling.

It is unsatisfying to require the client developer to remember and perform all these tasks over and over again. Further, optimizations on single cases are lost for the common case, where they might be applicable as well. Developers creating application software should be able to focus on the application logic, and not have to deal with these low-level details.

Therefore:

In the client application, use a REQUESTOR for accessing the remote object. The REQUESTOR is supplied with the ABSOLUTE OBJECT REFERENCE of the remote object, the operation name, and its arguments. It constructs a remote invocation from these parameters and sends the

invocation to the remote object. The REQUESTOR hides the details of the client side distributed object communication from clients.



To invoke an operation on a remote object, the client passes the data necessary to invoke the operation to the REQUESTOR who then sends this invocation across the network to the remote object. It handles the details of crossing the process and/or machine boundary. Note that the illustration does not explicitly show how return values are handled.



The REQUESTOR accepts parameters that the client passed to it. It then delegates the tasks of marshalling and de-marshalling of parameters to the MARSHALLER. Next, it starts sending the request and receiving the result using the CLIENT REQUEST HANDLER. If necessary, it triggers INVOCATION INTERCEPTORS.

Remote communication is inherently unreliable and the invocation process is more complex than in case of local invocations. New kinds of errors can result as a consequence. For communicating such exceptional issues to clients, REMOTING ERRORS are used. Client developers cannot assume that the remote object is reachable all the time, for instance, because of network delays, network failures, or server crashes. The REQUESTOR abstracts these remoting details for clients by raising REMOTING ERRORS if necessary.

The REQUESTOR is independent of the specific remote object details, such as the type of the remote object or the signature of the operation, as defined by the INTERFACE DESCRIPTION. It dynamically constructs invocations from the information received from the client. To ensure type safety, statically typed abstractions such as those provided by CLIENT PROXIES are used. Internally, they use the dynamic construction provided by the REQUESTOR. Since the remote object's interface can change without the CLIENT PROXIES taking notice, type safety problems are only minimized, but not completely avoided. Type mismatches and dispatch failures on the server side are communicated to clients by the REQUESTOR using REMOTING ERRORS.

The REQUESTOR can be deployed in various ways. For a specific client, there might be one REQUESTOR that handles invocations for all objects in all server applications. In this case invocations to the REQUESTOR need to be synchronized, for instance using the *Monitor Object* pattern [SSRB00]. Alternatively, the REQUESTOR can be instantiated more than once, and synchronization of common resources, such as connections, is handled internally. In either case the REQUESTOR needs to be supplied with the complete ABSOLUTE OBJECT REFERENCE - or all necessary information to construct it, as well as all the data describing an invocation. If a REQUESTOR is only dedicated to a specific remote object, you can pre-initialize that REQUESTOR, which saves resources at runtime.

Client Proxy

A REQUESTOR is provided by the distributed object middleware to access remote objects.



One of the primary goals of using remote objects is to support a programming model for accessing objects in distributed applications that is similar to accessing local objects. A REQUESTOR solves part of this problem by hiding many network details. However, using the REQUESTOR is cumbersome, since the methods to be invoked on the remote object, their arguments, as well as location and identification information for the remote object have to be passed by the client in a format defined by the REQUESTOR for each invocation. The REQUESTOR also does not provide static type checking, which further complicates client development.

The main purpose of a distributed object middleware is to ease development of distributed applications. Thus, developers should not be forced to give up their accustomed 'way of programming'. In the ideal case, they simply invoke operations on remote objects just as if they were local objects - taking into account the additional error modes described in the REQUESTOR pattern.

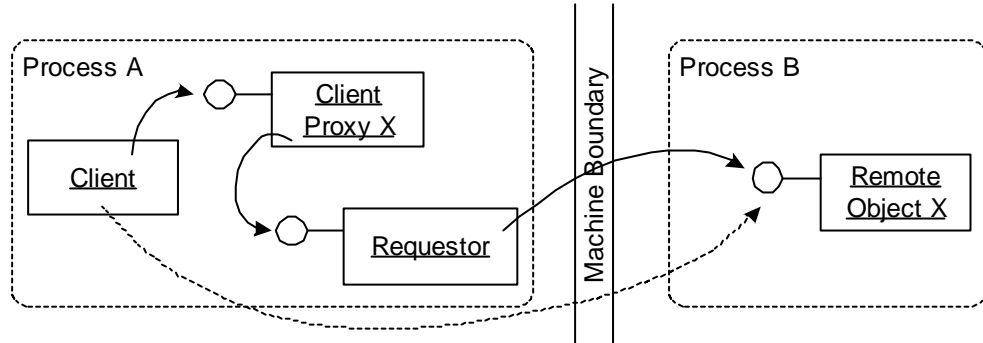
The advantage of the REQUESTOR of being generic and type-independent has to be paid with a complex interface, which makes it cumbersome for client developers to perform remote invocations.

Using only a REQUESTOR bears the risk that invocations might not get understood by the target remote object. Simple typing errors on the name of the operation or slight differences of the operation arguments can lead to REMOTING ERRORS since no compile-time type checking is possible.

Therefore:

Provide a CLIENT PROXY to the client developer that supports the same interface as the remote object. For remote invocations, clients only interact with the local CLIENT PROXY. The CLIENT PROXY translates the local invocation into parameters for the REQUESTOR, and triggers the invocation. The CLIENT PROXY receives the result from

the REQUESTOR, and hands it over to the client using an ordinary return value.



The client interacts with the CLIENT PROXY which supports the same interface as the remote object. The CLIENT PROXY uses the REQUESTOR as part of the distributed object middleware to construct and send the remote invocation.



The CLIENT PROXY uses a REQUESTOR to construct and send invocations across the network. As the CLIENT PROXY is specific to the type of a remote object, it is typically generated from the remote object's INTERFACE DESCRIPTION.

To be available on the client side, the CLIENT PROXY has to be deployed to the client somehow. In the simplest case, the source code of the CLIENT PROXY is compiled with the client application. This has the obvious drawback, that on every change of the CLIENT PROXY's implementation, the client would need to get recompiled as well. The client should not have to necessarily change because of any CLIENT PROXY changes.

Alternatively CLIENT PROXIES can be bound late for example during loading, linking, or at runtime. That is, the CLIENT PROXY implementation class is designed to be exchangeable in the client. The client uses a stable CLIENT PROXY interface, but the implementation of this interface is provided by the server application, which usually happens during startup of the client.

The distribution of CLIENT PROXIES can also be done as part of a LOOKUP process. This has the advantage that CLIENT PROXIES can be exchanged transparently. Sometimes this is necessary in cases where the CLIENT PROXY takes part in failover or load-balancing policies. On the downside, this approach incurs the liability of sending CLIENT PROXY implementations across the network. This can be avoided by downloading or sending only the INTERFACE DESCRIPTION to the client, and the client generates the late bound CLIENT PROXY from this interface at runtime.

In some rare cases, even the remote object interfaces change during runtime. Such changes can be handled on server side only, for instance in the INVOKER. However, if this is not possible and the client needs to align with the interface change, the use of CLIENT PROXY should be avoided, and clients should use the REQUESTOR directly to construct remote invocations dynamically. To help the client do this, it needs runtime access to the INTERFACE DESCRIPTION, for example by looking it up in an interface repository, by querying the remote object, or by using reflection. Of course, the client's application logic has to be changed in any case if it wants to make use of the new interface provided by the remote object.

By directly providing the remote object's interface and by representing a specific remote object directly in the client process, a CLIENT PROXY is typically easier to use than a REQUESTOR, especially for unexperienced developers. Its look-and-feel is more aligned with non-distributed applications, it provide a higher level of transparency.

However, a consequence is that a CLIENT PROXY is less flexible than a REQUESTOR. Because a CLIENT PROXY uses an REQUESTOR internally, the solution is (slightly) slower and consumes more memory than a pure REQUESTOR solution. After all, we need a dedicated CLIENT PROXY instance per each remote object that we want to talk to.

The CLIENT PROXY applies the *Proxy* pattern from GoF [GHJV95] and POSA1 [BMR+96] for hiding remote communication. POSA1 introduces a variant called *Remoting Proxy*. The CLIENT PROXY pattern is more specific than *Remoting Proxy* as it hides the REQUESTOR but does not handle the remote communication itself.

Invoker

A REQUESTOR sends a remote invocation for a remote object to a server.



When a client sends invocation data across the machine boundary to the server side, somehow the targeted remote object has to be reached. The simplest solution is to let every remote object be addressed over the network directly. But this solution does not work for large numbers of remote objects, there may not be enough network endpoints for all the remote objects. Also, the remote object would have to deal with handling network connections, receiving and demarshalling messages, etc. This is cumbersome and too complex.

If a large number of remote objects were addressed directly over the network, the system would quickly run out of resources, such as connection ports, handles, or other system resources, such as the threads used to listen for incoming connections on the communication endpoints.

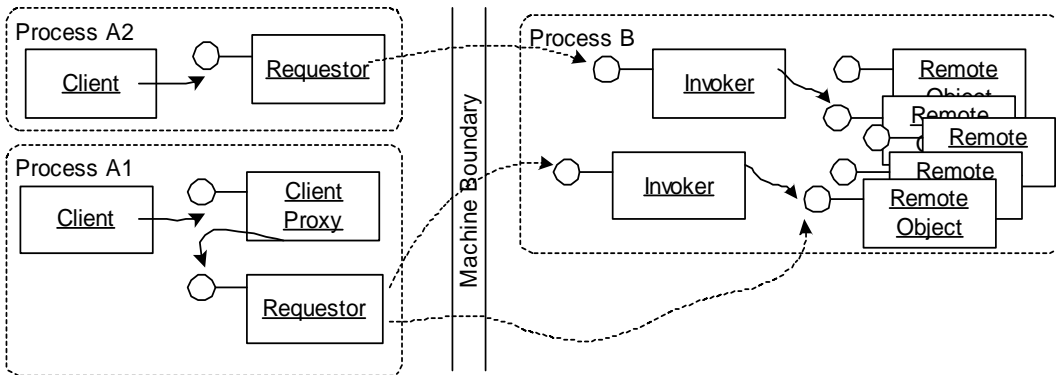
Even if there is only a limited number of associations between clients and remote objects, there are still issues. Client developers would have to be aware of all network addresses of the required remote objects. The mapping of addresses to remote objects will be tedious to establish and maintain.

Instead, a client should only have to provide the necessary information required to select the appropriate remote object and have the server application deal with dispatching and invoking that object.

The remote object implementation should be independent of any communication details, such as listening for incoming invocation messages as well as demarshalling/marshalling of invocation parameters. Keeping this responsibility in the remote object itself is not good practice of separation of concerns. Additionally, in some cases the remote object might not be available all the time and is only activated on demand, which requires someone to accept invocations and trigger the (re-)activation of the target remote object.

Therefore:

Provide an INVOKER that accepts client invocations from REQUESTORS. REQUESTORS send requests across the network, containing the ID of the remote object, operation name, operation parameters, as well as additional contextual information. The INVOKER reads the request and demarshalls it to obtain the OBJECT ID and the name of the operation. It then dispatches the invocation with demarshalled invocation parameters to the targeted remote object. That is, it looks up the correct local object and its operation implementation as described by the remote invocation, and invokes it.



The REQUESTOR sends a request to the INVOKER. The INVOKER demarshalls the request and dispatches it to the remote object implementation. Results are returned as responses in the reverse direction.



The INVOKER is located on the server side within the process of the server application. Depending on the configured CONFIGURATION GROUPS, one or several INVOKER instances exist, since both concepts are closely related. An associated SERVER REQUEST HANDLER handles lower-level network communication. This includes receiving remote messages, management of threads, pooling of connections, as well as sending of responses.

When a request message has been received completely by the SERVER REQUEST HANDLER, it hands it over to the INVOKER. In case of multiple INVOKERS, the message must contain enough information to allow the

SERVER REQUEST HANDLER to contact the correct INVOKER. The message therefore must contain information about the INVOKER, such as a INVOKER name or ID. If necessary, the INVOKER ID must be part of the ABSOLUTE OBJECT REFERENCE, as explained later in this book. Note that in order for the SERVER REQUEST HANDLER to be able to hand the message to the correct INVOKER, it has to partially demarshal the request message.

If not already done, the INVOKER demarshalls the request message further to identify the target remote object, the operation name and the invocation parameters. The actual de-marshalling is handled by a separate concept, the MARSHALLER.

In case a target remote object cannot be found by the INVOKER, the INVOKER can delegate dispatching to a LOCATION FORWARDER. The LOCATION FORWARDER might be able to delegate to a remote object in another server application. Such approaches are used in cases when remote objects have been moved, or when load is balanced between multiple remote objects.

More than one servant might be used to implement one remote object at runtime. For instance, when using the POOLING pattern, a remote object is realized by multiple servants managed in a pool. The INVOKER has to select one of these servant or trigger the selection of a servant. Also, the servant of a remote object might have been temporarily evicted to reduce the resource consumption. Here, the INVOKER requests the LIFECYCLE MANAGER to make the remote object's servant available again.

The INVOKER can trigger functionality, such as lifecycle management or other orthogonal tasks, by using INVOCATION INTERCEPTORS. INVOCATION INTERCEPTORS can generally be used to extend the dispatching in the SERVER REQUEST HANDLER and INVOKER with custom functionality.

Determining and invoking the target object and operation can be performed dynamically or statically:

- *Static dispatch.* Static dispatching is done in so called server stubs, sometimes also called skeletons. They are part of the INVOKER and are generated from INTERFACE DESCRIPTIONS. For each remote object type a separate server stub is generated. Thus the INVOKER *knows* the remote object type, including operations, and operation

signatures in advance. With this type information, the INVOKER can directly invoke the target operation, and it does not have to find the operation implementation dynamically.

- *Dynamic dispatch.* When it is undesired or not possible to use server stubs, for example because remote objects interfaces are not known at compile time, dynamic dispatch is advisable. The INVOKER has to decide at runtime which operation of which remote object be invoked. From the information in the demarshalled request message the INVOKER finds the corresponding remote object and operation implementation in the local process, for instance, using runtime dispatch mechanisms, such as *Reflection* [Mae87, BMR+96] or a dynamic lookup in a table. Using dynamic dispatch an invocation can possibly contain a non-existing operation or use a wrong signature. In this case a special REMOTING ERROR has to be returned to the REQUESTOR on the client. Finally, the INVOKER invokes the operation on the target remote object. This form of dispatch is called dynamic dispatch, as the INVOKER dispatches each invocation by dynamically resolving the actual operation at runtime (for more details of this implementation variant see the pattern *Message Redirector* [GNZ01]).

Comparing dynamic dispatch with static dispatch, static dispatch is typically faster as the overhead of looking up operation implementations at runtime is avoided, whereas dynamic dispatch is more flexible. Dynamic dispatch is—in contrast to static dispatch—not type-safe; that’s the reason why it introduces a new kind of REMOTING ERROR.

The INVOKER on the server side and the REQUESTOR on the client side have to be compatible with respect to the messages they exchange. The REQUESTOR has to put all the information required by the INVOKER into an request message, and the INVOKER has to send well-formed response messages.

For extension and integration of application-specific functionality during dispatching, INVOCATION INTERCEPTORS in combination with INVOCATION CONTEXTS can be used. To configure dispatching constraints the remote objects in a server application are typically partitioned into CONFIGURATION GROUPS, which encapsulate behavior specific to a group of objects. In cases where the remote objects have non-trivial lifecycles, the use of a LIFECYCLE MANAGER is advisable.

Using an INVOKER, instead of direct network connections to remote objects, reduces the number of required network addressable entities. It also allows the SERVER REQUEST HANDLER to efficiently share connections to servers hosting several remote objects.

Client Request Handler

The REQUESTOR has to send requests to and receive responses from the network.



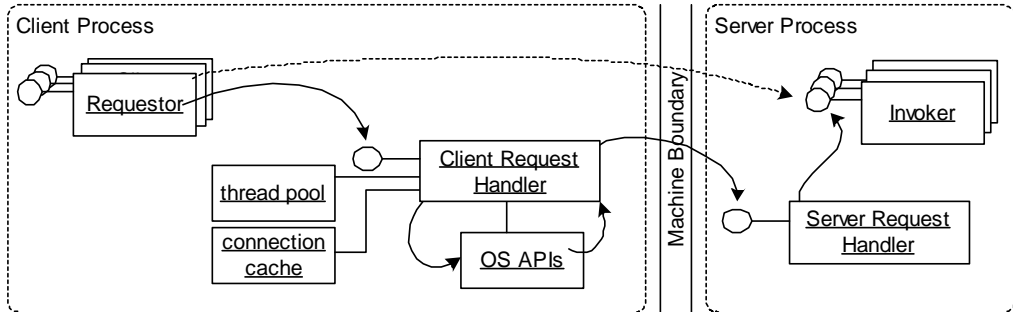
To send requests from the client to the server application, several tasks have to be performed: connection establishment and configuration, result handling, timeout handling, and error detection. In case of timeouts or errors the REQUESTOR, and subsequently the CLIENT PROXY, have to be informed. Client-side connection management, threading, and result dispatching need to be managed in a coordinated and optimized fashion.

CLIENT PROXIES support a proper abstraction of remote object access, REQUESTORS support proper invocation construction, but connection management, threading, and result dispatching are not handled efficiently by these patterns. For simple, single-threaded clients with only a few requests, these networking issues can be handled on a per CLIENT PROXY or per REQUESTOR basis. But in more complex clients, it is possible that a large number of requests are to be sent simultaneously. In such cases, the network access needs to be optimized across multiple CLIENT PROXIES and REQUESTORS. For example, network connections can be kept open and subsequently reused for additional requests to the same server application.

Therefore:

Provide a common CLIENT REQUEST HANDLER for all REQUESTORS within a client. The CLIENT REQUEST HANDLER is responsible for opening and closing the network connections to server applications, sending of requests, receiving of responses, and dispatching them back to the appropriate REQUESTOR. Additionally, the CLIENT

REQUEST HANDLER copes with timeouts, threading issues, and invocation errors.



When a client issues a request using a REQUESTOR, the CLIENT REQUEST HANDLER establishes the connection, sends the request, receives the response, and returns it to the REQUESTOR.

* * *

The CLIENT REQUEST HANDLER is responsible for handling network communication inside client applications. The internally used *Connector* [SSRB00] establishes and configures connections to the remote server. A connection handle is used to identify connections and associate a responsible handler. Typically a connection to a particular server and server port can be shared with other invocations to the same server application. The actual task of sending invocation data, especially in the context of several available transport protocols, is done using a PROTOCOL PLUG-IN.

It is the job of the CLIENT REQUEST HANDLER to ensure scalability. That is, it has to be designed in such a way that it handles concurrent requests efficiently. To do this, the *Reactor* [SSRB00] pattern is used to demultiplex and dispatch response messages. A reactor uses the connection handle to notify the responsible handler about available results. There are different ways how the result is handled in the client:

- In the case of synchronous invocations, after an invocation is sent across the network, the CLIENT REQUEST HANDLER has to wait for the result of the invocation. That is, it blocks until the result has arrived.

- For asynchronous invocations the REQUESTOR and CLIENT REQUEST HANDLER have to follow one of the client asynchrony patterns, described in the *Invocation Asynchrony Patterns* chapter. In the case of RESULT CALLBACK or POLL OBJECT it dispatches the result to a POLL OBJECT or callback object respectively, instead of handing the result back to the client invocation on the callstack.

When timeouts have to be supported, the CLIENT REQUEST HANDLER informs the REQUESTOR about timeout events. For this purpose, the CLIENT REQUEST HANDLER registers for a timeout when the invocation is sent to the remote object. If the reply does not arrive within the timeout period, the CLIENT REQUEST HANDLER receives the timeout event and informs the REQUESTOR. Depending on the configuration, it might also retry sending an invocation several times before raising the REMOTING ERROR. The CLIENT REQUEST HANDLER must also detect other error conditions such as an unavailable network or server application. In this case, it has to raise a REMOTING ERROR and forward it to the REQUESTOR.

For thread management the CLIENT REQUEST HANDLER makes use of the same patterns as the SERVER REQUEST HANDLER: *Half-sync/Half-async* [SSRB00] and/or *Leader/Followers* [SSRB00]. Further, *Caching* [KJ04] can be used to reuse connections to the same server application. *Pooling* [KJ04] can be used to create thread pools to reduce the overhead of creating threads over and over again.

For many tasks, CLIENT and SERVER REQUEST HANDLER have to work in concert, especially for extensions of request handling such as INVOCATION INTERCEPTORS, INVOCATION CONTEXTS, PROTOCOL PLUG-INS, or other add-on services.

The CLIENT REQUEST HANDLER is typically shared between multiple REQUESTORS. CLIENT REQUEST HANDLERS hide connection and message handling complexity from the REQUESTORS. For very simple clients the indirection and required resource management complexity in the CLIENT REQUEST HANDLER might be an overhead in some scenarios, for instance, if a high invocation performance is required as well. In such cases it might make sense to avoid this overhead by implementing REQUESTORS that directly connect to the network.

The role of the CLIENT REQUEST HANDLER is analogous to the *Forwarder* in the *Forwarder-Receiver* pattern in POSA1 [BMR+96].

Server Request Handler

You are providing remote objects in a server application, and INVOKERS are used for message dispatching.



Before a request can be dispatched by an INVOKER, the server application has to receive the request message from the network. Managing communication channels efficiently and effectively is essential, since typically, many requests may have to be handled, possibly even concurrently. Network communication needs to be managed in a coordinated and optimized way.

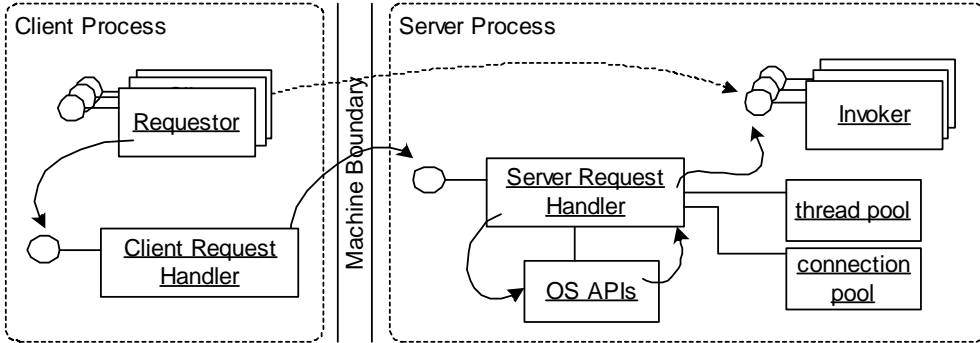
Simple implementation of the server side parts of a distributed object system would have each INVOKER listen to the network and receive request messages. But this solution does not scale if multiple INVOKERS are used. In case a client uses remote objects from several of those INVOKERS, separate resources, for example connections, would be required. Further, it should be transparent to the client how many INVOKERS the server application uses.

Managing request and response messages can be complex, as it involves the management of several — typically scarce — system resources, such as connections, threads, synchronization primitives, and memory. For reasons of performance, scalability, and stability, it is crucial that these aspects are handled effectively and efficiently.

Therefore:

Provide a SERVER REQUEST HANDLER that deals with all communication issues of a server application. Let the SERVER REQUEST HANDLER receive messages from the network, combine the message fragments to complete messages, and dispatch the messages to the correct INVOKER for further processing. The SERVER REQUEST

HANDLER will manage all the required resources, such as connections and threads.



Incoming messages are received by the SERVER REQUEST HANDLER on server side using suitable operating system APIs. The messages are then dispatched to the responsible INVOKERS using threads acquired from a thread pool.



The SERVER REQUEST HANDLER deals with messages and how they are sent and received over the network, whereas the INVOKER sitting "on top of it" deals with requests and how they are dispatched and invoked. When multiple INVOKERS are present, the SERVER REQUEST HANDLER demarshalls the message at least to the point where it can decide which INVOKER to dispatch the message to. The rest of the message is demarshalled at latest by the INVOKER.

The SERVER REQUEST HANDLER has to be designed in such a way that it handles concurrent requests efficiently. In many cases, efficient request handling requires a number of concurrent instances that can handle requests simultaneously. Several patterns in POSA2 [SSRB00] support such efficient SERVER REQUEST HANDLER designs.

The SERVER REQUEST HANDLER listens for connection requests on a network port, and, if a connection is established, for incoming messages. A *Reactor* [SSRB00] is typically used for demultiplexing and dispatching those connection requests and messages to the SERVER REQUEST HANDLER, which handles them accordingly.

Connection management is typically implemented by the SERVER REQUEST HANDLER using an *Acceptor* [SSRB00]. Connection requests are accepted asynchronously, and connection handles, identifying individual connections, are created.

For integrated thread and connection management the SERVER REQUEST HANDLER uses the patterns *Half-sync/Half-async* [SSRB00], which describes how to split common functionality between two objects, or *Leader/Followers* [SSRB00], which describes how efficiently manage threads and their access to shared resources.

To optimize resource allocation for connections, the connections can be shared in a pool, as described by the *Pooling* [KJ04] pattern. It depends on the number of connections, whether one thread per connection or an connection-independent thread pool performs better. Some SERVER REQUEST HANDLERS allow for connection establishment policies, typically implemented as *Strategies* [GHJV95]. Such a strategy could allow a server to start up using a thread per connection strategy and then transition to connection-independent thread pooling in case many connections are used. In cases where request priorities need to be obeyed, different threads might have different SERVER REQUEST HANDLERS associated.

In some application scenarios it can be expected that a certain client communicates with remote objects in the same server application again and again, in other scenarios this might not be the case. Thus there are different strategies for connection establishment. For instance, a new connection can be opened and closed for each message sent across the network. Or, alternatively, the connection can be held open for a certain amount of time and then used again for subsequent invocations. The latter alternative has the advantage that it avoids the overhead of establishing and destroying connections for repeated client requests. However, connections that are held open consume resources. The connection establishment strategy of the server application is implemented by the SERVER REQUEST HANDLER.

For the actual network communication the SERVER REQUEST HANDLER uses PROTOCOL PLUG-INS. Since it is required that the CLIENT and SERVER REQUEST HANDLER are aligned and work in concert, they have to use compatible PROTOCOL PLUG-INS.

An important consequence of using a SERVER REQUEST HANDLER is that low-level networking issues are centralized in a single system component. Connection pools and thread pools allow it to be highly concurrent, avoiding the risk of bottlenecks.

Note that there are some application areas where multiple SERVER REQUEST HANDLER instances are useful, such as in the case of the direct binding of remote objects to network ports. For instance, if there are only a few remote objects but those require a high performance, the result are different communication channel configurations and concurrency settings per remote object. It is hard to cope with this scenario in a centralized fashion by only one SERVER REQUEST HANDLER.

The role of the SERVER REQUEST HANDLER pattern is analogous to the *Receiver* in the *Forwarder-Receiver* pattern in POSA1 [BMR+96].

Marshaller

Request and response messages have to be transported over the network between REQUESTOR and INVOKER.



For remote invocations to happen, invocation information has to be transported over the network. The data to describe invocations consists of the target remote object's OBJECT ID, the operation name, the parameters, the return value, and possibly other INVOCATION CONTEXT information. For transporting this information over the network, only byte streams are suitable as a data format.

For sending invocation data across the network there has to be some concept for transforming invocations of remote operations into a byte stream. For simple data, such as an integer value, this is trivial, but there are some additional cases that make this task complicated, for instance:

- Complex, user-defined types typically have references to other instances, possibly forming a complex hierarchy of objects. Such hierarchies might even contain multiple references to the same instance, and in such cases, logical identities have to be preserved after the transport to the server.
- Local objects might have to be transmitted with an invocation. The object identity and attributes of these local objects need to be transported across the network.

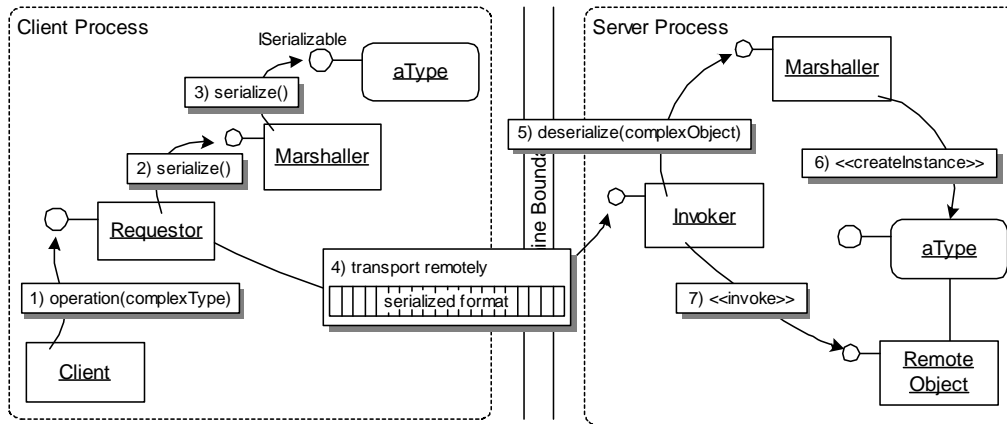
Generating and interpreting a byte stream representation should not require additional programming efforts per instance, but should only be defined once per type. The generation and interpretation of transport formats as well as the data formats should be extensible for developers to allow for customization and optimization.

A simple solution for transporting an object or data structure across the network is to send the memory representation of it. However, if different platforms have to be supported this might lead to platform incompatibilities such as Big Endian versus Little Endian byte order. Similar problems arise if different programming languages are

involved, since these often use different in-memory representations of the data types, or even support completely different data types.

Therefore:

Require each non-primitive type used within remote object invocations to be serializable into a transport format that can be transported over a network as a byte stream. Use compatible MARSHALLERS on client and server side that serialize invocation information. Depending on the environment, the serialization might be provided by the data types themselves. References to remote objects are serialized as ABSOLUTE OBJECT REFERENCES.



The client invokes an operation with a local object as parameter. The REQUESTOR uses the responsible MARSHALLER to serialize the object into a byte stream. The serialized object is then transported along with other invocation information across the network as a byte stream. On the server side, the stream is de-serialized by the MARSHALLER and an instance of the object is recreated before operation invocation. Finally, the remote object is invoked using the recreated object as a parameter.



A MARSHALLER converts remote invocations into byte streams. The MARSHALLER provides a generic mechanism that is not specific for one particular remote object type.

The REQUESTOR, INVOKER, and REQUEST HANDLERS use the MARSHALLER to retrieve the invocation information contained in the message byte stream. For complex data types the MARSHALLER recursively parses their type hierarchy. Object identities are handled as special data type, but are marshalled similarly to complex data types.

References to other remote objects are translated into ABSOLUTE OBJECT REFERENCES.

Local objects containing many data elements used by the invoked remote object, should not be referenced remotely, but marshalled by value. To transport such a data type across the network, a generic transport format is required. For this purpose, a MARSHALLER uses the *Serializer* pattern [RSB+98]. The serialization of a complex type can be done in multiple ways:

- The programming language provides a generic, built-in facility for serialization. This is often the case in interpreted languages which can use *Reflection* [Mae87, BMR+96] to introspect the type's structure, such as Java or .NET.
- Tools generate serialization code directly from the INTERFACE DESCRIPTION, assuming that the structure of such types is expressed in the INTERFACE DESCRIPTION (for example structs in CORBA).
- Developer's have to provide serialization functionality. In this case, the developer usually has to implement a suitable interface that declares operations for serialization and de-serialization.

The concrete format of the data inside a byte stream depends on the distributed object middleware used. At the bottom line, everything is a byte stream as soon as it is transported across the network. To represent complex data structures, it is advisable to use a structured format such as XML, CDR (CORBA's marshalling format), or ASN.1 (a marshalling format used mainly in telecom systems, see [Dub01]).

The distributed object middleware might support a hook to let developers provide a custom MARSHALLER. Reasons for supporting custom MARSHALLERS are that generic marshalling may be too complex or inefficient for marshalling complex data structure graphs. A generic marshalling format can never be optimal for all data structures or scenarios. Depending on the use case, some serialization formats are

better than others. XML is human-readable, but too inefficient for some domains, such as the embedded systems domain. CDR is a binary format and standardized, which means many tools are available for its usage. For real-time and embedded applications it is often further optimized. Other binary formats, are either similar to CDR, or specifically optimized for compactness, reliability, or other non-functional requirements.

Exchanging a MARSHALLER is not necessarily transparent for the applications on top of the distributed object middleware. A custom MARSHALLER might, for instance, transport all attributes of an object, or it might only transport the public ones, or it might just ignore those it cannot serialize instead of throwing an exception.

If the MASHALLER pattern has to serialize and de-serialize complete objects, the pattern *Serializer* [RSB+98] describes generically how efficiently stream objects into data structures. The MASHALLER uses *Serializers* for object parameters and results that have to be sent across the network in call-by-value style.

Interface Description

A client wants to invoke an operation of a remote object using a CLIENT PROXY.



The interfaces of a CLIENT PROXY and remote object need to be aligned to ensure that an INVOKER can properly dispatch invocations. Also, to ensure that messages arrive properly at the INVOKER, the marshalling and de-marshalling needs to be aligned. Client developers need to know the interfaces of the remote objects the client application may use.

A CLIENT PROXY, used as a local representative of a remote object, needs to expose the same interface as provided by the remote object. The CLIENT PROXY is responsible for ensuring that the remote object is used correctly regarding operations and their signature.

On the server side, the INVOKER dispatches the invoked operation of the remote object. For static dispatch, the INVOKER has to know the operations and their signature before an invocation request arrives. In the case of dynamic dispatch this information will be retrieved at run-time and is therefore not as important.

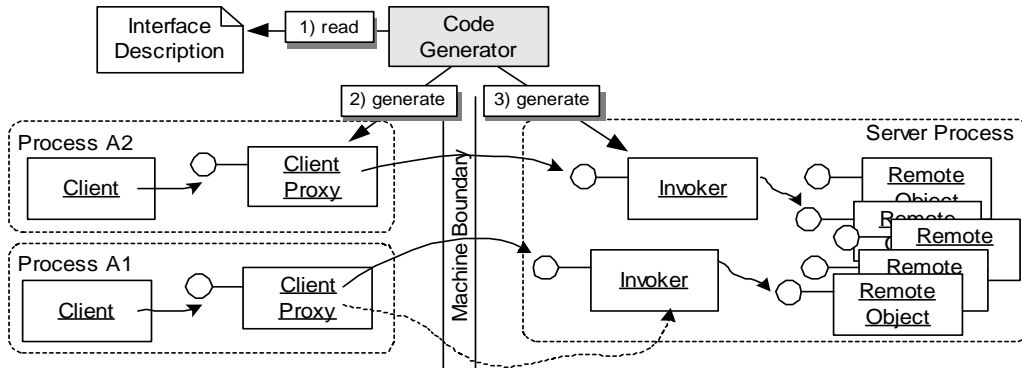
To summarize, CLIENT PROXY and INVOKER have to work together to ensure that no violation of the remote object's interfaces occurs. Client developers and remote object developers should not have to deal with propagating and ensuring remote object interfaces manually. Instead, the distributed object middleware should provide suitable means for automating these issues as much as feasible.

In addition to these problems, the client and server application might be written in different programming languages. The two languages might expose different data types, operation types, invocation styles, and other interface elements. Developers should not have to deal with converting these elements from one language context to another.

Therefore:

Provide an INTERFACE DESCRIPTION in which you describe the interface of remote objects. The INTERFACE DESCRIPTION serve as the

contract between CLIENT PROXY and INVOKER. CLIENT PROXY and INVOKER use either code generation or runtime configuration techniques to adhere to that contract.



Using an INTERFACE DESCRIPTION in a separate file, a code generator generates code for CLIENT PROXY and INVOKER. The CLIENT PROXY used by the client adheres to the same contract as the INVOKER which dispatches the invocation to the remote object. This way, details of ensuring type safety are hidden from client and remote object developers. All they have to care about is the INTERFACE DESCRIPTION and their implementation according to that interface.

* * *

Usually an INTERFACE DESCRIPTION contains interface specifications including their operations and signatures, as well as the definition of user-defined complex types. The INTERFACE DESCRIPTION itself can exist in various forms:

- *Interface description language:* The INTERFACE DESCRIPTION is separated from the program text, for instance, in a file provided by the remote object developer. An interface description language defines the syntax and semantics of the declarations. A code generator generates the remote object type specific parts of CLIENT PROXY and INVOKER. That is, interface violations can be automatically detected when compiling client and CLIENT PROXY. Note that this will only work if mechanisms exist which ensure that the isolated INTERFACE

DESCRIPTION really conforms to the latest version of the remote object interface.

- *Interface repository:* The INTERFACE DESCRIPTION is provided at runtime to clients using an interface repository exposed by the server application or some other entity. The interface repository is used for building CLIENT PROXIES at compile time, just as in the interface description language variant above. In addition, an interface repository can be used for dynamically constructed invocations to provide remotely accessible interface information at runtime. The INVOKER has similar options.
- *Reflective interfaces:* The INTERFACE DESCRIPTION is provided by means of reflection [Mae87] either offered through the server side programming language or using the *Reflection* pattern [BMR+96]. For clients to make use of reflection the server application has to provide some means to query these information remotely.

Note that the different variants of INTERFACE DESCRIPTION correspond to the implementation variants of CLIENT PROXY and INVOKER. Static variants of CLIENT PROXY and INVOKER use code generation techniques and thus primarily use interface description languages directly. Dynamic variants require INTERFACE DESCRIPTIONS either on client or on server side at runtime, thus reflective interfaces or interface repositories are used. In many distributed object middleware solutions more than one INTERFACE DESCRIPTION variant is supported, as more than one variant of REQUESTOR, CLIENT PROXY, and/or INVOKER are supported. For example, interface repositories might derive their information from interfaces defined using an interface definition language.

In distributed object middleware that supports more than one programming language, INTERFACE DESCRIPTIONS must be given in a programming-language independent syntax. This requires tools that translate the INTERFACE DESCRIPTION to different languages. INTERFACE DESCRIPTIONS are an important means to building interoperable distributed systems. The distributed object middleware provides a language binding in addition to the INTERFACE DESCRIPTIONS to define generic conversion rules to/from a programming language.

INTERFACE DESCRIPTIONS separate interfaces from implementations. Thus the software engineering principle of *separation of concerns* is supported, as well as exchangeability of implementations. That means

clients can rely on stable interfaces, while remote object implementations can be changed. However, interface changes cannot be avoided in all cases, especially, in a distributed setting, where server application developers have no control over client code (and deployment of it). The research community still searches for ways to cope with those interface changes. The *Extension Interface* pattern [SSRB00] is one possible solution.

The CLIENT PROXY typically represents the same interface as described by the INTERFACE DESCRIPTION, though in some cases there might be differences. Firstly, the operations may have additional parameters for technical reasons, such as explicit context passing. Alternatively, the CLIENT PROXY might provide additional operations, for example for lifecycle management. Lastly, the client proxy may act as an *Adapter* [GHJV95]: For instance, when the INTERFACE DESCRIPTION is updated, a CLIENT PROXY may represent the “old” interface as it was before the interface change and translate invocations, if at all possible.

Note that in some cases, using REQUESTORS directly without an INTERFACE DESCRIPTION might be an option. Consider, for instance, you receive the invocation or location information (such as remote object type, OBJECT ID, method name, or host name and port) at runtime from a server and the client cannot be stopped. Performing the invocation and handling the REMOTING ERROR in cases of interface violations might be the much simpler solution than, for instance, compiling an CLIENT PROXY from an INTERFACE DESCRIPTION on the fly.

Remoting Error

Remote communication between clients and remote objects is inherently unreliable.



Although it is desirable that accessing remote objects is transparent for clients, actual implementations of distributed object middleware can never completely achieve this goal due to the inherent unreliability of communication networks. Apart from errors in the remote object itself, new kinds of errors can occur when communicating across machine boundaries. Examples are network failures, server crashes, or unavailable remote objects. Clients need cope with such errors.

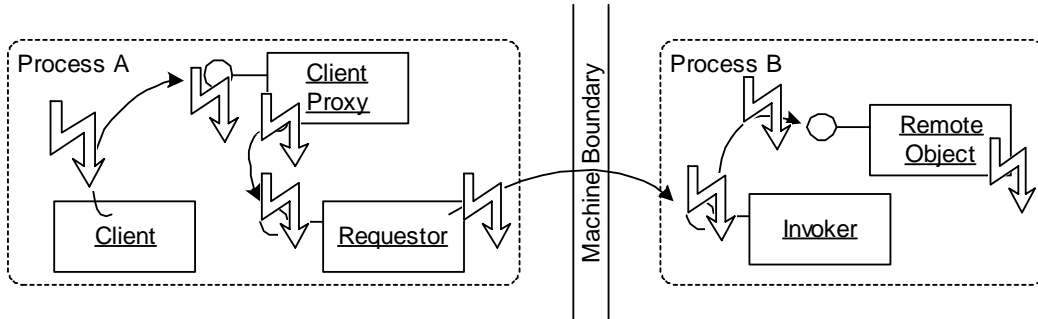
In addition to errors occurring within the remote object as a consequence of errors in the business logic, such as an attempt to transfer money from an empty account, other kinds of errors are a consequence of the distribution of client and remote object in separate processes and possibly also machines. Clients and servers fail typically independent of each other, since they run in separate processes. In addition, the network connection between them can fail, or provide only intermittent service.

The client should be able to distinguish between errors that are due to the use of a distributed infrastructure and errors that are due to the application logic inside remote objects.

Therefore:

Detect and propagate errors inside the distributed object middleware. Distinguish between errors that arise from distribution and remote communication from errors that arise from the application logic inside the remote objects. Clients will handle the types of REMOTING ERRORS differently. INVOKER, REQUESTOR, and the

REQUEST HANDLERS detect REMOTING ERRORS that are due to communication and dispatching problems.



Problems that occurred during a remote invocation are reported via the REQUESTOR as REMOTING ERROR to the CLIENT PROXY or directly to client.



The client has to be able to differentiate critical REMOTING ERRORS, such as server crashes or network outages, from less critical errors, such as “object not found” errors. For instance, as a reaction to a unavailable server, the client including the involved REQUESTOR and CLIENT REQUEST HANDLER, might have to cleanup resources related to the remote objects on that server application. If only the remote object is unavailable, but the server is generally available, the client might have to verify its LEASE for the remote object or access a different remote object.

REMOTING ERRORS detected inside the server application need to be transported back to the client as the result of a remote method invocation. INVOKER, REQUESTOR, and/or CLIENT PROXY will forward the error to the client. Other errors, such as timeouts or an unreachable server application, can directly be detected by the client infrastructure itself. Again, REQUESTOR and/or CLIENT PROXY will forward the error to the client.

INVOKERS and SERVER REQUEST HANDLERS send back special response messages in the case of a REMOTING ERROR. Those messages should include a reason or at least a guess on what might have gone wrong. If REMOTING ERRORS should not influence regular communication, sepa-

rate communication channels dedicated to error reporting may be used.

Note that it depends on the programming language which mechanisms are employed to report a REMOTING ERROR to the actual client within the client process. In most modern programming languages, exceptions are used. To distinguish REMOTING ERRORS from application errors, exceptions reporting REMOTING ERROR typically inherit from a system-provided exception base class such as *RemotingError*. Specific kinds of REMOTING ERRORS are implemented using various subclasses. Alternatively, such as DCOM, special return values (integers in a specific range) are used to signal REMOTING ERRORS. The language also determines whether the client can be forced to handle the exception, such as in Java, or whether it is left to the client to provide a catch clause, such as in C++ and C#. In the latter case, when no catch-clause is provided, the client developer of course risks undefined application behavior.

The reported error must contain enough information for the client to take sensible action in order to handle the error and possibly to recover from it. If the REMOTING ERROR is caused by a problem in the local network stack, there is usually not much the client can do. But if there is a problem on the server side, the client may find another compatible remote object or server application using LOOKUP, and try again.

Not all REMOTING ERRORS raised by the server side need to be reported to the client application; the client-side distributed object middleware may handle some of these automatically, depending on the configuration of the system. For example, a REQUESTOR can be configured to contact a different server application in case the original one is not available, or the CLIENT REQUEST HANDLER can automatically retry sending messages to handle short, temporary network outages transparently for the client. A MESSAGE QUEUE can be used to temporarily store messages, and send them later in time – when the server is available again.

Besides recovery from an error, another important goal of sending detailed REMOTING ERRORS is debugging of distributed applications. The client and server application should log REMOTING ERRORS in the order they occurred to help in the localization of bugs.

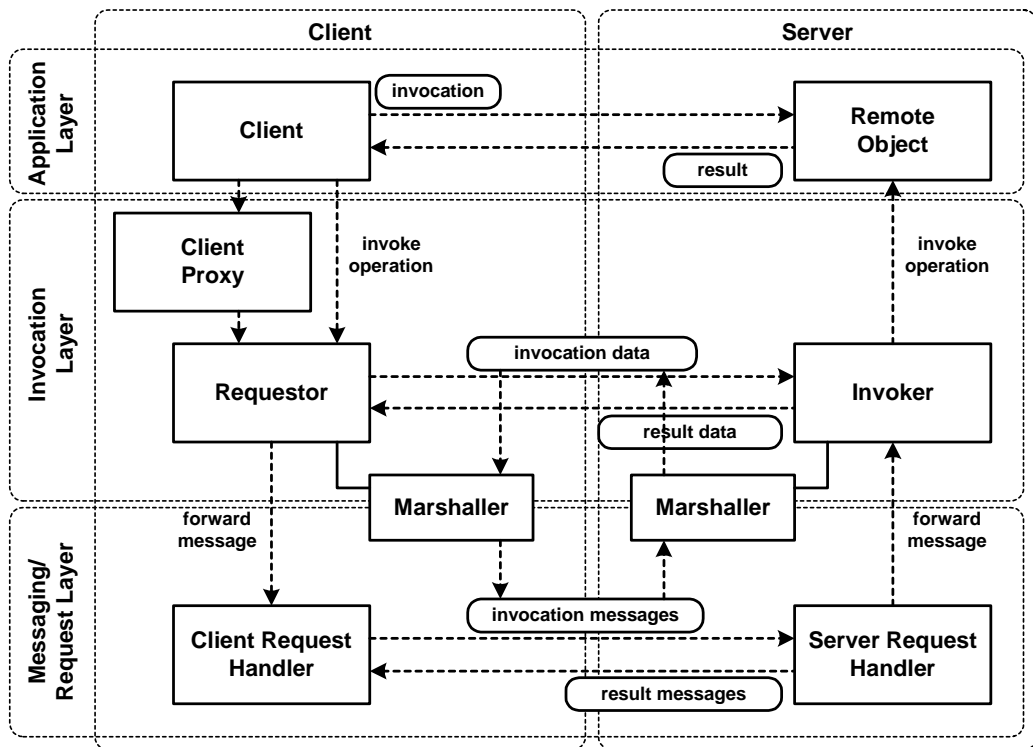
By making server and network errors visible to the client, the REMOTING ERROR pattern addresses the inherent unreliability of network commu-

nication in distributed environments. A liability of using the pattern is the necessity for programmers to deal with the new kinds of errors, when developing client applications. However, ignoring these errors is not an option either - the REMOTING ERROR pattern gives the client at least a chance to recover from the error and/or perform a meaningful error logging.

Interactions among the Patterns

The patterns illustrated in this chapter form the basic building blocks of a distributed object middleware. Note that it is not necessary that each pattern actually forms a component, or module, in the software architecture. The patterns should rather be seen as roles. A software component can play multiple roles at the same times - that is, it can participate in a number of patterns. Or, a pattern implementation can be spread over several software components.

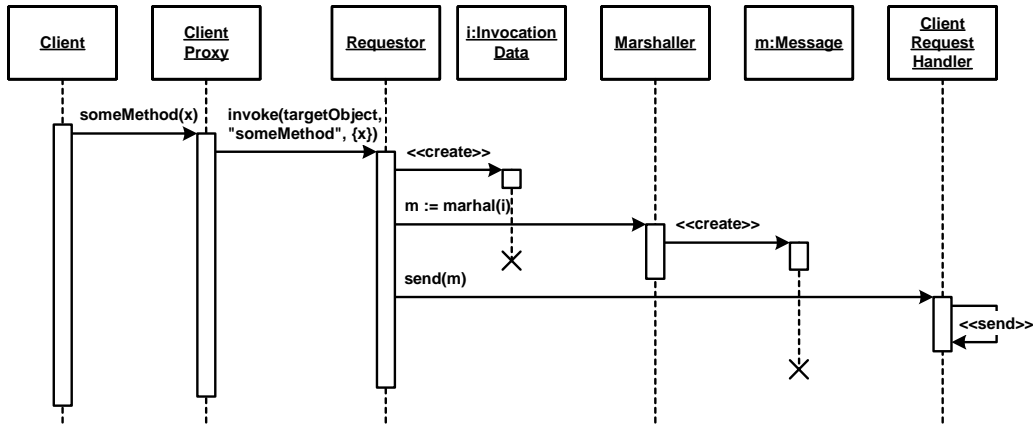
The basic remoting patterns form a logical *Layers* architecture, which is symmetrical between client and server. The following illustration shows this.



The sequence diagrams that follow illustrate the dynamics of some of the interactions among the patterns. Note that for reasons of brevity, we do not consider all technical implementation details in those

diagrams. That is, only the necessary operation parameters are shown, helper objects are omitted, and so on.

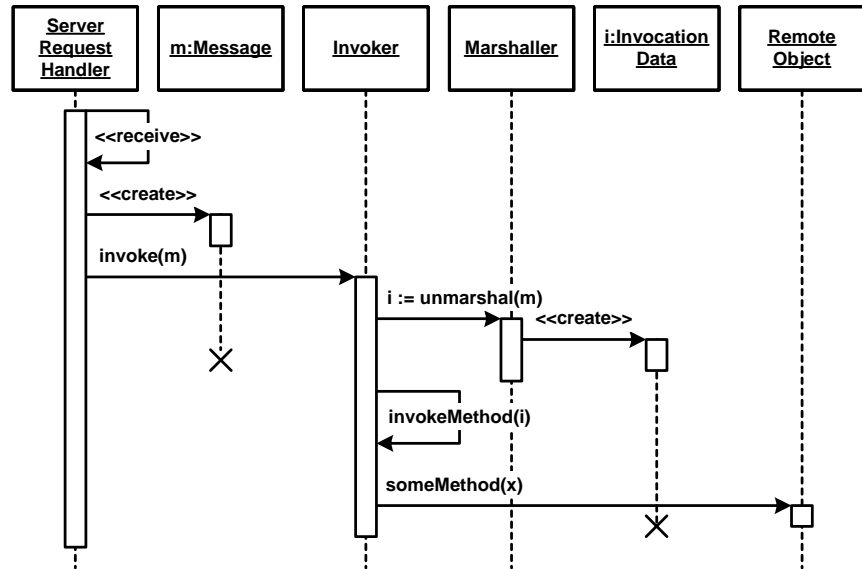
The following sequence diagram shows a basic invocation sequence on the client side. A CLIENT PROXY builds up the invocation using the



REQUESTOR. All invocation data is stored in an invocation data object so that the invocation data can be passed around between the participants of the invocation process. The REQUESTOR uses the MARSHALLER to build up a *Message* object, which is sent across the network by the CLIENT REQUEST HANDLER. A similar approach is used for the return value although this is not shown in the diagram. The message is received by the CLIENT REQUEST HANDLER and returned to the REQUESTOR. It demarshalls the message using the MARSHALLER and returns the de-marshalled result to the CLIENT PROXY which, in turn, returns this result to the client.

On the server side, the mechanics are similar, but happen in reverse order with respect to the involved layers. This is exemplified in the next

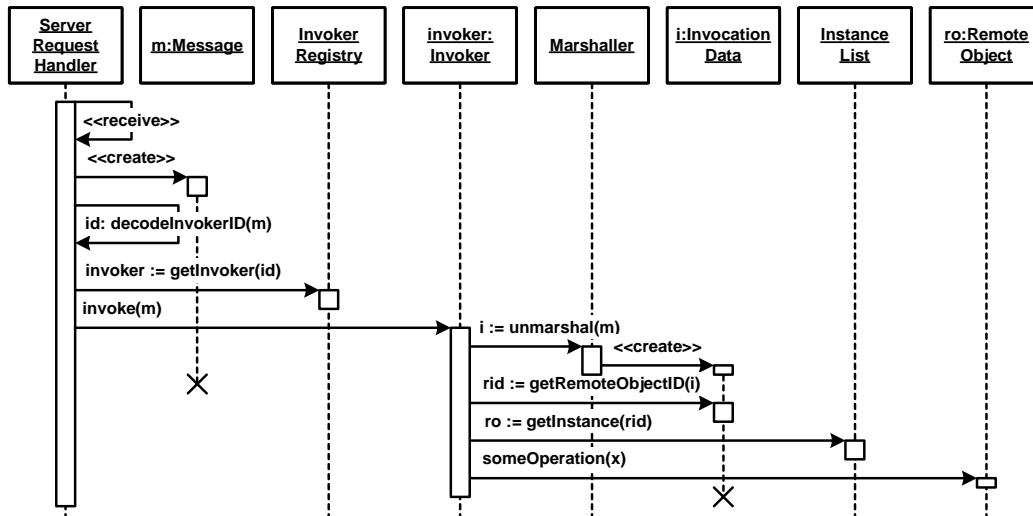
sequence diagram. Again, we do not show how return values are handled.



In this example, we presume that there is only one INVOKER. The SERVER REQUEST HANDLER does not have to do any demarshalling in order to find out which INVOKER to use. The INVOKER uses the MARSHALLER to re-create the invocation data object that was marshalled on client side. Using the information in that object, the INVOKER performs the invocation.

Now consider a situation that is a bit more complex: In the next diagram, we use a registry for INVOKERS, where INVOKERS can be registered to be responsible for particular remote object types. For instance, different INVOKERS might implement different activation strategies. The SERVER REQUEST HANDLER must be able to decode the ID of the remote object type or the OBJECT ID which contains the remote object's type. Using this information the INVOKER can be selected from the registry.

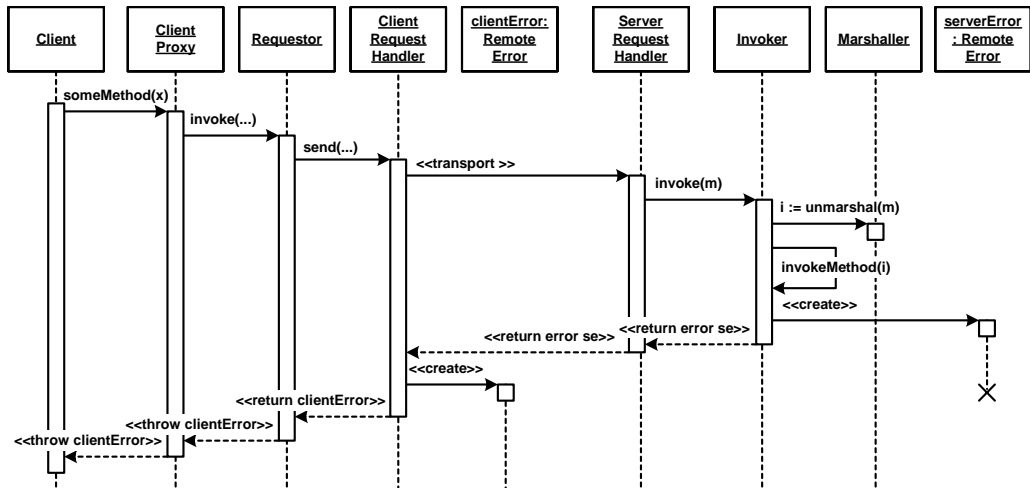
Further, each INVOKER contains an instance list of the objects that it can invoke. Thus the INVOKER has to get the target remote object from the instance list - using its ID - before the object can be invoked.



Assuming everything works fine in the above sequence diagram, the result of the invocation is returned to the client. If no object corresponding to the OBJECT ID is registered in the instance list, a REMOTING ERROR would have to be raised. In this case a REMOTING ERROR is sent back to the client.

If a REMOTING ERROR arrives at the client side, the REQUESTOR raises an exception instead of handing back the result. This sequence is illustrated in the following sequence diagram. In the diagram, the REMOTING ERROR is propagated to the client. Here, the CLIENT REQUEST

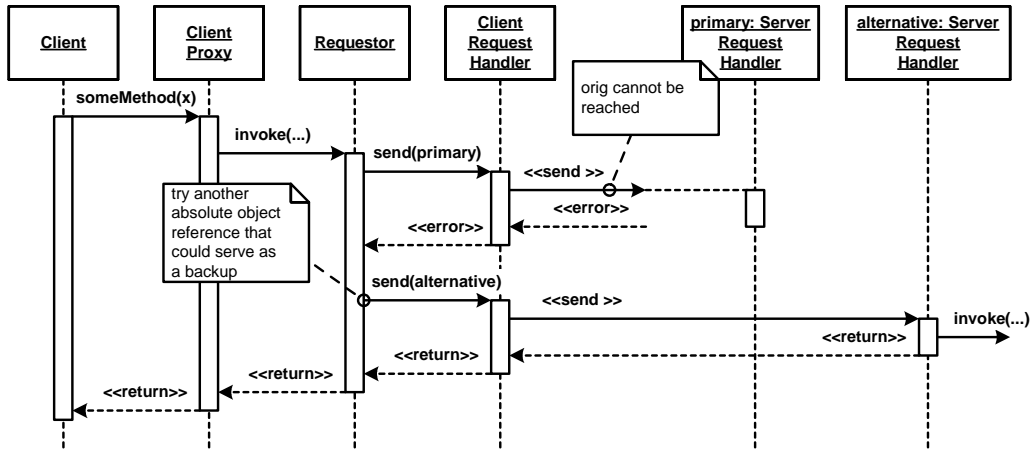
HANDLER creates an object describing the error and then raises an exception to be handled by the client.



Note that technically, there are two REMOTING ERROR objects involved: *serverError* is the object created in the server application, whereas *clientError* is the error object that is created on the client side to signal the REMOTING ERROR to the original caller. The two objects contain the same error information: this information must be marshalled on server side and be transmitted to the client side.

Instead of propagating the REMOTING ERROR to the client, the REQUESTOR or CLIENT PROXY might handle the error transparently. The next diagram shows an example, where the REQUESTOR tries to contact

a different server application because the original one could not be reached.

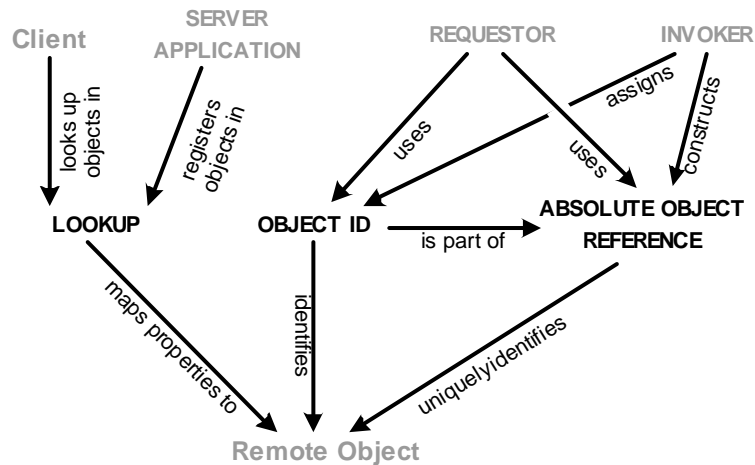


8

Identification Patterns

This chapter documents pattern for identification, addressing, and lookup of remote objects. This includes the assignment of logical OBJECT IDS for each remote object instance, as well as the notion of ABSOLUTE OBJECT REFERENCES that allow remote objects to be uniquely identified and addressed by remote clients. The pattern describes how to associate remote objects with properties and how to search for them later using those properties. One typical example of a property used for LOOKUP are human readable names; others are name-value pairs.

The figure below illustrates the relationship of the Identification patterns among each other and to the Basic Remoting patterns from the previous section.



Object ID

The INVOKER has to select between registered remote objects to dispatch an invocation request.



The INVOKER is responsible for dispatching invocations received from the SERVER REQUEST HANDLER. It invokes the operation of a remote object on behalf of a client. However, the INVOKER handles invocations for several remote objects, and the INVOKER has to determine the remote object corresponding to a particular invocation.

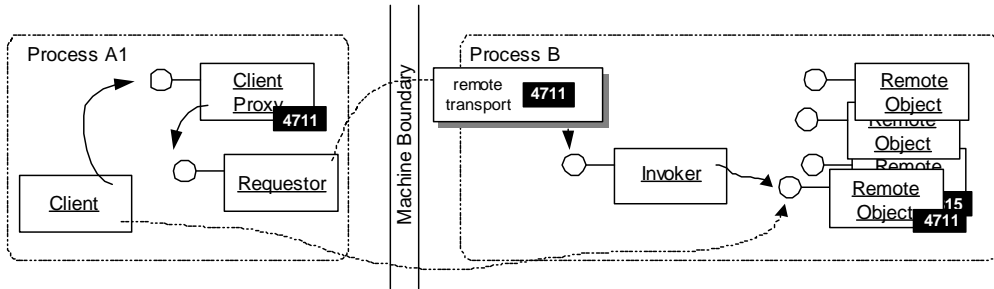
In a local environment, when a client invokes an operation on an object, the object is typically identified by its unique memory address. In distributed object middleware using the memory address for identification does not work because client and server run in different address spaces. A pointer to a memory address in a server refers to different data than in the client. Thus, memory addresses are not suitable for identification of remote objects in distributed object middleware.

When a request arrives from a client at the INVOKER, the INVOKER needs to know to which of the potentially many remote objects the invocation has to be dispatched to. On the client side, the remote object must be identified so that an invocation request can refer to the correct remote object, when it arrives at the INVOKER.

Therefore:

Associate each remote object instance with an OBJECT ID that is unique in the context of the INVOKER the remote object is registered with. The client, or the CLIENT PROXY respectively, has to provide the OBJECT ID so that the REQUESTOR can include it in the invocation

request and the INVOKER can dispatch the request to the correct remote object.



The CLIENT PROXY hands over an invocation to the REQUESTOR. It provides the OBJECT ID of the target remote object to the REQUESTOR. The OBJECT ID is sent with the invocation request to the INVOKER. Once the invocation arrives at the INVOKER, the INVOKER uses the OBJECT ID contained in the request to look up the correct instance. Then it invokes the operation on this instance.

* * *

A remote object's OBJECT ID can be any kind of identifier, but usually it is a numerical value (e.g. with 32 or 64 bits) or a string-based identifier. In any case, the range of possible values of the used type must be large enough to distinguish all remote objects registered at the same time in an INVOKER. The OBJECT ID has to be unique only inside an INVOKER as we assume the correct server and correct INVOKER have already been identified by other means. For an absolute identification of the server, the INVOKER, and the remote object, clients use ABSOLUTE OBJECT REFERENCES.

Another important aspect of OBJECT IDS is comparing for equality of remote objects. It is often necessary to find out whether two CLIENT PROXIES refer to the same remote object or not. It should be possible to perform this comparison without contacting the remote object and incurring communication overhead. The most efficient means to perform such a comparison is by comparing the OBJECT IDS of the targets in the client. But since the OBJECT IDS are often only unique inside one INVOKER, such comparisons need to be handled with care. Therefore, some distributed object middleware implementations use

globally unique OBJECT IDS. There are several algorithms available to create identifiers that are unique over space and time and still have a practical and useful length (see for example [Ope97]). A globally unique OBJECT ID is different from an ABSOLUTE OBJECT REFERENCE, as the ABSOLUTE OBJECT REFERENCE also contains location information on how to contact the server over the network.

Using the OBJECT ID pattern together with the INVOKER pattern enables advanced lifecycle control features, such as those provided by CLIENT-DEPENDENT INSTANCE, STATIC INSTANCE, PER-REQUEST INSTANCE, LAZY ACQUISITION, and POOLING. Those patterns require remote objects to be logically addressable. That is, clients are able to invoke operations on instances that are not actually physically present but are instantiated or acquired from a pool on demand by the INVOKER. This behavior is typically configured and is delegated by the INVOKER to the LIFECYCLE MANAGER.

OBJECT IDS identify remote objects rather than the servants realizing the remote object at runtime, because remote objects do not necessarily map one-to-one to physical instances within the server application. For instance, the OBJECT ID of a PER-REQUEST INSTANCE identifies the remote object rather than the individual servant handling the request. This is necessary because the servant is created only after the request arrives - thus it is not possible to reference the servant with an ID before the request arrives at the server.

A particular remote object can be represented by a CLIENT PROXY in the client address space. All invocations that originate from this CLIENT PROXY should reach one and the same particular remote object. Therefore, as the REQUESTOR and CLIENT REQUEST HANDLER are generic, the identity of the remote object is stored inside the CLIENT PROXY.

Note that the use of the OBJECT ID pattern in distributed object middleware is not mandatory. It is also possible to identify an object by the connection over which a request arrives. However, this only works if each remote object has its own dedicated connection. This is done in some embedded systems, where there is only a very limited and statically known number of remote objects. Using endpoint-based object identification saves dispatching overhead and improves performance. For more details about server-side connection handling in a distributed object middleware refer to the SERVER REQUEST HANDLER pattern.

Absolute Object Reference

The INVOKER uses OBJECT IDS to dispatch the invocation to the target remote object.



The OBJECT ID of a remote object allows the INVOKER to dispatch the remote invocation to the correct target object. However, the REQUESTOR in combination with the CLIENT REQUEST HANDLER first has to deliver the invocation containing the target remote object's OBJECT ID to the SERVER REQUEST HANDLER and INVOKER.

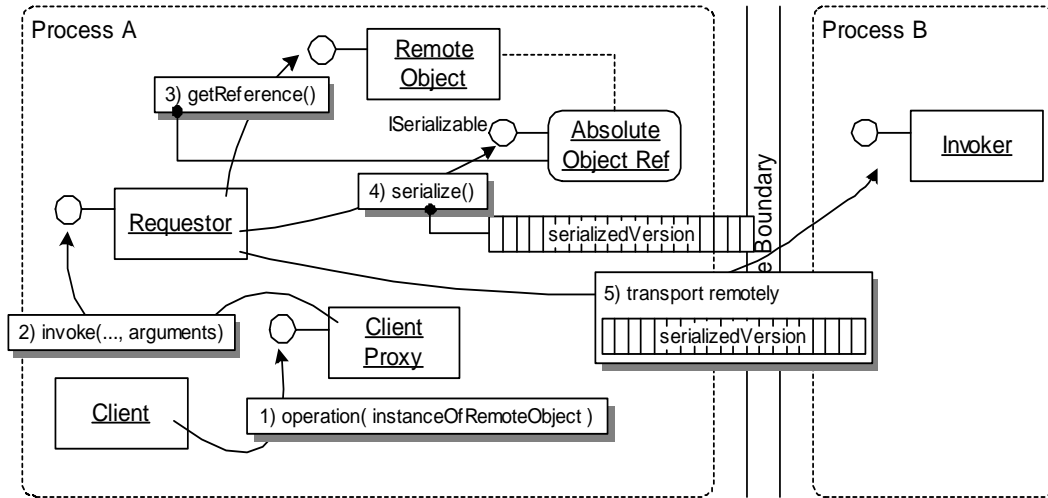
The client invokes operations via a REQUESTOR or using a CLIENT PROXY and the REQUESTOR forwards the invocation via REQUEST HANDLERS to the INVOKER. To allow a client to address the remote object, its OBJECT ID is included in the remote invocation, allowing the INVOKER to dispatch it correctly. Before the INVOKER can do this, however, the invocation request has to reach the correct INVOKER. Thus, the client needs to associate the network endpoint of the INVOKER, and the SERVER REQUEST HANDLER respectively, with the OBJECT ID. Note that this is independent of the uniqueness of the OBJECT ID. Even if the OBJECT ID is globally unique, the REQUESTOR, and CLIENT REQUEST HANDLER respectively, need to determine where to send the invocation request to.

The simplest way for associating network endpoints with OBJECT IDS would be for the client to keep a map between OBJECT IDS and network endpoints. The manual and explicit maintenance of such a mapping is tedious; further it is cumbersome in case references to remote objects have to be exchanged between clients.

Therefore:

Provide ABSOLUTE OBJECT REFERENCES that uniquely identifies the INVOKER and the remote object. Let the ABSOLUTE OBJECT REFERENCE include endpoint information, for example host and port number of the network peer, the ID of the INVOKER, as well as the OBJECT ID of

the target remote object. Clients exchange references to remote objects by exchanging the respective ABSOLUTE OBJECT REFERENCES.



A client invokes an operation that accepts a reference to another remote object as parameter. Before marshalling the parameter, the REQUESTOR retrieves the ABSOLUTE OBJECT REFERENCE to that remote object. The serialized version of the instance is then sent as part of the invocation request to the INVOKER.



An ABSOLUTE OBJECT REFERENCE must contain enough information for the client to contact the INVOKER the remote object is registered with, as well as the OBJECT ID of the remote object. Depending on the environment and technology, endpoint information (such as host and port) is sufficient, but sometimes also protocol information needs to be added. Additionally, in the case of multiple INVOKERS, the INVOKER needs to be identified, too.

If the distributed object middleware uses a predefined default protocol, it is sufficient if the reference contains the target endpoint. Since there is no choice, the client knows which protocol to use to contact the INVOKER.

Alternatively, if a distributed object middleware supports several protocols, the protocol to be used by the CLIENT REQUEST HANDLER has

to be included in the reference. For example, CORBA object references always specify the protocol besides the host name, port, INVOKER ID, and OBJECT ID, where INVOKER ID and OBJECT ID are encoded in binary format as a separate opaque part of the object reference.

When references to other remote objects are passed as parameters between clients and remote objects, the ABSOLUTE OBJECT REFERENCES are actually exchanged between them. Since the references are locally represented as CLIENT PROXIES of the corresponding remote objects, the MARSHALLER gets the ABSOLUTE OBJECT REFERENCE from the CLIENT PROXY. In the reverse direction - from server to client - a parameter containing an ABSOLUTE OBJECT REFERENCE leads to the instantiation of a CLIENT PROXY, requiring the availability of the necessary CLIENT PROXY classes on the receiver's side. In some systems/languages (especially if they are interpreted) the construction of CLIENT PROXIES can be done on-the-fly using reflection and runtime code generation. Alternatively, some distributed object middleware systems allow to send complete CLIENT PROXY implementations across the network.

The choice between making an object remotely accessible via pass-by-reference or pass-by-value, which involves the serialization and transmission of the objects state and sometimes even code, depends on the use case and the heterogeneity of the environment. If client and server application use the same programming languages, it might be possible to pass state and code. For example, to pass code, RMI transmits a URL reference to a location that provides the Java class code. But in most cases of pass-by-value only the state is transmitted, a suitable implementation is assumed to be available in the receiving process. For more details see the *Related Concepts, Technologies, and Patterns* chapter, specifically the discussion on mobile agents.

ABSOLUTE OBJECT REFERENCES might be opaque to client applications. That is, it is not possible for the client to construct a reference and access the remote object. The main reasons for opaqueness are to avoid the risk of creating unusable references based on wrong identity assumptions, and to allow the middleware to add special dispatching hints to the ABSOLUTE OBJECT REFERENCE, e.g. for optimization. Nevertheless, practice proved opaque references to be cumbersome, since it requires LOOKUP or explicit/manual distribution of the opaque references. Therefore, later versions of distributed object middleware systems also allow for non-opaque references, see corbaloc [Omg04a] and Ice

[Zer03]. Today, middleware technologies support either one or both forms of ABSOLUTE OBJECT REFERENCES: Opaque references that contain dispatching and optimization hints, as well as non-opaque references that can be constructed by the client application. Non-opaque references are typically only used for initial access to key remote objects of the distributed application. Often, URIs are used for that purpose.

ABSOLUTE OBJECT REFERENCES are either transient or persistent:

- Transient ABSOLUTE OBJECT REFERENCES become invalid after the remote object has been deactivated or the server application has restarted. A client who has obtained an ABSOLUTE OBJECT REFERENCE cannot be sure that the remote object it references still exists. Often, LEASES are used to manage the validity of references and to allow the INVOKER to passivate or remove the remote object to which no client reference exists anymore.
- Persistent ABSOLUTE OBJECT REFERENCES are valid even after the server application has been restarted, but the remote object needs to be registered with the same INVOKER and endpoint configuration. Persistent object references are supported by CORBA, for example.

To cope with changing transient references, distributed systems often use an indirection through a lookup service. LOOKUP allows to map 'persistent' logical names to (transient) ABSOLUTE OBJECT REFERENCES. The same principle is applied by LOCATION FORWARDERS in load balancing scenarios: An additional level of indirection translates a persistent reference into a transient reference.

To keep existing ABSOLUTE OBJECT REFERENCES valid after remote objects have been moved to a different server application, either LOCATION FORWARDERS are used, or the ABSOLUTE OBJECT REFERENCE is made to contain several endpoints, so that the REQUESTOR has alternatives when communication with the original endpoint fails. This is interesting for scenarios requiring automatic failover. Note that this only works when the remote objects are stateless or the state of the remote objects is replicated among the various server applications.

Lookup

Client applications want to use services provided by remote objects.



To use a service provided by a remote object, a client has to obtain an ABSOLUTE OBJECT REFERENCES to the respective object. Further, the remote object providing the service might change over time, either because another instance takes over the role, or because the server application has been restarted, and thus, the transient reference has changed. Despite such changes clients need a valid initial reference to access the service.

Most straight-forward options of distributing object references are not practical, those include:

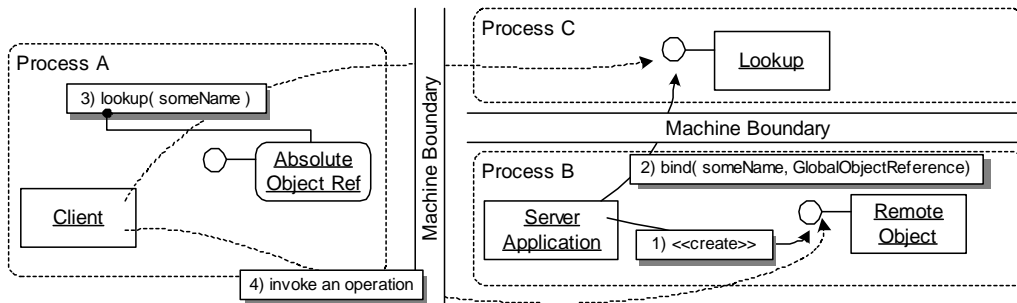
ABSOLUTE OBJECT REFERENCES for the remote objects can typically be serialized into a string and exchanged between server application and client manually via configuration files or even e-mail. But doing this on every client restart or change of the configuration is tedious.

Reusing references obtained during previous runs of the client program has to be done with care, because the reference to the remote object might be transient and therefore might have become invalid as a consequence of a server application restart or the relocation of the service on another remote object, hosted by a different server application. Even in the case of non-opaque ABSOLUTE OBJECT REFERENCES—references constructed by the client—the client would have to know the endpoint information, INVOKER id and OBJECT ID. Clients typically do not know this information because this would compromise the goal of location transparency, one of the fundamental concepts of distributed object middleware.

Another way to obtain a references for a specific remote object is to invoke an operation of some “other” remote object that returns the desired reference. However, where does the client get the reference to the “other” remote object from?

Therefore:

Implement a lookup service as part of the distributed object middleware. Let server applications register references to remote objects and associate them with properties. Clients then use the lookup service to query for ABSOLUTE OBJECT REFERENCES of remote objects based on these properties. The most common case is to use unique names for each reference of a remote object.



The server application binds the reference of a remote object to a symbolic name in the lookup service after instantiating the remote object. The client looks up the ABSOLUTE OBJECT REFERENCE using the same symbolic name, gets the reference returned, and finally invokes an operation on the remote object.



The lookup service provides an interface to bind and lookup ABSOLUTE OBJECT REFERENCES to properties and is typically implemented as a remote object itself. Of course, clients need to know the reference to the lookup service remote object before they can query it. This leads to a kind of hen-and-egg problem, as LOOKUP cannot be used to find the lookup service. To solve this, clients are typically manually configured using a persistent ABSOLUTE OBJECT REFERENCE to the lookup service or use some kind of multicast to search in the local network for a lookup service instance. The lookup service is considered a *well-known* object.

If the lookup service uses names, the names are typically structured hierarchically, just like qualified names in a file system. For example the following string could be a valid name:

/com/mycompany/accounting/accountfactory

The last element of such a structured name is usually called a binding, whereas all other parts are called contexts. Obviously, contexts can be nested. Lookup services based on unique, structured names are often called *naming services*.

More complex lookup services use arbitrary properties, typically implemented as name-value pairs. The server application uses the lookup service to associate the properties and their associated values with the respective remote object. Clients can query for references by specifying expressions such as:

```
printertype="hplaser" and location="building10" and color="true"
```

The lookup service will return a set of references that conform to the query. Lookup services supporting such complex queries are often referred to as *traders*.

Even complex lookup services provide a generic interface regarding the type of remote object returned from a query. When using CLIENT PROXIES, clients have to cast the returned reference to the correct type. It is important to understand that the registration of a remote object in a lookup service does not necessarily mean that the instance is actually physically created, it only means that it is legal for clients to invoke operations on it. Of course, this only works if INVOKERS are able to create references for not yet activated remote objects and maintain the mapping between the reference and the remote object type. As a consequence, these not-yet-existing objects need to be created on demand once an invocation arrives at the respective INVOKER. More details on decoupling of reference creation and remote object activation are provided in the chapter *Lifecycle Management Patterns*.

To avoid the lookup service being filled with dangling references—references to objects that are no longer available—a server application should unbind a reference when it stops serving it. To manage unbinding automatically, the lookup service should use LEASING for the bindings: when a server application registers a remote object, the lookup service establishes a lease that the server application must periodically renew. If the server does not remove the binding, for example due to a crash of the server application, the lease will eventually time out and the lookup service can remove the reference along with the properties.

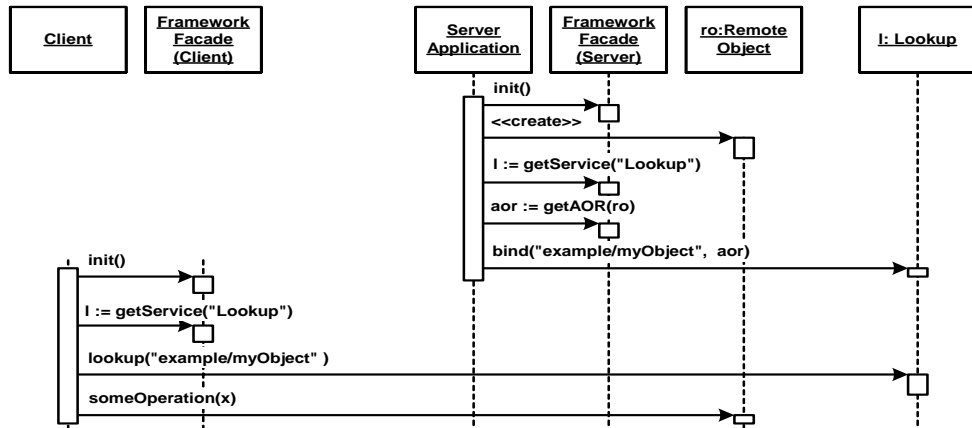
LOOKUP can also be used for load balancing by registering more than one remote object for a property/name entry in the lookup service and having the lookup service dispatch those reference randomly, round-robin, or load-based.

To avoid performance bottlenecks and single points of failure, clients should use LOOKUP sparsely. For example, you should only register factory objects, in the sense of Factory Method [GHJV95] and Abstract Factory [GHJV95], and obtain further references through them. To further reduce the chances for failure, lookup services are typically replicated and/or federated on several hosts. Keeping the replicas consistent does incur a performance overhead, which is another reason for only storing a few important “initial” objects in the LOOKUP service.

Interactions among the Patterns

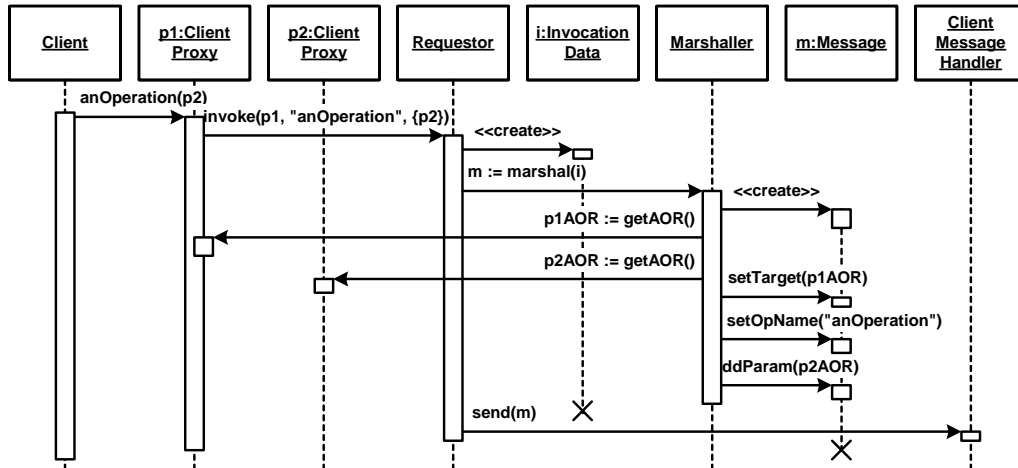
The three patterns introduced in this chapter are used for locating objects. The use of the OBJECT ID pattern has already been implicitly shown in the sequence diagram examples of the *Basic Remoting Patterns* chapter, where the OBJECT ID was used by the INVOKER to find the target remote object in its list of instances.

The following sequence diagram shows how a server application registers a reference of a remote object in LOOKUP and how a client uses the lookup service to retrieve this reference. The ABSOLUTE OBJECT REFERENCE of the lookup service has been configured manually, both in the server application and in the client application. Both server application and client use a framework facade to retrieve the initial reference to the lookup service. Here the framework facade acts a utility class providing access to the middleware's core features. The lookup service is implemented itself as a remote object, offering the operations *bind()* for registering remote objects and *lookup()* for looking up ABSOLUTE OBJECT REFERENCES of remote objects.



The next illustration shows how a client invokes an operation of a remote object passing another remote object as parameter. The MARSHALLER gets the invocation parameters and recognizes that some of them are remote objects located on some server. In this example, the MARSHALLER converts these parameters into ABSOLUTE OBJECT REFERENCES. The ABSOLUTE OBJECT REFERENCES are stored in the CLIENT PROXIES of the remote objects and are obtained using the operation

getAOR(). This information is then handed over to the CLIENT REQUEST HANDLER by the REQUESTOR.



9

Lifecycle Management Patterns

This chapter presents patterns for managing the lifecycle of remote objects. The patterns describe behaviors and strategies, not building blocks, and are therefore different from the patterns in the previous chapters. The patterns are typically implemented with the help of the LIFECYCLE MANAGER pattern, which is presented in the *Extended Infrastructure Patterns* Chapter.

This chapter is structured as follows: First, we describe three basic patterns of remote object lifecycle management: STATIC INSTANCE, PER-REQUEST INSTANCE, and CLIENT-DEPENDENT INSTANCE. These patterns are motivated via usage examples of how clients typically use remote objects. Then we present three resource management patterns, LAZY ACQUISITION [KJ04], POOLING [KJ04], and LEASING [KJ04], as well as a pattern for state management in the context of remote objects: PASSIVATION [VSW02]. All four patterns have already been described in existing literature, albeit in different contexts. Finally, we describe typical use cases that show how the patterns can be combined to optimize the resource usage of distributed applications.

In the context of this chapter, we need to explicitly consider the *servant* of a remote object. The servant is the runtime object that represents a remote object in the server application. Since the relationship between a servant and a remote object is not strictly one-to-one for all activation strategies described in this chapter, it is essential to distinguish these two concepts. The basic idea of distinguishing servant and remote object is also described as a *Virtual Instance* [HV99, VSW02].

To better understand the issues of lifecycle management, the concept of stateful and stateless objects is important. In object-oriented programming each object has state as soon as it has member variables. This is also true for remote objects, but additionally it has to be distinguished whether the remote object state is persistent or transient with respect to subsequent invocations. If the state is transient, the remote object is

considered stateless, if the state is persistent, the remote object is considered stateful.

Whether the state of a remote object is persisted in a database or not, is orthogonal to the issue whether clients expect the remote object to maintain state (transient versus persistent state). Transient state can, but need not, be persisted. For example, if the object just has read-only state, it does not require persistence of state. Persistent state must be written to a database in case the remote object is temporarily deactivated. Since persistent state ‘must’ be persisted (see *PASSIVATION*), stateful remote objects are more costly to activate and deactivate, and in consequence are kept longer in memory, using valuable resources. Remote objects with transient state can be managed using *POOLING*, which requires fewer memory resources.

A brief note on activation and deactivation. By the term *activation* we mean every action that is necessary for the distributed object middleware to make a remote object ready to receive an invocation. This typically includes instantiation of the servant, initialization, and registration with the *INVOKER* and *LIFECYCLE MANGER*. Additionally, it can also include the registration with the lookup service and restoring the remote object state. *Deactivation* means to reverse the activation steps, such as persisting remote object state, unregistering the remote object, and possibly even destroying the servant. Note that deactivation is not the same as *PASSIVATION*. *PASSIVATION* keeps the remote object available to clients while temporarily removing its servant from the server applications’s memory.

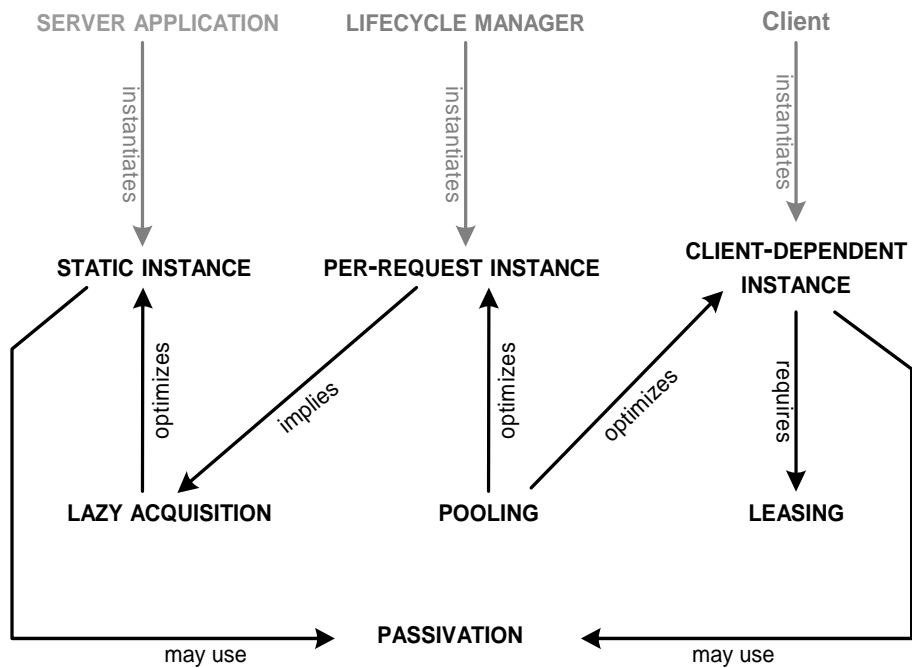
Basic Lifecycle Patterns

The chapter starts with the following three basic lifecycle patterns. They describe strategies of lifecycle management of remote objects:

- *STATIC INSTANCES* are remote objects whose servant’s lifetime is independent of the lifetime and the access patterns of their clients. Notionally, they live forever and are accessible to all clients in the distributed system.
- In case of a *PER-REQUEST INSTANCE*, the distributed object middleware’s *LIFECYCLE MANAGER* (logically) creates and destroys new servants for each and every individual method invocation.

- **CLIENT-DEPENDENT INSTANCES** rely on the client to instantiate a remote object and its servant explicitly. A **CLIENT-DEPENDENT INSTANCE** is destroyed either on a client's explicit request, or by the **LIFECYCLE MANAGER** once its **LEASE** has expired.

The following illustration shows the relationships among the patterns. We already show their relationships to the patterns **LAZY ACQUISITION**, **POOLING**, **LEASING**, and **PASSIVATION** which are introduced in the second part of this chapter.



Static Instance

A remote object offers a service that is independent of any specific client.



The server application has to provide a number of previously known remote objects instances - maybe the number is even fixed over the lifetime of the server application. The remote objects must be available for a long period without any predetermined expiration timeout. The remote object's state must not be lost between individual invocations, and be available to all clients.

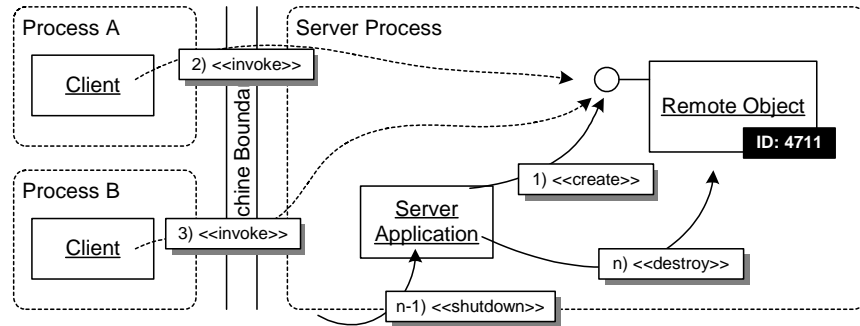
As an example, consider a factory control system where a central control system coordinates a set of machines. Each machine has its own control computer that runs a server application. Such a machine typically consists of a predefined number of subsystems such as sensors and actuators. The central control software needs to communicate with these subsystems in order to coordinate the manufacturing process. Using a distributed object middleware each subsystem (actuator, sensor) can be realized as a remote object that will be accessed by the central control software and potentially other clients (such as a monitoring tool).

When remote objects represent physical devices, as in this scenario, the number and types of remote objects is previously known and typically does not change because it directly corresponds to the physically installed devices. The lifecycle and state of the remote objects is independent of the client's lifecycle: in the case of the scenario, the state and lifecycle is associated with the physical device.

Therefore:

Provide STATIC INSTANCES of remote objects that are independent of the client's state and lifecycle. They are activated before any client's usage, typically as early as during application startup, and deacti-

vated when the application shuts down. For easy accessibility by clients the instances are registered in LOOKUP after their activation.



During initialization of the server application a number of STATIC INSTANCES are activated, becoming available for use by clients. Once the server application shuts down all STATIC INSTANCES are deactivated.



STATIC INSTANCES are typically activated eagerly, as described by the *Eager Acquisition* [KJ04] pattern. STATIC INSTANCE servants are acquired before their first usage in order to optimize runtime behavior, allowing for fast and predictable access times. Such behavior is especially important for embedded and real-time scenarios.

Using eager activation of remote objects can be a problem because the servant's resources stay acquired for the lifetime of the server application and the system might not have enough physical resources to host all the remote objects at the same time. Applications with a large numbers of STATIC INSTANCES have to consider alternatives to reduce the resource consumption. Examples of such alternatives are:

- Instead of *Eager Acquisition*, LAZY ACQUISITION can be used to only instantiate a servant for the remote objects when they are actually accessed. Though, as a consequence, predictability is lost and invocation times might increase, which is often a reason for using STATIC INSTANCES in the first place.
- To save and restore the state of the remote object, PASSIVATION can be used to temporarily remove servants from memory when the remote object is not accessed by clients for a certain period of time.

A new servant - containing the state that has been saved previously - will be created when the next invocation request arrives.

Stateless STATIC INSTANCES are advantageous to support load balancing and failover techniques. The remote objects are simply replicated using LOOKUP-based techniques or applying LOCATION FORWARDER to redirect messages to one of the replicas. If state has to be replicated, matters become more complicated. One of the easiest ways out is to establish shared state between the STATIC INSTANCES, for example in a shared database. In the chapter *Related Concepts, Technologies, and Patterns* we present some further patterns from [DL03, DL04] to improve availability and scalability.

Since a STATIC INSTANCE is accessed by many clients concurrently, either the servant needs to be thread-safe, or the INVOKER needs to serialize the access.

Per-Request Instance

The remote objects are stateless.



A server application serves remote objects that are accessed by a large number of clients and accessing the remote objects needs to be highly scalable regarding the number of clients. When many clients access the same remote object concurrently the performance of the server application decreases dramatically because of synchronization overhead.

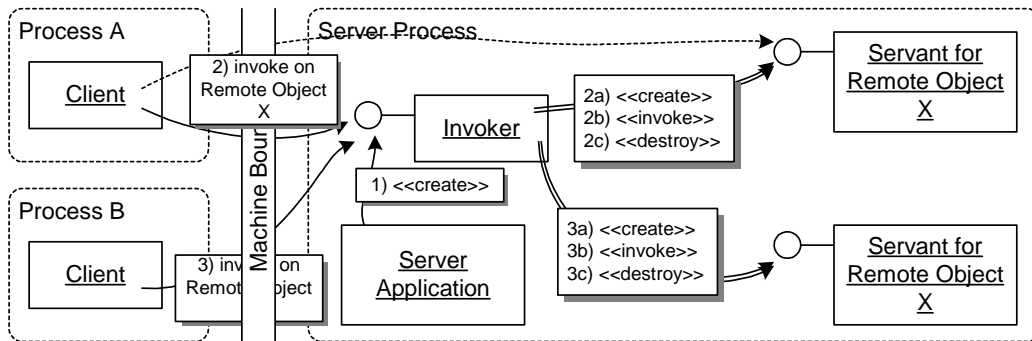
Consider a system that provides information on stock prices. A client invokes operations passing in a stock symbol (such as 'DCX' for Daimler Chrysler) to query the current quote, as well as some historical data to compute a graph. To retrieve the stock information, the remote object implementing the queries typically performs one or several queries to a database, processes the results, and returns them to the client.

In this and similar scenarios, individual invocations are independent of each other. Semantically, it does not matter whether the invocations of individual clients are handled sequentially or concurrently. They do not modify any shared state, they just return read-only information.

Implementing this service as a `STATIC INSTANCE` would cause a bottleneck, because all clients would use the same instance that consequently needs to serialize the access. On the other side, using multiple `STATIC INSTANCES` to better distribute the load would unnecessarily waste resources in situations of low system load.

Therefore:

Let the distributed object middleware activate a new servant for each invocation. This servant handles the request, returns the results, and is then deactivated again. While the client expects to get a “fresh” instance from request to request, the activation and deactivation can internally be optimized, for instance, using `POOLING` of servants.



For each invocation on a specific remote object, the INVOKER creates a new servant, dispatches the invocation to the newly created servant, waits for the result, and then destroys the servant again.



Regarding the activation strategy, the PER-REQUEST INSTANCE pattern uses the *Lazy Acquisition* [KJ04] activation strategy. Activating a PER-REQUEST INSTANCE results in the preparation of the LIFECYCLE MANAGER to accept requests for the respective remote object. No servant is created to actually represent the remote object. Only when a request arrives for the previously activated remote object, the LIFECYCLE MANAGER creates a new servant to handle the request, and then immediately discards it again. The client's reference to the remote object will be valid all the time. The OBJECT ID of a PER-REQUEST INSTANCE logically identifies the REMOTE OBJECT - and not the individual servant. This is necessary because the servant does not exist before the request arrives and thus cannot possibly be referenced by the OBJECT ID.

Depending on the resources available to the server application, the strategy chosen by PER-REQUEST INSTANCES can scale very well, and concurrency is not an issue on the servant level since each request has its own instance. However, there are also some drawbacks. For example, PER-REQUEST INSTANCES require that the remote objects are stateless with regards to the client. That is, all invocations must be self-contained. Clients must not expect to see changes made to the object's

state between two invocations of the same (or different) client — after all they talk to a new servant for each invocation.

Of course, a PER-REQUEST INSTANCE can change shared state that is kept outside the servant itself, for example in an associated database. The invocation would have to refer to that shared state somehow, for example specifying the primary key of the data to be altered.

As long as the servants of such remote objects do not access and change shared state (in session objects or in a database), developers need not care about multi-threading issues and it can be left to the LIFECYCLE MANAGER to decide how many requests should be handled in parallel; it mainly depends on the number of threads that can run concurrently without too much performance penalty. However, if the LIFECYCLE MANAGER uses several threads, developers need to be careful when accessing shared resources (especially for write access). Access to such resources must be protected using synchronization primitives, for example by introducing critical regions in the servant or by adding a monitor “around” the shared data. Managing concurrent access to shared resources can be handled inside remote objects themselves or by using *Proxies* [GHJV95] that protect the resources (see the *Managed Resource* pattern in [VSW02]).

Distributed object middleware implementations leave optimizations, such as the number of threads, to the LIFECYCLE MANAGER and INVOKER. Server developers only have to limit this number in order to make sure that the system is not overloaded with too many threads. If too many concurrent requests arrive, it is the task of the INVOKER to serialize these invocations or alternatively signal an ‘overload’ condition back to the client using a suitable REMOTING ERROR.

In some systems, creating and destroying large numbers of servants as well as acquiring and releasing their resources, can be very expensive, limiting the scalability of the system. To remedy this problem, POOLING techniques can be used, as will be explained later.

Client-Dependent Instance

Clients use services provided by remote objects.



In situations where the application logic of the remote object extends the logic of the client, it becomes important to think about where to put the common state of both. Keeping the common state solely in the client requires the client to transmit its state with every invocation. Keeping the common state solely in the server, for example in a STATIC INSTANCE, requires complex mechanisms to keep the states of individual client's separate inside the remote object.

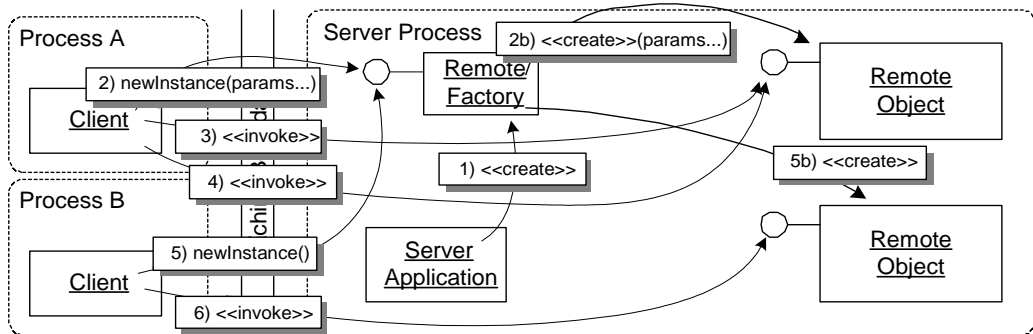
As an example, consider a server farm that provides sophisticated image processing of radio-astronomical data to clients. The server application provides a set of stateless algorithm objects, each performing a specific transformation of the image. A client processes an image using a sequence of such algorithms, each subsequent algorithm operates on the result of the previous one. The exact nature, sequence, and configuration of the algorithms used depends on the scientific goals of the client. That is, the client inspects intermediate results and decides about how to go on.

To implement this scenario, a large volume of state needs to be stored for each client: the original image, the current image after the execution of the last algorithm, diagnostic messages and problems that occurred during processing that could influence the accuracy of the result, as well as the algorithm sequence applied to the image and their parameterization. All this state is client-specific, the images and results of one client have typically nothing to do with the images and results of another client. Transmitting the image data repeatedly between client and server application is very inefficient because of the huge size of astronomical data sets.

Therefore:

Provide remote objects whose lifetime is controlled by clients. Clients create CLIENT-DEPENDENT INSTANCES on request and destroy them when no longer needed. As an instance is *owned* by a client, the client can consider the state of this instance to be private. Clients

typically use a *factory* remote object to request the creation of new instances.



A *factory* remote object is created by the server application and made available to clients through LOOKUP. Clients use this factory to create and configure CLIENT-DEPENDENT INSTANCES. The client then invokes operations on its instance, and destroys the instance when it does not need the instance anymore.

* * *

The most obvious difference of CLIENT-DEPENDENT INSTANCES to STATIC INSTANCES and PER-REQUEST INSTANCES is that an additional component is introduced: the factory. This is necessary to allow clients to request a CLIENT-DEPENDENT INSTANCE that is specific to the client. In case of a STATIC INSTANCE or a PER-REQUEST INSTANCE, creation and activation of the remote object is handled by the SERVER APPLICATION and/or the distributed object middleware.

The factory itself is typically a STATIC INSTANCE. When invoked by a client, it creates the requested CLIENT-DEPENDENT INSTANCE and returns the ABSOLUTE OBJECT REFERENCE of the newly created instance to the client.

Usually, as explained above, CLIENT-DEPENDENT INSTANCES are typically considered to be accessed by a single client only. As a consequence, no concurrency issues need to be taken care of in the servant—with regard to client requests and its own state. In case CLIENT-DEPENDENT INSTANCES can be accessed concurrently (for example because a client can open multiple browser windows that are

not coordinated among each other) then some kind of synchronization has to be implemented on server side.

Also, access to global shared resources from within CLIENT-DEPENDENT INSTANCES still needs to be serialized.

To ensure that only one client accesses a CLIENT-DEPENDENT INSTANCE, the factory will only provide operations to create *new* instances. It is not possible to ask the factory to return some previously created instance. Thus, the client has to store the reference to the instance, and it is the client's own responsibility whether to pass the reference to other clients.

Lifecycle management of CLIENT-DEPENDENT INSTANCES is easy if clients behave well. The clients are required to signal when to *destroy()* the remote object, permitting the server application to remove the servant from memory. However, if a client forgets to invoke *destroy()* or if a client crashes, the server application may end up with orphan CLIENT-DEPENDENT INSTANCES. To avoid this, *Leasing* [KJ04] is used. This allows the removal of orphan CLIENT-DEPENDENT INSTANCES when the lease period expires.

There are some potential problems with the lifecycle model of CLIENT-DEPENDENT INSTANCES, though. The more clients a server application has, the more CLIENT-DEPENDENT INSTANCES might be created. Thus, the server application is not in control of its own resources anymore. This can be a serious issue for large, publicly available systems where you cannot control the number and behavior of the clients. To overcome this problem, the LIFECYCLE MANAGER can use PASSIVATION. PASSIVATION temporarily evicts a servant from memory, persist its state and create a new servant - with the previously saved state - it upon the next client request.

General Resource Management Patterns

After introducing the three basic lifecycle management patterns for remote objects, we will now describe three resource management patterns originally described in [KJ04]: LAZY ACQUISITION, POOLING, and LEASING, as well as a state management pattern originally described in [VSW02]: PASSIVATION. We provide specializations of these patterns in the context of distributed object middleware.

After the introduction of the patterns, we will combine them with the basic lifecycle management patterns to show typical implementation strategies for lifecycle management of remote objects.

LAZY ACQUISITION defers the creation of a servant for a remote object to the latest possible point in time: the actual invocation of an operation on the remote object.

POOLING describes how instantiation and destruction of servants of remote objects can be avoided by recycling servants no longer needed. Besides servants there are many other resources, that can be pooled in a distributed object middleware, such as threads and connections – see the pattern descriptions of CLIENT REQUEST HANDLER and SERVER REQUEST HANDLER, for instance. Note that we refer to the original *Pooling* pattern in [KJ04] in these cases.

LEASING explains how the resources of remote object's servants, which are not needed anymore, can be reclaimed reliably. LEASING associates time-based leases with a servant that clients have to extend in order to use the remote objects.

Finally, PASSIVATION temporarily evicts servants from memory in case the remote object is, for instance, not accessed by clients for a certain period of time.

The implementation of the lifecycle management patterns in a distributed object middleware relies on the LIFECYCLE MANAGER as the key participant. The LIFECYCLE MANAGER is responsible for actually acquiring, pooling, leasing, and passivating the remote objects.

Lazy Acquisition

STATIC INSTANCES must be efficiently managed.



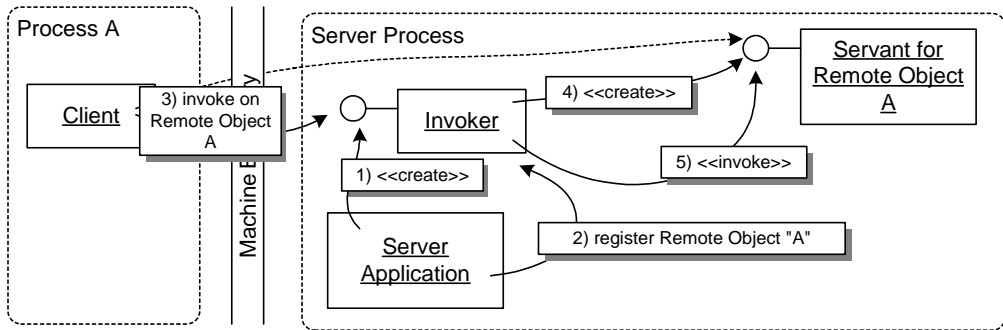
Creating servants for all remote objects that might possibly be accessed by clients during server application startup can result in wasting resources. The total amount of resources needed for all servants might even exceed the physically available resources in the server process. Additionally, instantiating all servants during server application startup leads to long startup times.

Acquiring resources and instantiating servants always costs valuable CPU cycles. When servants are instantiated long before their use, valuable resources such as server application memory are wasted to keep them alive. This influences the overall availability of resources as well as the stability of the system; if resources get scarce, client requests might stay unfulfilled or the system might crash since it cannot gracefully cope with resource starvation.

Additionally, if servants are instantiated too eagerly, the system startup time will be unnecessarily long. But systems are expected to startup quickly to be available for use by clients.

Therefore:

Instantiate servants only when their respective remote objects are actually invoked by clients. Still, let clients assume that remote objects are always available. The INVOKER triggers the LIFECYCLE MANAGER to lazily instantiate the servant when it is accessed by a client.



A remote object type has been registered with the INVOKER during server application startup, but the servant has to be instantiated lazily. Thus, upon the first request, the INVOKER lazily creates a servant. This servant is subsequently used by other client requests.

As soon as clients hold an ABSOLUTE OBJECT REFERENCE of a remote object, they naturally assume the remote object is available. While this is true, there need not necessarily be a servant created to serve the requests. For instance, resource constraints might lead to decoupling of remote objects and their servants.

Upon a request to a lazily acquired remote object, the INVOKER checks, whether there is already a servant available for the remote object. If this is not the case, it will trigger the LIFECYCLE MANAGER to instantiate a servant for the remote object. This way, only the remote objects that are actually accessed have a “physical representation” in the server application.

Using LAZY ACQUISITION, resource consumption is significantly lower compared to instantiating remote objects eagerly during startup. But as a drawback, this pattern also introduces some complexity in the LIFECYCLE MANAGER, which now has to check for the existence of servants. Further, the check and the LAZY ACQUISITION of the servants introduces some delay. Access times become more unpredictable, which might be a problem in time-sensitive, embedded systems. Large and complex objects save most resources when acquired lazily, but also take the

longest time to acquire. If this is a problem, *Partial Acquisition* [KJ04] is a possible solution. It acquires parts of a servant immediately and other parts lazily.

Before considering LAZY ACQUISITION as a solution to a resource shortage, you should be sure that the large number of remote object servants is actually needed. There is the danger that LAZY ACQUISITION is used to minimize the consequences of poor design in which too many remote objects are unnecessarily activated. That is, you should first think about ways to reduce the resource consumption of your server application, for instance, by evicting or reusing unused remote objects. Only if this fails, think about more complex resource management techniques such as LAZY ACQUISITION.

Also, note that LAZY ACQUISITION only solves half of the problem. Eventually, after some time, all remote objects will have been accessed. This means that eventually a large number of remote objects will have a servant, consuming large amounts of server application resources. So, in addition to instantiating servants lazily, we also need a means to get rid of servants not needed at the moment. PASSIVATION, LEASING, and POOLING can help here. Also, an *Evictor* [KJ04] that manages the destruction (eviction) of objects, is a possible solution.

A PER-REQUEST INSTANCE can be seen as an extreme form of LAZY ACQUISITION: a new servant for a PER-REQUEST INSTANCE is lazily instantiated when an invocation arrives and directly evicted after the invocation.

The LAZY ACQUISITION pattern is documented in more detail in [KJ04].

Pooling

Every remote object instance consumes server application resources.



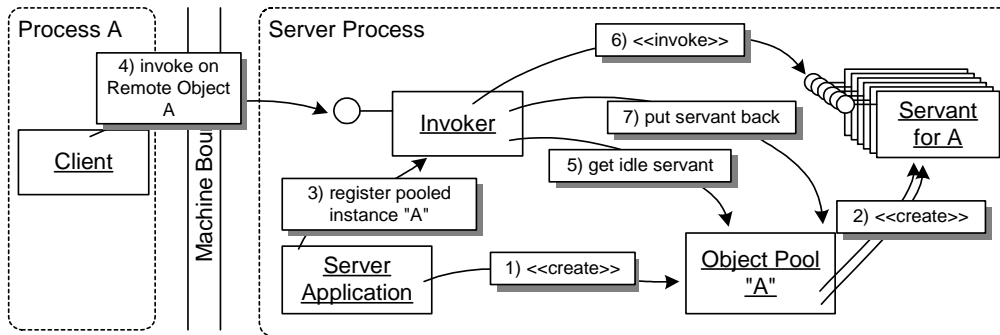
Instantiating and destroying servants regularly, as in the case of PER-REQUEST INSTANCES, causes a lot of overhead in terms of additional load for the server. Among other issues, memory needs to be allocated, initialization code for the servants needs to be executed, and the servants have to be registered with the distributed object middleware.

The overhead of creating and destroying servants heavily influences the scalability, performance, and stability of the server application. Scalability is impacted negatively because the CPU cycles needed for the acquisition and release of resources are not available to serve requests of other clients. The performance for each invocation is slow because the resource acquisitions during each invocation need time to execute. When servants are not destroyed, for example to save the overhead of destruction, stability is at risk, since resource contention might occur and new invocations might not get served.

Besides those reasons, predictability of invocation times is degraded - especially in environments that rely on garbage collection, such as Java. Garbage collection will be necessary in intervals to clean up the large number of discarded servants, causing the server performance to degrade substantially.

Therefore:

Introduce a pool of servants for each remote object type hosted by the server application. When an invocation arrives for a remote object, take a servant from the pool, handle the request, and put it back to the pool. Subsequent invocations will follow the same procedure when reusing the instance. The remote object's state is removed from the servant before it is put into the pool, servants taken from the pool are initialized with the state of the remote object they should represent.



The server application creates an object pool, which in turn creates a number of servant instances. The remote object type is registered as a pooled instance. Upon an invocation, the INVOKER takes an idle servant from the pool and performs the invocation using this servant. After the invocation has finished, the servant is put back into the pool.



The pool uses *Eager Acquisition* [KJ04] to instantiate a limited number of servants during pool creation in order to improve performance. If more servants are needed, the pool will grow dynamically up to a predefined limit. If that limit is reached, further invocations need to be queued. Of course, dynamic growth and queuing negatively impact performance and predictability. When fewer servants are needed than available in the pool, the pool will evict servants.

POOLING avoids the overhead of repeated instantiation, initialization, and destruction of servants. A servant managed by a pool might handle invocations for different remote objects of the same type during its lifetime. That is, the identity and the state the servants represents changes over time. The LIFECYCLE MANAGER has to initialize the servant when taken from the pool, and deinitialized it when it puts it back. Initialization and deinitialization result in some overhead. As a consequence, POOLING is especially well suited for remote objects that do not have identity and state (i.e. stateless remote objects). Here, all servants in a pool are considered equal. An invocation can be handled by any of the instances in the pool without any further setup or initialization.

The management of stateful remote objects is more complicated. When servants for remote objects are fetched from the pool, the LIFECYCLE MANAGER has to trigger them via a *Lifecycle Callback* operation [VSW02]. Triggered by the lifecycle callback operation, the servant taken from the pool acquires the state of the entity it represents in the context of the invocation. Alternatively, the LIFECYCLE MANAGER has to supply the stateful servant with the required state directly. To associate the state correctly, the LIFECYCLE MANAGER differentiates between remote objects using their OBJECT IDS. Such advanced lifecycle management strategies are implemented, for instance, by server-side component infrastructures, as documented in [VSW02].

The POOLING pattern is presented in more detail in [KJ04].

Leasing

Clients consume server application resources, for example by using CLIENT-DEPENDENT INSTANCES.



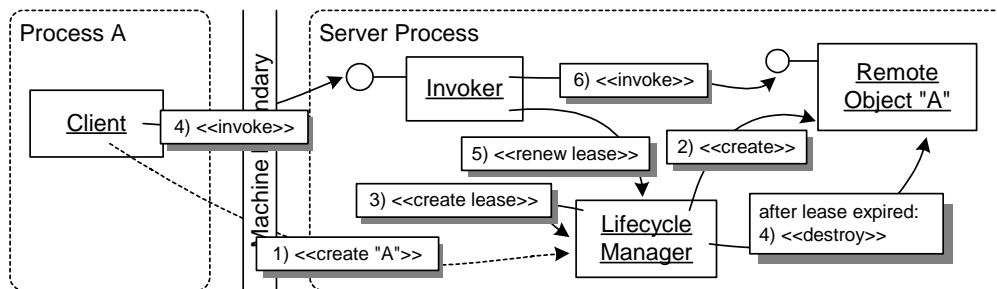
Remote objects and their servants no longer needed should be released in time to free unused system resources. However, the LIFECYCLE MANAGER cannot determine when a particular remote object is not used anymore in order to release it safely. In most scenarios, neither LIFECYCLE MANAGER nor the remote object itself know the clients and whether these still intend to access the remote objects in the future.

Consider a stateful remote object that is accessed by a single client, such as a CLIENT-DEPENDENT INSTANCE. The easiest way for a LIFECYCLE MANAGER to know that a remote object is not needed anymore is that the client just invokes a destroy operation on the remote object. When this operation is invoked by a client, the remote object will destroy itself and its resources, specifically, the servant, will be reclaimed. There is a problem though: if a client does not call this operation, the remote object might live forever and the server application cannot reclaim the servant resources.

There might be a couple of reasons why a client fails to call the destroy operation. Clients can simply be buggy and thus not call the destroy operation, they can crash unexpectedly, or the network connection may become unavailable.

Therefore:

Associate each client's use of a remote object with a time-based lease. When the lease for a particular client expires, the server application assumes that this client does not need the remote object anymore. As soon as all leases for a remote object have expired, the servant is destroyed by the LIFECYCLE MANAGER, and the remote object can be unregistered from the distributed object middleware. It is the responsibility of the client to renew the lease as long as it expects the remote object to be available.



A CLIENT-DEPENDENT INSTANCE is created by the LIFECYCLE MANAGER together with a lease for this instance. During an invocation the lease is automatically renewed. After a specified period of time without lease renewal, the lease expires, and thus the LIFECYCLE MANAGER destroys the leased instance.



The LEASING pattern frees clients from explicitly releasing unused remote objects. Since leases are valid only for a limited time, they need to be renewed regularly. There are three basic ways how lease renewal can be handled:

- A client's lease for a remote object is implicitly renewed with each invocation of the remote object. This requires the client to invoke the remote object in intervals shorter than the lease period, to keep remote objects "alive".
- The client can be required to explicitly invoke the renewal operation on the lease before the lease period ends. Here, the client can influence the duration of the lease by specifying an "extension period" as part of the lease extension request. Explicit renewal might also be needed in the case above, when the client's invocation interval is larger than the lease period. Network latency should be considered when short lease periods are used.
- The distributed object middleware informs the client of a LEASE'S upcoming expiration, allowing the client to specify an extension period. The client does not have to keep track of lease expiration itself, and thus the logic to manage the lifecycle of its remote objects becomes simpler. On the other hand, as network communi-

cation is unreliable, LEASE expiration messages might get lost and remote objects might unintentionally get destroyed. A further liability is that clients need to be able to handle those messages; thus they have to be servers, too.

Regarding the lease periods, there is a trade-off between:

- short periods, which cause less resources to be wasted, when clients no longer use remote objects, and
- long periods, which minimize communication overhead due to regular lease renewals.

Leases are not only beneficial within server applications, but also for clients. Clients typically only notice the unavailability of remote objects, when an REMOTING ERROR is returned in response to an invocation. Using LEASING, clients will potentially notice the unavailability of remote objects earlier: when the LEASE expires and LEASE renewal fails. This avoids the accumulation of dangling references, keeping the system more stable and current.

To avoid complicating the client application logic, the lease management is often handled by the CLIENT PROXY - to make it transparent for the client application. Note that this still preserves the most important goal of using LEASES: if the client crashes, the CLIENT PROXY will also fail, and no further LEASE extensions will be requested.

If multiple clients use the same remote object, reference counting [Hen01] helps to keep track of the individual leases. When a lease is granted, the reference count is increased; when a lease expires, the reference count is decreased.

The LEASING pattern is documented in more detail in [KJ04].

Passivation

The server application provides stateful remote objects such as CLIENT-DEPENDENT INSTANCES or STATIC INSTANCES.



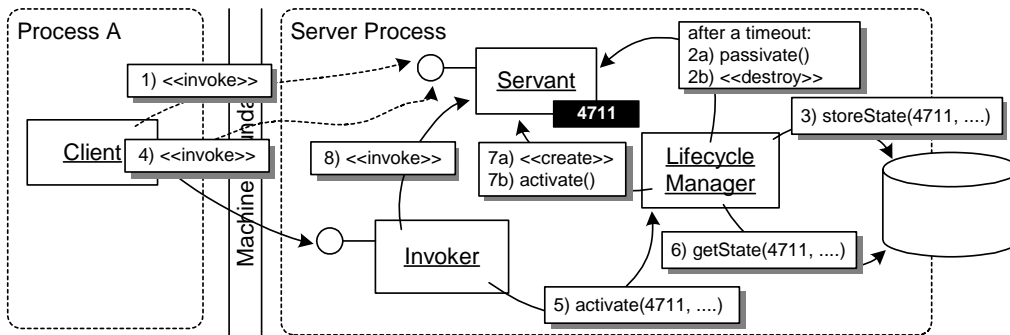
Remote objects may not be accessed by a client for a long time. Even if remote objects are not used, their servants they still occupy server resources. As a consequence, the server application uses more resources than actually necessary at a certain point in time. In systems with large numbers of remote objects this can compromise performance and stability.

Neither the server application nor the distributed object middleware know when clients will access a remote object. From a client's perspective, the remote object must be available to handle invocations all the time.

However, keeping stateful remote objects, such as CLIENT-DEPENDENT INSTANCES or STATIC INSTANCES, active even when clients do not invoke operations for longer periods of time, consumes server resources. Occupied resources lead to more contention, which degrades server application performance, scalability, and ultimately stability.

Therefore:

Temporarily remove an unused servant from memory when it has not been accessed by clients for a predefined period of time; this process is called PASSIVATION. If the INVOKER receives an invocation for a remote object that has been passivated, the LIFECYCLE MANAGER creates a new servant for the remote object before dispatching the invocation. In case of a stateful remote object, the state of the remote object is stored in a persistent storage during PASSIVATION. During recreation, the LIFECYCLE MANAGER initializes the newly created servant with the persisted state.



A remote object is passivated after an invocation and its state stored to a persistent storage. Later in time, a subsequent invocation arrives for that object, and the INVOKER informs the LIFECYCLE MANAGER. The LIFECYCLE MANAGER creates a new servant for the remote object, and then the INVOKER can perform the invocation on that new servant.



The PASSIVATION pattern typically requires *Lifecycle Callback* operations [VSW02] to be implemented by the remote object's servant, such as *activate()* and *passivate()*. The operations are used by the LIFECYCLE MANAGER to notify the servant before it is passivated. This way the servant can release used resources. A similar notification is necessary right after the servant has been reactivated in order to re-acquire the previously released resources.

The remote object has to persist its state while it is passivated. It can either do it on its own in the context of the *Lifecycle Callback* operations just mentioned. Alternatively, the LIFECYCLE MANAGER can automatically persist and read the state of its objects to and from a persistent storage.

PASSIVATION usually involves a significant overhead, comparable to paging in operating systems. The need to passivate should therefore be avoided whenever possible. To achieve this, the server should have enough memory to keep all servants active, at least for relatively short-lived instances. Heavy PASSIVATION is usually a clear indication of insufficient server resources.

This pattern has been described in the context of server component infrastructures in [VSW02].

Interactions among the Patterns

This third and last section details the relationships among the patterns presented in this chapter. We will first show which of the four patterns described in the second half of this chapter can be usefully combined with the three basic activation strategies. We will then provide typical best practices of such combinations. The following table presents an overview.

	STATIC INSTANCE	PER-REQUEST INSTANCE	CLIENT-DEPEN- DENT INSTANCE
LAZY ACQUISITION	<i>useful</i>	<i>implicitly useful</i>	<i>implicitly useful</i>
POOLING	<i>not useful</i>	<i>very useful</i>	<i>useful</i>
LEASING	<i>sometimes useful</i>	<i>not useful</i>	<i>very useful</i>
PASSIVATION	<i>sometimes useful</i>	<i>not useful</i>	<i>very useful</i>

Let's discuss these combinations. Instead of eagerly instantiating a STATIC INSTANCE servant right at program startup, you can actually wait until a client attempts to invoke an operation on the object - using LAZY ACQUISITION. This does not change anything about the fact that the number of instances is predetermined, for example, by the presence of physical devices. But acquisition of resources can be postponed until they are actually needed. The caveat is, though, that the first invocation on the instance takes longer than the subsequent ones, which can be problematic for instance in the context of embedded or real-time systems.

POOLING is typically not very useful for STATIC INSTANCES because in most cases STATIC INSTANCES represent physical entities or similar concepts with their own identity and state. LEASING might be useful to remove a STATIC INSTANCE when no client is interested in using this remote object anymore. PASSIVATION can be used to temporarily evict servants until they are used again. Of course, this scarifies some of the predictability of access times for STATIC INSTANCES, because an invocation will take longer in case a new servant has to be created.

For pure PER-REQUEST INSTANCES, LAZY ACQUISITION is implicitly used. For each request arriving at the server application a new servant is lazily created just in time. LEASING does not make sense at all for PER-REQUEST INSTANCES, because the servant is destroyed right after

handling an invocation. PER-REQUEST INSTANCES are suited ideally for optimization through POOLING. As PER-REQUEST INSTANCES are always stateless, a servant pool can easily be created, taking an instance from the pool to serve each invocation. PASSIVATION is not useful, since there is no state.

If many CLIENT-DEPENDENT INSTANCES are created *and* destroyed by clients, it makes sense to use POOLING and to initialize the servants with the parameters supplied by the client, instead of creating new servants all the time. Regarding LEASING, CLIENT-DEPENDENT INSTANCES are the primary use case for the pattern. As outlined in the LEASING pattern, CLIENT-DEPENDENT INSTANCES are typically implemented by using LEASING to make sure the server application has a way to reclaim remote objects and their servants once they are not accessed anymore but have not been explicitly destroyed, for example as a consequence of a crashed client. Since CLIENT-DEPENDENT INSTANCES are typically stateful, PASSIVATION is useful. Support of PASSIVATION is important since the state must stay available even when the client is offline for an extended period of time. In such scenarios, PASSIVATION of CLIENT-DEPENDENT INSTANCES can reduce a server's resource usage significantly.

Lifecycle Management Best Practices

Embedded and real-time applications typically rely on STATIC INSTANCES. This is because the remote objects often resemble physical devices (or functionality associated with them), and the configuration is pretty static. Thus, there is no need to use techniques that mainly address dynamic resource management scenarios, such as LAZY ACQUISITION and PASSIVATION. The static configuration allows to ensure that enough resources are available and that they can be acquired at server application startup.

While the number of clients in embedded and real-time applications is often known in advance, the number of clients is usually not known in business applications. Business applications are often built using server-side component technologies, such as Enterprise Java Beans (EJB, see [Sun03a]), CORBA Components (CCM, see [Sev03]), or Microsoft's COM+ [Mic02]. Thus, let's have a brief look at how component containers manage the lifecycles of their components.

Containers help the developer to focus on functional concerns when implementing components, leaving the management of technical concerns to the container (for details see [VSW02]). All major component containers, available today, support different kind of components, featuring different lifecycle models. The selection of lifecycle models implemented by component containers can be seen as best practices because they intend to cover the typical requirements of most developers.

All component containers provide so-called *Service Components* [VSW02]. These are stateless components whose lifecycle is typically implemented by PER-REQUEST INSTANCES, optimized using POOLING.

Another kind of components are *Session Components* [VSW02]. They are basically CLIENT-DEPENDENT INSTANCES that provide no support for concurrent access because it is assumed that only one client accesses a specific instance at a time. They are created on the client's request and often have LEASES associated with them. LEASING is a two step process here: when an instance is not accessed for a specific period of time, the instance is passivated, which means its state is written to persistent storage and the servant itself is destroyed. When a new request arrives for such a passivated instance, a new servant is lazily created and its state restored from disk. PASSIVATION helps to save the resources of those instances that are not accessed for a while. Only after another, longer period of time, the remote object (including its state) is actually destroyed and cannot be reactivated.

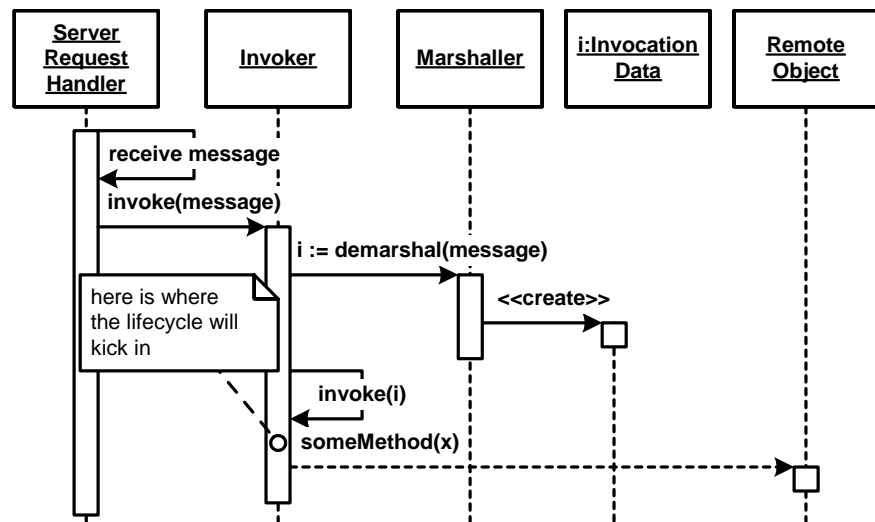
Entity Components [VSW02] are typical examples of stateless remote objects that use POOLING to optimize resource consumption. However, from a client's perspective, *Entity Components* have state that represents business entities (such as customers, orders, products). The state of these business entities is kept in a persistent storage and transferred to and from pooled servants by the LIFECYCLE MANAGER of the container. As with all stateless pooled remote objects, LEASING is not necessary (the servants stay in the pool, anyway, and the state is persistent). In case of entity components, however, destroying an instance means to delete the persistent state of an instance. This is therefore a business operation and not a means to technically manage resources.

Note that, as mentioned before, these more elaborate lifecycle management strategies are typically the domain of component containers.

These provide a lifecycle management framework that also deals with advanced technical aspects such as transactions, security, and failover. More details on component infrastructures can be found in [VSW02] (see also the Chapter *Related Concepts, Technologies, and Patterns* and the Appendix).

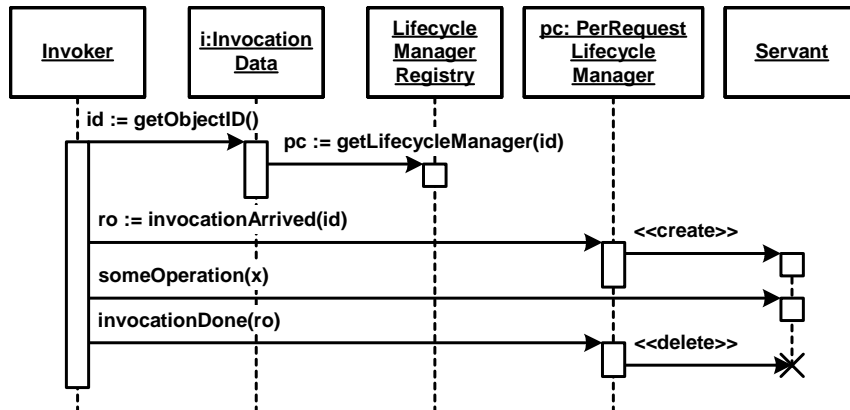
Lifecycle Interaction Sequences

This section gives some examples of typical lifecycles. In all the following diagrams we assume that the invocation data is demarshalled and an object containing the invocation data is already instantiated. To illustrate these tasks, the following sequence diagram shows the steps that we assume to have already happened in all the subsequent diagrams. The INVOKER triggers demarshalling and receives the newly created invocation data object as a result. This object is used for further tasks like lifecycle management, as indicated in the diagram. After these tasks have happened, the actual invocation of the remote object takes place.



In the following examples, we also assume that there is only one INVOKER in the system. It manages a set of LIFECYCLE MANAGERS that implement different activation strategies.

The following sequence diagram shows a simple, non-pooled PER-REQUEST INSTANCE.

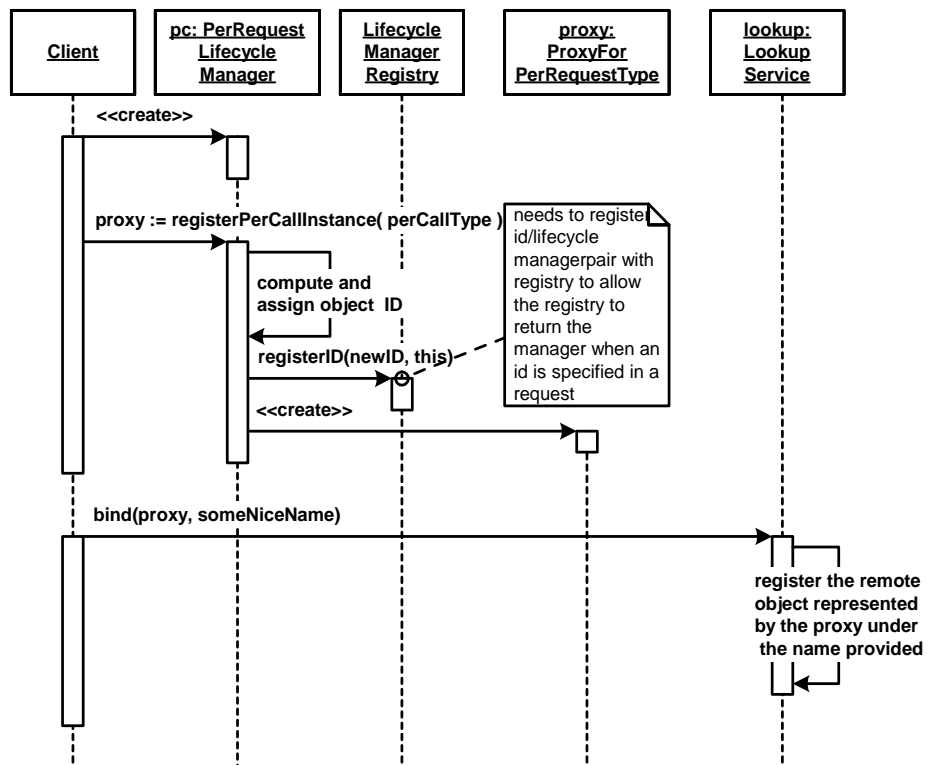


The LIFECYCLE MANAGER implements the *Strategy* pattern (see [GHJV95]). All concrete LIFECYCLE MANAGERS support the same interface, but have different implementations. The *LifecycleManagerRegistry* class keeps track of all the LIFECYCLE MANAGERS in the system, as well as their relationship to remote objects and their servants.

To associate a remote object with a specific LIFECYCLE MANAGER, a well-defined registration procedure must be obeyed. In the example case above, we will not register a specific remote object instance with the LIFECYCLE MANAGER, but rather a specific *type*. The LIFECYCLE MANAGER realizes PER-REQUEST INSTANCES. That is, it instantiates the remote object type for each request by creating a new servant.

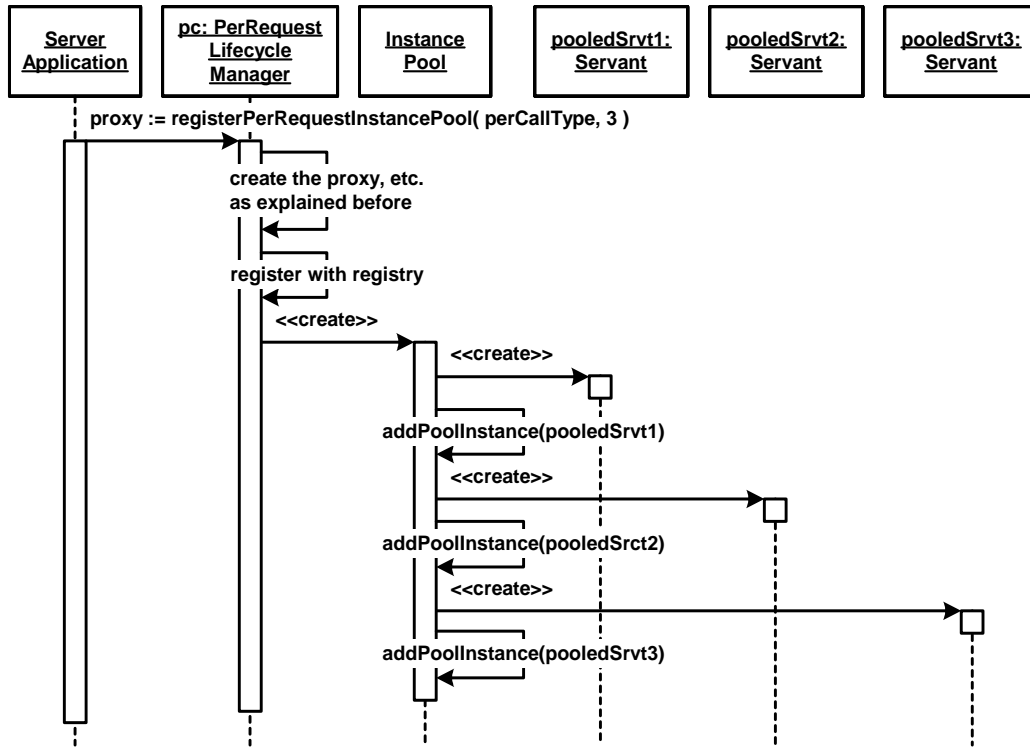
The process of registering a remote object with its LIFECYCLE MANAGER depends on the strategy implemented by the LIFECYCLE MANAGER. In the other two lifecycle strategy patterns, STATIC INSTANCE and CLIENT-DEPENDENT INSTANCE, concrete remote object instances are registered with the LIFECYCLE MANAGER. It is therefore a good idea to let the LIFECYCLE MANAGER handle the registration process. Internally, the LIFECYCLE MANAGER updates the *LifecycleManagerRegistry*. In this example, the registry must support both registration of particular objects (for STATIC INSTANCES and CLIENT-DEPENDENT INSTANCES) and of object types (for PER-REQUEST INSTANCES), of course. This can be realized by registering the OBJECT ID of the remote object rather than the individual servants. The different LIFECYCLE MANAGERS map the OBJECT IDS to servants, when an invocation arrives.

The next diagram shows an example of how to simplify the management tasks of PER-REQUEST INSTANCES, such as registering them in the lookup service. Here, the registration operation of the LIFECYCLE MANAGER returns a proxy for the PER-REQUEST INSTANCE. This proxy is used inside the server application, when other components refer to the PER-REQUEST INSTANCES. Thus it is possible to interact with PER-REQUEST INSTANCE as ordinary objects even though no servant is yet instantiated or multiple servants are instantiated at the same time. The following sequence diagram shows how all this could look like.



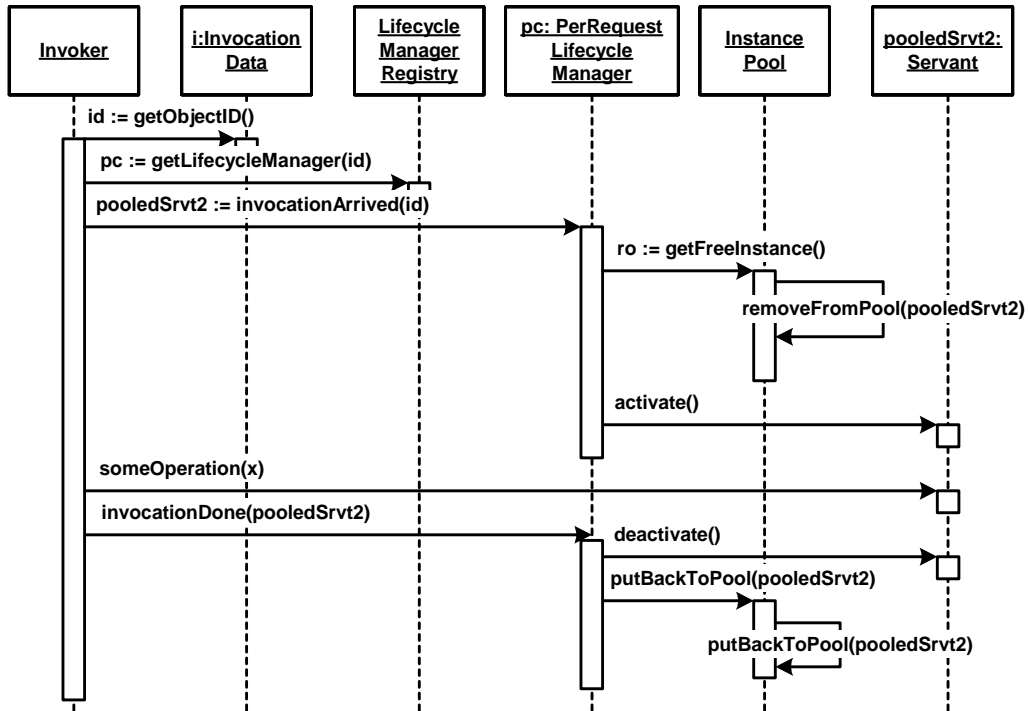
Back to the activation strategies. As explained above, PER-REQUEST INSTANCES are typically optimized by using POOLING. The following sequence diagrams shows how this could be implemented. The first one shows the pool creation process. First, a PER-REQUEST INSTANCE is registered with the LIFECYCLE MANAGER. This triggers creating an

instance pool. The pool in turn instantiates a number of servants as pooled instances.



The following diagram shows how an PER-REQUEST INSTANCE is actually accessed, in case an invocation arrives for it. The LIFECYCLE MANAGER is informed that the invocation has arrived. Next, it looks up an idle servant in the instance pool and returns it. The instance is then

used to execute the invocation. Finally, the instance is put back into the pool again.



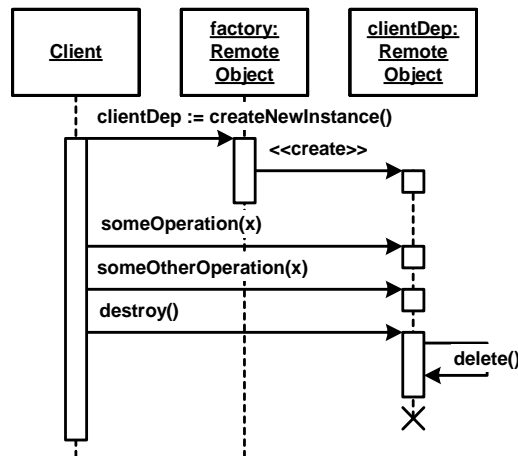
Note how the LIFECYCLE MANAGER uses the *activate()* Lifecycle Operation [VSW02] to allow the servant to do some initialization when it is taken from the pool, for example, to acquire resources. The reverse is done in *deactivate()* that is called after the invocation has been handled, and just before the servant is put back to the pool.

We do not illustrate STATIC INSTANCES for two reasons. First, a STATIC INSTANCE that is not lazily instantiated is trivial and implicitly illustrated with many other examples given in the chapters before: the server application simply instantiates the remote object's servant and registers it with a LIFECYCLE MANAGER for STATIC INSTANCES which just remembers the servant and forwards invocations to it. In case LAZY ACQUISITION is used, the process is basically the same as with the PER-REQUEST INSTANCE example shown above, except that the single created servant is not destroyed and serves subsequent invocations. Specifi-

cally it also requires a type-based registration and a proxy, as in the example illustrated above.

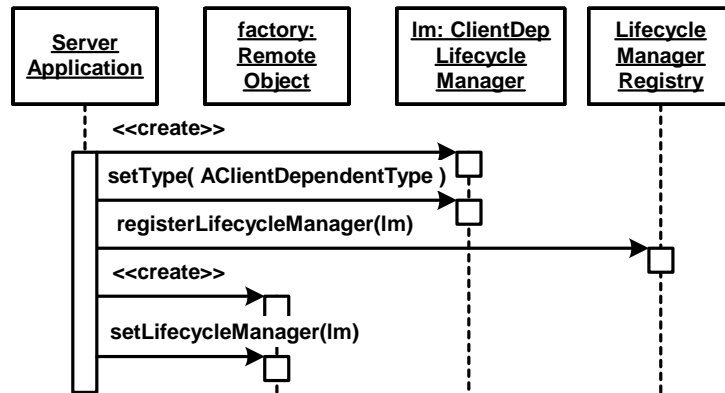
More examples of POOLING, especially POOLING of persistent, stateful remote objects are shown at the end of the *Extended Infrastructure Patterns* chapter to illustrate the LIFECYCLE MANAGER and CONFIGURATION GROUP patterns.

The following diagrams illustrate CLIENT-DEPENDENT INSTANCES in some more detail. The next diagram shows how a client accesses a CLIENT-DEPENDENT INSTANCE, using a factory remote object. Here the factory is used to create the CLIENT-DEPENDENT INSTANCE, and the instance is explicitly destroyed by the client after a number of invocations.

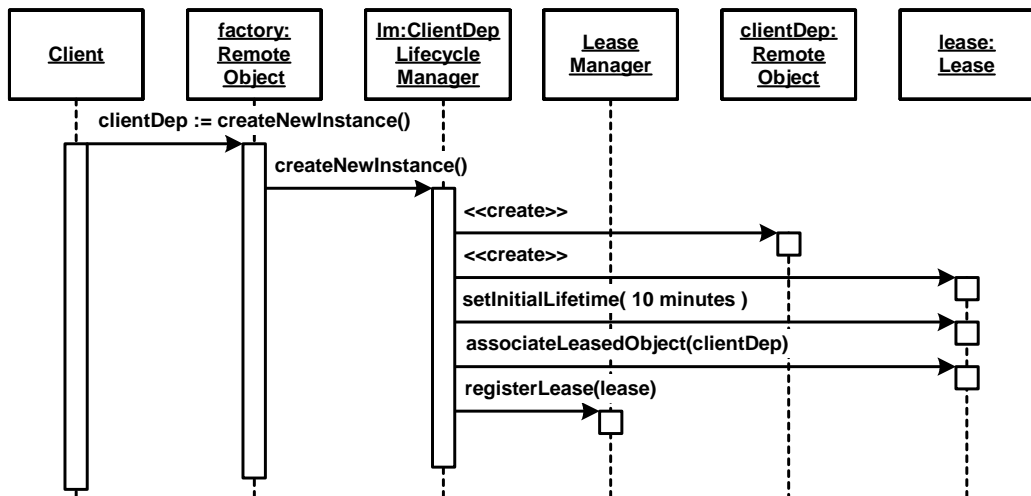


The more interesting tasks happen under the hood of the distributed object middleware, of course. The next sequence diagrams tries to illustrate this. First of all, we have to initialize the respective LIFECYCLE

MANAGER and create the factory. This job is done by the server application, as the next sequence diagram shows.



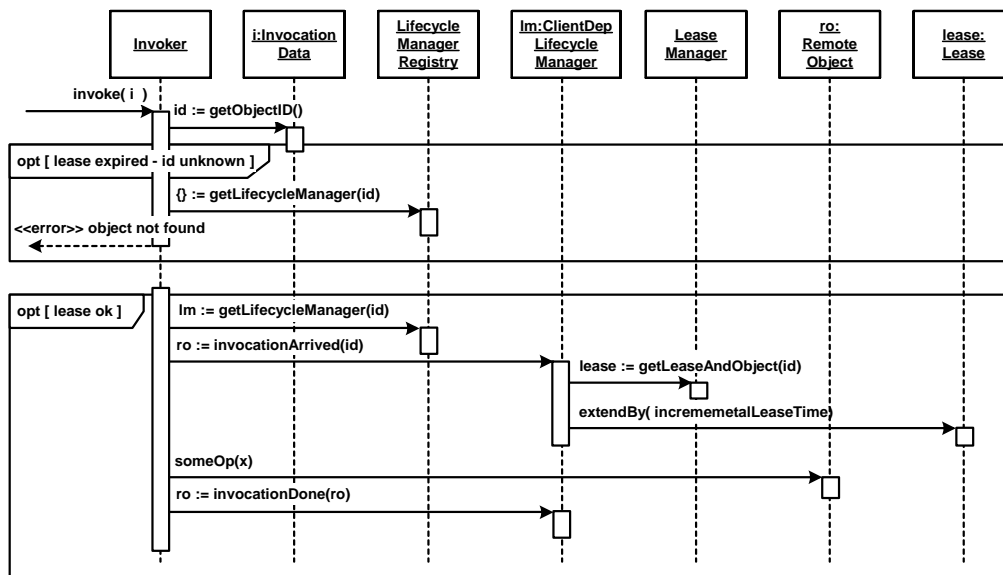
An important aspect for CLIENT-DEPENDENT INSTANCES is the management of LEASES - so that remote object instances that are no longer needed are automatically removed. The diagram below shows how a client requests the creation of a new CLIENT-DEPENDENT INSTANCE. The LIFECYCLE MANAGER keeps track of the instance and the lease. The lease is configured with the lease time and the leased object instance. The configured lease is registered with a lease manager that keeps track of the instantiated leases.



The next illustration addresses the issue of invoking an operation of a CLIENT-DEPENDENT INSTANCE with leases. In this case, the INVOKER and the associated LIFECYCLE MANAGER have to make sure the lease is still valid before the object can be invoked. There are two scenarios what might happen:

- The OBJECT ID is unknown to the lifecycle manager, for instance, because its lease has expired and it has been removed. Then the invocation results in a REMOTING ERROR.
- The lease is ok and the object can be found. Then the lease is renewed and the invocation is performed on the leased instance.

We show these two cases in the following diagram.



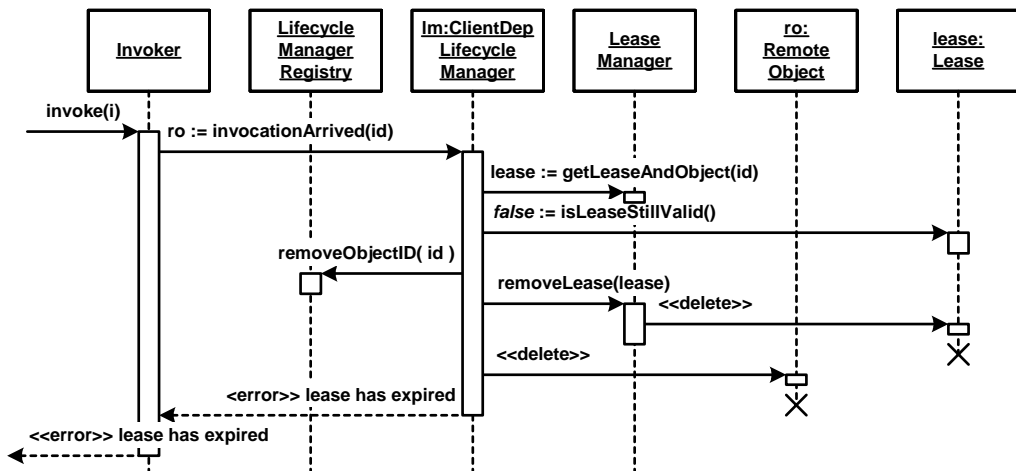
The next diagram shows how lease expiration is handled. Leases are handed out on a per client and remote object basis. There exist several potential approaches:

- One approach is to check lease validity whenever a request arrives. If the lease has expired in the meantime, the remote object is deactivated and the lease deleted. An error is reported to the client.

- Another approach is to have a background thread check all leases in turn, all the time, and delete those remote objects and leases that are not valid any more.
- A third alternative that is often used in event-based systems is to register a callback function for the point in time when a remote object's lease is expected to expire. The callback function can then check if the remote object can be deactivated or not.

The first approach seems simplest, but it has a serious drawback: if a client does not regularly invoke the remote object, the system will not notice the lease expiration, and the remote object will not get deactivated. As a consequence, one of the other two approaches is necessary in all cases. However, it can still be useful to combine on-request-checks with a background thread solution, as it avoids situations where the lease has expired, but the background thread has not run, yet.

The figure below shows the first alternative. An invocation arrives and the lease is obtained. However, this lease is expired. As a reaction, the remote object instance, the lease instance, and the registration of the remote object in the lifecycle manager registry are deleted. A REMOTING ERROR is reported to the client.

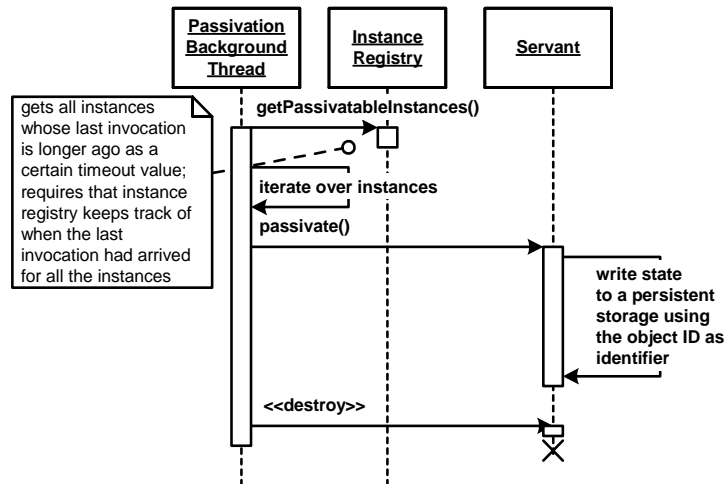


The background thread variant is rather simple and therefore not shown; it just scans all the LEASES managed by the LEASE manager and if one expires, it executes the same deactivation steps as shown above.

In this example, the background thread and the lease manager are integral parts of the CLIENT-DEPENDENT INSTANCE'S LIFECYCLE MANAGER.

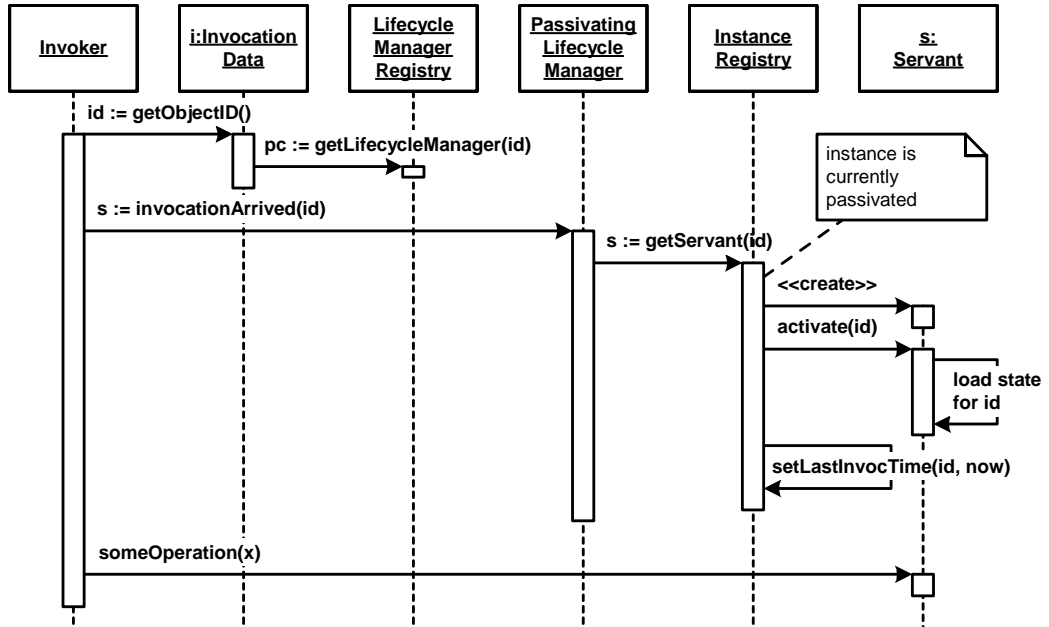
The third variant using an event-based callback needs to register the callback during creation. All deactivation steps, shown above, need to be performed by the callback.

The next sequence diagrams show how PASSIVATION works. A background thread checks for instances whose PASSIVATION timeout has been reached, and must be PASSIVATED. The instance is passivated by a *passivate()* operation and then destroyed. Again, this variant can also be implemented using an event-system-based callback that is executed recurringly after a specific period of time.



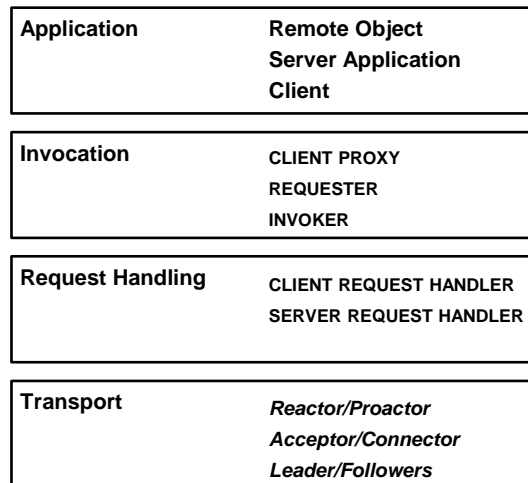
The next and final diagram shows how the LIFECYCLE MANAGER reactivates an previously passivated remote object once a new invocation for the remote object arrives. First an "empty" instance is created and then

it is filled with the state from the database during the *activate()* operation. The INVOKER performs the invocation on this new instance.



10 Extension Patterns

The message processing in distributed object middleware follows a *Layers* [BMR+96] architecture as shown in the illustration below. The layers architecture partitions and encapsulates responsibilities that depend on each other. Each layer interacts only with the previous and next layer respectively.



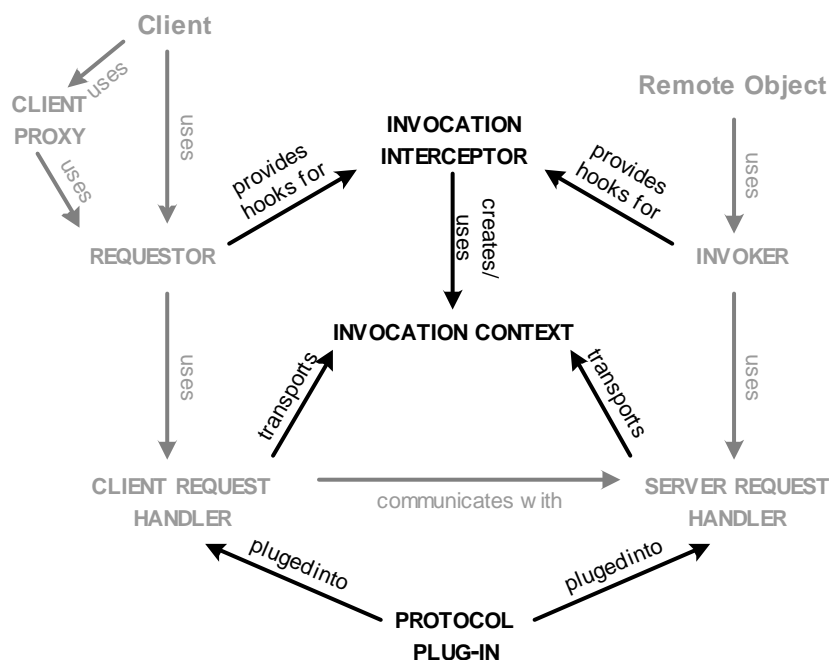
The following layers can be identified:

- **Application** – The top-level layer of the message processing architecture of a distributed object middleware consists of clients that perform invocations and remote objects that serve invocations.
- **Invocation** – Underneath the application layer, CLIENT PROXY, REQUESTOR, and INVOKER are responsible for marshalling/demarshalling and multiplexing/demultiplexing of messages.
- **Messaging** – At the next layer, the REQUESTOR uses a CLIENT REQUEST HANDLER, and the INVOKER is served by a SERVER REQUEST HANDLER. This layer is responsible for the basic tasks of establishing connections and message passing between client and server.

- **Communication**—The communication layer is responsible for defining the basic message flow and manages the operating system resources, such as connections, handles, or threads. Depending on the concrete architecture, this layer uses the *Reactor* [SSRB00] pattern and/or *Acceptor/Connector* [SSRB00] pattern.

Each layer only depends on the layers next to it. In order to introduce new services, layers might have to be extended. Depending on the purpose of an extension, the extension can be made at any layer, on both the client and the server side. Typical extensions are the introduction of security, transaction, or logging support of remote invocations. For example, security concerns such as authorization are typically introduced by extending the invocation layer, so that the extended behavior is transparent for the application. Other security concerns, such as using a secure transport protocol, require modifications at the request handling and communication layer.

In this chapter, we present patterns for extending the *Layers* architecture of message processing in a distributed object middleware. Note that there is a certain symmetry between client and server in the layered architecture. This is because the processing patterns at each layer depend with their remote counterpart. The same is true for many typical extension tasks: they also have to be performed both on client side and server side. Just consider authorization as a possible security extension. The client would have to provide the user credentials, and the server side would have to verify those credentials. Thus, both sides have to be extended in parallel to support authorization.



INVOCATION INTERCEPTORS extend message processing by intercepting a message before and/or after it passes a layer. They can be configured at almost every level, even though typically they are used in the REQUESTOR and INVOKER. The granularity can be per server application, per remote object, or per CONFIGURATION GROUP. Other combinations are also possible, but less common.

To handle concerns such as transactions or security, invocations need to contain more information than just the operation name and its parameters; for example, some kind of transaction ID or security credentials. For that purpose INVOCATION CONTEXTS are transparently exchanged between CLIENT REQUEST HANDLER and SERVER REQUEST HANDLER. INVOCATION INTERCEPTORS on client side and server side may create and consume INVOCATION CONTEXT data to communicate with the counterpart on the remote side. INVOCATION INTERCEPTORS are apply the *Interceptor* [SSRB00] pattern to remote communication.

A PROTOCOL PLUG-IN can be used by developers to exchange or adapt the transport protocol. Such plug-ins are plugged into the CLIENT

REQUEST HANDLER and SERVER REQUEST HANDLER. This can be used for optimizing the communication protocols, used for particular applications, or supporting new protocols that were formerly not supported.

Invocation Interceptor

Applications need to transparently integrate add-on services.



In addition to hosting remote objects, the client and server application often have to provide a number of add-on services, such as transactions, logging, or security. The clients and remote objects themselves should be independent of those services.

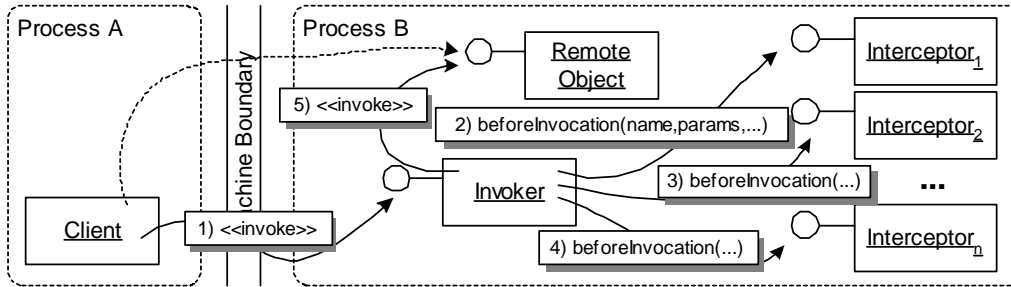
Consider the typical concern of security in distributed object middleware: remote objects need to be protected from unauthorized access. The remote objects themselves should not have to worry about authorization; they should actually be accessible with or without the security concern “authorization” enabled. While the server application needs to enforce security, the client needs to add credentials, such as user name and password, to the request.

In addition to security, there are many other concerns that potentially might have to be added to the invocation path, such as transactions, logging, persistence, etc. Since those concerns are add-on services, clients and remote object implementations should not depend on these features. It should be possible to selectively enable and disable them without changing their implementations.

Therefore:

Provide hooks in the invocation path, for example in the INVOKER and REQUESTOR, to plug in INVOCATION INTERCEPTORS. INVOCATION INTERCEPTORS are invoked before and after request and response messages pass the hook. Provide the interceptor with all the necessary information to allow it to provide meaningful add-on services,

such as operation name, parameters, OBJECT ID, and, if used, the INVOCATION CONTEXT.



The INVOKER receives an invocation and checks whether INVOCATION INTERCEPTORS are registered for that remote object. If so, the interceptor is invoked before and after the actual invocation.



INVOCATION INTERCEPTORS allow for the transparent integration of additional orthogonal services, and concerns. Those additional services are independent of the individual clients or remote objects.

If more than one INVOCATION INTERCEPTOR is applicable for an invocation, the interceptors are often interdependent. Therefore, in most cases, they are arranged in a *Chain of Responsibility* [GHJV95]. Each INVOCATION INTERCEPTOR forwards the invocation to the next interceptor in the chain. Alternatively, the components that provide the hooks, such as REQUESTOR and INVOKER, can manage passing information between elements of the chain.

An INVOCATION INTERCEPTOR can use the invocation data to do whatever it wants. The INVOCATION INTERCEPTOR might just read the passed parameters, modify the parameters, or even interrupt further processing and report a REMOTING ERROR to the client. Note that not all distributed object middleware permit modification of all the invocation data.

INVOCATION INTERCEPTORS are typically configured via external configuration files or via programming interfaces. External configuration has the advantage that they can be modified without recompilation. A

programmatic interface has the advantage that INVOCATION INTERCEPTORS can be dynamically added and removed at runtime.

INVOCATION INTERCEPTORS can only access the information that is available at the abstraction layer where their hook is located. For example in the CLIENT REQUEST HANDLER and SERVER REQUEST HANDLER, only marshalled information in byte stream format is available, whereas in hooks in the REQUESTOR/INVOKER layer, individual data items, such as operation name and parameters, are directly accessible. Based on that difference in the supplied information, the use cases are different. For example, encryption is typically handled at the request handling layer, as this task operates on byte streams and not individual data items. On the other hand, logging will typically be implemented on the REQUESTOR/INVOKER level, since the data items that should be logged are available in their native format.

If the interceptor requires more information than is usually available in the invocation message, an INVOCATION CONTEXT can be used to transport additional data from client to server (and back). A typical design is to provide an INVOCATION INTERCEPTOR in the client that adds a particular information to the INVOCATION CONTEXT, and one corresponding INVOCATION INTERCEPTOR in the server that consumes this information, providing some service using this data.

INVOCATION INTERCEPTORS consume additional resources and add complexity to the overall architecture. Further, entities, such as the INVOKER, can no longer rely on what they receive or send, as parameters might have been modified by intermediate INVOCATION INTERCEPTORS. If safety is a primary concern, this can become problematic and therefore requires proper control, for example via programming guidelines.

The concerns addressed by INVOCATION INTERCEPTORS can be also solved using aspect-oriented solutions [KLM+97]. Such solutions are specifically interesting, if no hooks are available in the distributed object middleware. General purpose aspect languages, such as AspectJ [KHH+01], can be used to transparently integrate the same concerns, that INVOCATION INTERCEPTORS are typically used for. For an discussion of aspect-oriented programming (AOP) in the context of remoting check the *Related Concepts, Technologies, and Patterns* chapter and the *Appendix*.

The INVOCATION INTERCEPTOR is a specialization of *Interceptor* [SSRB00] to distributed object middleware.

Invocation Context

Add-on services are plugged into the distributed object middleware.



Remote invocations typically only contain the necessary information, such as operation name, OBJECT ID, and parameters. But INVOCATION INTERCEPTORS often need additional information in order to properly provide add-on services. A straight-forward solution would be to add this information to the operation signature of remote objects. But this would prevent the transparent integration of the add-on services, which is the goal in the first place, as signatures would change depending on the requirements of the add-on services. Changing operation signatures for other reasons than business logic is tedious and error-prone.

Clients as well as server applications often have to provide add-on services transparently. For this purpose, they often have to share contextual information, such as a transaction ID or security credentials. If this information changes on either side, it needs to be transported to the other side. In case of transactions, the transaction ID has to be transported between clients and servers for every remote invocation so that both can participate on the same transaction.

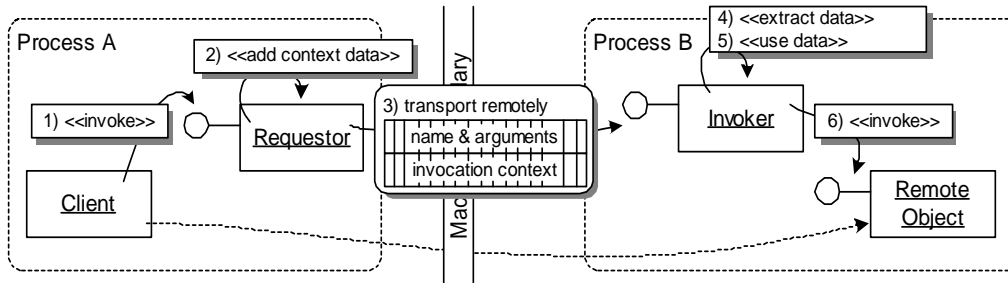
The operation signature of remote objects should not be touched in order to transport this additional information. Not only because it is tedious to maintain, but also because the contextual information is not always necessary, since the add-on service may be optional and can be turned on and off. If the contextual information would be part of the operation signature, it would waste precious network bandwidth in case the add-on service is “turned off”.

In general, the contextual information required by add-on services cannot not be anticipated by either the distributed object middleware itself, or by remote object developers; the contextual information needs to be extensible independently.

Therefore:

Bundle contextual information in an extensible INVOCATION CONTEXT data structure that is transferred between client and remote

object with every remote invocation. For transparent integration INVOCATION INTERCEPTORS can be used to add and consume this information. The format and data types used for the contextual information in INVOCATION CONTEXT depends on the use case.



The client invokes an operation. The REQUESTOR adds the required contextual information in the INVOCATION CONTEXT. The INVOKER extracts the information, uses it, and invokes the operation of the remote object. Typically, the information is added and used by (application-specific) INVOCATION INTERCEPTORS.



Insertion, extraction, and use of INVOCATION CONTEXT information is hidden inside distributed object middleware components, specifically INVOCATION INTERCEPTORS, the REQUESTOR or the INVOKER. The application logic of clients and remote objects stays independent of INVOCATION CONTEXT information, which works well because in most cases the information in the INVOCATION CONTEXT is only of interest to the add-on service. For situations, where the remote object needs access to INVOCATION CONTEXT data, the distributed object middleware provides a well-defined API.

The data in the INVOCATION CONTEXT has to be marshalled just like every other information contained in an invocation. As a consequence, the same limitations apply as for operations parameters: the used MARSHALLER must support the used data types.

The client- and server-side components that make use of the INVOCATION CONTEXT information, must adhere to the same protocol, which means they must agree on a common data structure inside the INVOCATION CONTEXT.

- | Using INVOCATION CONTEXTS, add-on services, such as transaction support or security, can communicate transparently, that is, without the explicit support of clients and remote objects. The remote objects are kept independent of any contextual data transferred. While INVOCATION CONTEXTS provide an useful and flexible extension mechanism, using them increases the footprint of every remote invocation they are added to.

Protocol Plug-In

CLIENT REQUEST HANDLER and a SERVER REQUEST HANDLER adhere to the same communication protocol.



Developers of clients and server applications often need to control the communication protocol used by the CLIENT REQUEST HANDLER and SERVER REQUEST HANDLER. In some cases, differences in network technologies require the use of specialized protocols, in other cases, existing protocols need to be optimized in order to meet realtime constraints. Sometimes it might be even necessary to support multiple protocols at the same time. The communication protocols should be configurable by developers, who might only have a limited knowledge of low-level protocol details.

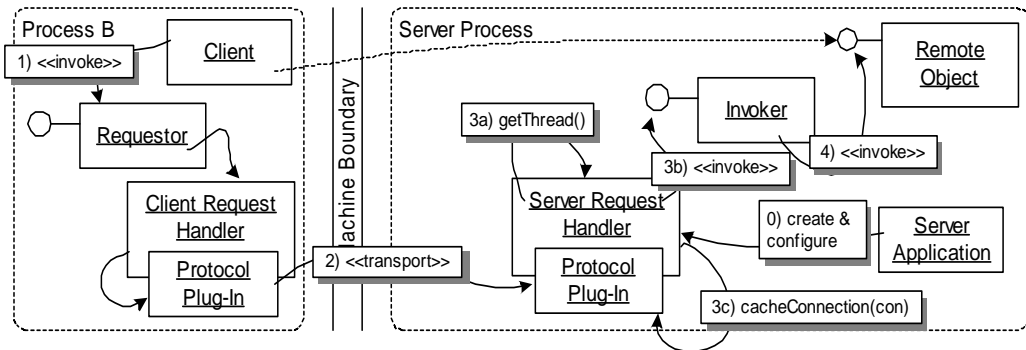
There are several situations where a developer of a distributed application cannot abstract from the implementation details of a distributed object middleware at the communication protocol layer and needs to explicitly control the communication protocol used:

- Specialized network technologies require specialized communication protocols. If multiple network adapters have to be used by the same client or server, each adapter might need to be accessed using a different communication protocol.
- Specialized custom MARSHALLERS that provide optimized serialization mechanisms, might require specialized communication protocols to be used to actually transport the serialized data.
- The distributed application needs to fulfil varying QoS requirements. To effectively fulfil these QoS requirements, the facilities provided by the communication protocol have to be used differently and perhaps need to be optimized at a low level.
- The implementation strategies of the communication protocol, such as the handling of priorities, need to be optimized for the most common use case of the distributed application in order to achieve the desired predictability.
- Firewalls might prohibit the use of default communication protocols. Specialized communication protocols are required that are

able to pass firewalls. Adaptations to the new protocol may be necessary to cope with differences in the network environment.

Therefore:

Provide PROTOCOL PLUG-INS to extend CLIENT REQUEST HANDLERS and SERVER REQUEST HANDLERS. Let the PROTOCOL PLUG-INS provide a common interface to allow them to be configured from higher layers of the distributed object middleware.



The PROTOCOL PLUG-INS are created, configured, and used inside the CLIENT REQUEST HANDLER and SERVER REQUEST HANDLER, respectively. In heterogeneous environments the CLIENT REQUEST HANDLER selects the plug-in based on the properties and supported communication protocol of SERVER REQUEST HANDLER. For optimization, connections may be cached using PROTOCOL PLUG-INS.

To ensure interoperability, the CLIENT REQUEST HANDLER and SERVER REQUEST HANDLER need to support the same communication protocol and the same or at least compatible PROTOCOL PLUG-INS.

PROTOCOL PLUG-INS can be integrated by using INVOCATION INTERCEPTORS in a *Chain of Responsibility* [GHJV95]. Architectures that follow this design, such as ART [Vin02b] or Apache Axis (see *Web Services Technology Projection* Chapter), distinguish several kinds of INVOCATION INTERCEPTORS, for instance: transports interceptors, protocol interceptors, and general interceptors.

Some communication protocols, such as CAN [Cia04], require very special handling. For example, packet sizes are severely limited, and usually, requests and responses have to be mapped to pre-defined messages types on the CAN network. This requires close cooperation between a custom MARSHALLERS and the PROTOCOL PLUG-IN.

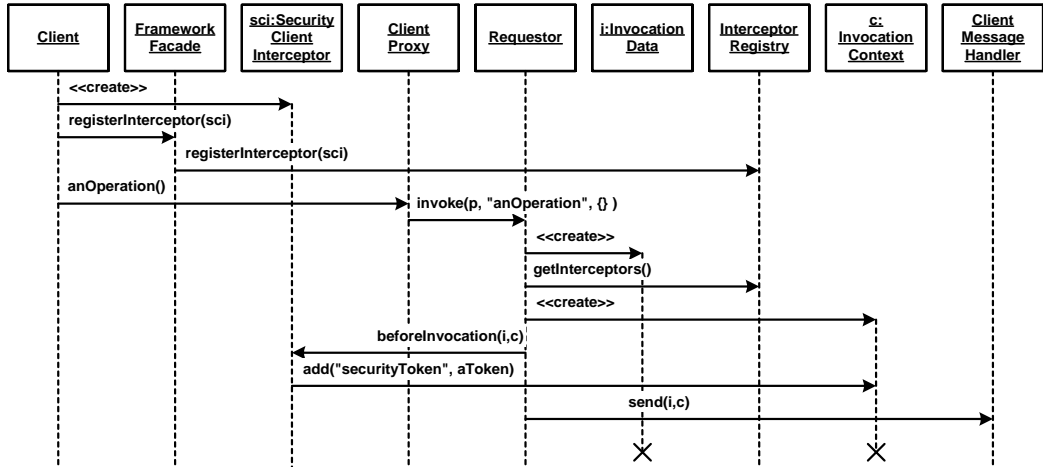
For configuration of PROTOCOL PLUG-IN parameters, the PROTOCOL PLUG-INS offer either an API or a configuration file. Typical parameters are QoS properties including various sizes and time-outs to be used by the communication protocol. In most cases those details are only customized by CLIENT REQUEST HANDLERS and SERVER REQUEST HANDLERS. It is transparent to the distributed application, which plug-in is used underneath. Though in certain cases, for example when *Reflection* [BMR+96] is supported, it can be beneficial to configure those parameters from inside the application. This allows the application to adapt to changing environments. However, most of this is topic of current research.

If more than one PROTOCOL PLUG-IN is involved in parallel, that is, if a remote object can be reached using several networks, the SERVER REQUEST HANDLER must make sure the reply message is sent using the correct protocol, or else the reply will never reach the original client.

To summarize, PROTOCOL PLUG-INS provide several advantages: they abstract from communication protocol details, they allow for flexible support of several communication protocols, and they allow for configuration and optimization of the used communication protocols.

Interactions among the Patterns

The following sequence diagram shows how a client uses an INVOCATION INTERCEPTOR to add a security token to each invocation. The security token will be used within the server application to authenticate the client.

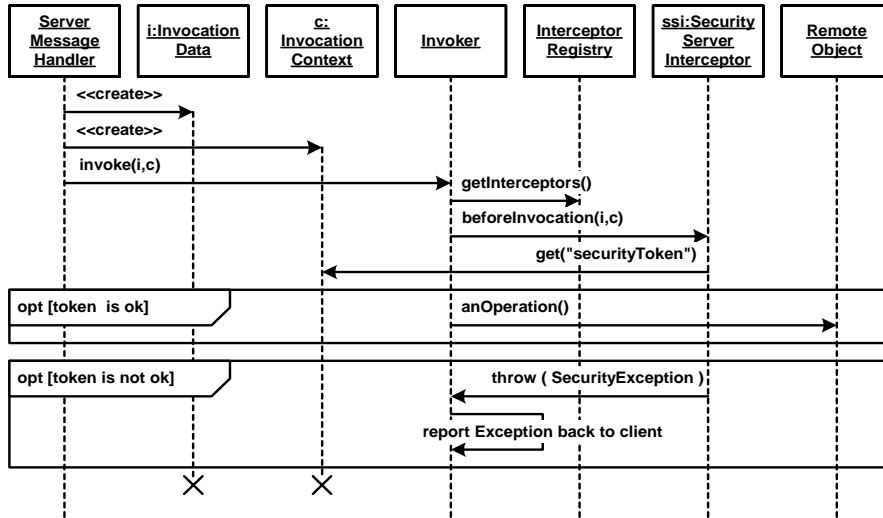


As can be seen in the diagram, the client registers a security interceptor with the framework facade (a *Facade* [GHJV95] object providing access to the distributed object middleware infrastructure) upon startup. When the client invokes an operation, the REQUESTOR uses the registered INVOCATION INTERCEPTORS to transparently add the security token to the INVOCATION CONTEXT.

The invocation itself is passed to the interceptors as an invocation data object. Potentially, the interceptors can change the information in this object and thereby manipulate the invocation. The same happens on server side.

The respective part in the server application is shown next. An INVOCATION INTERCEPTOR is used to access the security token in the INVOCATION CONTEXT. The token is used to authenticate the client, and only if access can be granted, the operation is invoked. In case the token is invalid, a security exception is raised and a REMOTING ERROR is sent back to the client. In the following sequence diagram, we assume that the interceptor has been registered by the server application upon startup - using the server-side framework facade. Here, the interceptor

decides whether the invocation can be performed or, if the security token is not valid, the invocation is rejected using a security exception.

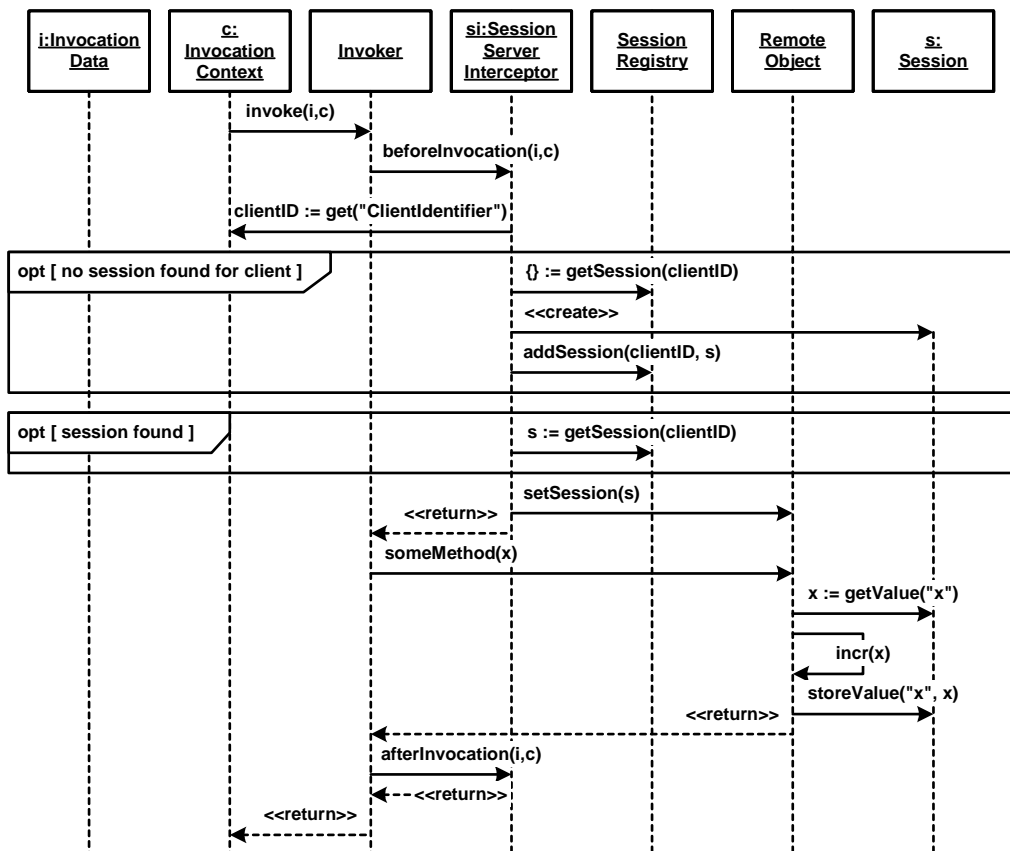


Another interesting feature that can easily be implemented using the patterns introduced in this chapter is session handling. As explained more deeply in the *Related Concepts, Technologies, and Patterns* chapter, sessions can be used to provide state to otherwise stateless remote objects. Using a client-specific session object instead of making all the remote objects stateful, CLIENT-DEPENDENT INSTANCES makes scalability, failover, and load-balancing easier to achieve - in most cases.

The example in the next diagram uses non-persistent sessions, which are only stored in memory by the *SessionRegistry*. In case persistent sessions must be supported, the respective INVOCATION INTERCEPTOR would have to make sure the session data is loaded in *beforeInvocation()* and also save potential changes in *afterInvocation()*. In the example below, *beforeInvocation()* handles the session using the session registry and *afterInvocation()* does nothing. The sequence diagram shows two session handling scenarios:

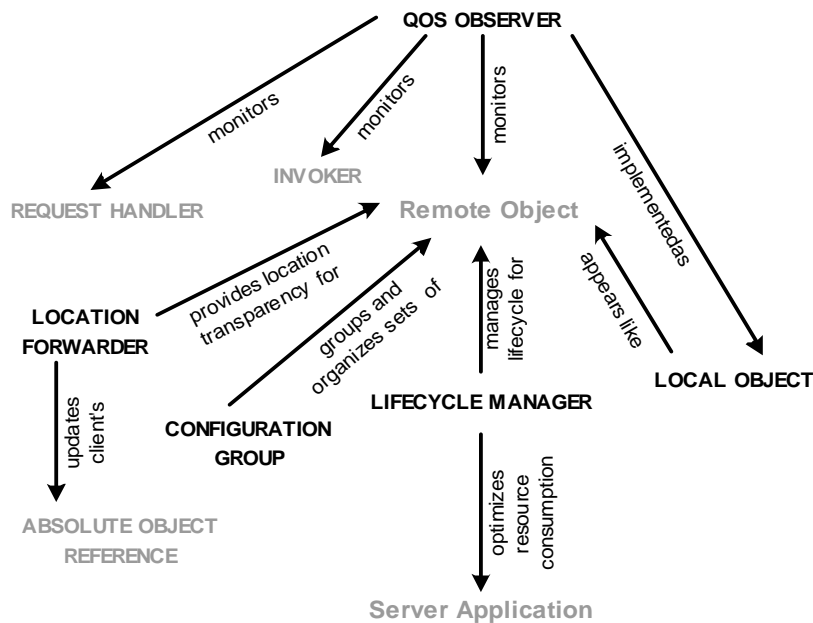
- The session is not yet existing. Then a session object is created, added to the session registry, and then used for the invocation.
- A session object is found. Then it can be obtained and used for the invocation.

In the sequence diagram, we see that the remote object instance can optionally interact with the session object. Here, we store and retrieve values from the session object. This requires the remote object to know the current session object. The INVOCATION INTERCEPTOR can be used to add this information to the invocation.



11 Extended Infrastructure Patterns

This chapter presents patterns that are used to implement advanced features of distributed object middleware. While they are part of the infrastructure they are still visible and relevant to the developer who is using a distributed object middleware.



The **LIFECYCLE MANAGER** is responsible for managing the activation, deactivation, and other more advanced lifecycle issues of remote objects. It is based on the idea of separating remote objects from their servants as introduced in the *Lifecycle Management Patterns* chapter.

CONFIGURATION GROUPS are used to configure groups of remote objects regarding their lifecycle, synchronization, and other properties.

QOS OBSERVERS are used to monitor service properties, such as performance, of various constituents of the system, such as the **INVOKER**, the

SERVER REQUEST HANDLER, or even the remote objects themselves. QOS OBSERVERS help to ensure overall quality of service (QoS) constraints by properly tuning the distributed object middleware.

The LOCATION FORWARDER takes care of automatic forwarding of invocation messages to other server applications, for example if the target remote object has moved to another server application. A LOCATION FORWARDER can also update clients by returning an ABSOLUTE OBJECT REFERENCE of the remote object to be used instead of the original one.

Distributed object middleware constituents that need to be accessed by application code appear as LOCAL OBJECTS. LOCAL OBJECTS are provided to simplify the programming models by supporting the same parameter passing and other rules, as they are used for remote objects. However, LOCAL OBJECTS are not accessible remotely.

Lifecycle Manager

The server application has to manage different kinds of lifecycles for remote objects.



The lifecycle of remote objects needs to be managed by server applications. Based on configuration, usage scenarios, and available resources, servants have to be instantiated, initialized, or destroyed. Most importantly, all this has to be coordinated.

The server application has to manage its resources efficiently. For instance, it should ensure that only those servants of remote objects are loaded into memory that are actually needed at a specific point in time.

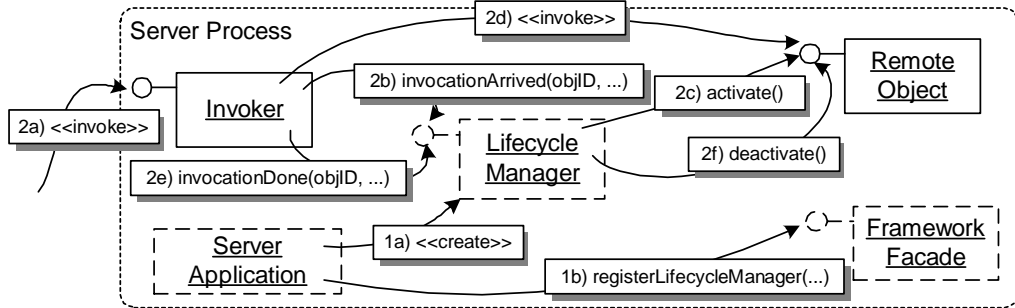
Not only the creation and loading of servants is expensive, but also their destruction and cleanup. Cleanup and destruction might involve invoking a destructor, releasing an ABSOLUTE OBJECT REFERENCE, invoking custom cleanup routines, and recycling servants using POOLING.

The lifecycle strategies should not be mixed with other parts of the distributed object middleware as the strategies can become quite complex, but also need to be highly configurable. Specifically, it should be possible for application developers to customize lifecycle strategies or even implement their own lifecycle strategies.

Therefore:

Use a LIFECYCLE MANAGER to manage the lifecycle of remote objects and their servants. Let the LIFECYCLE MANAGER trigger lifecycle operations for servants of remote objects according to their configured lifecycle strategy. For servants that have to be aware of lifecycle events, provide *Lifecycle Callback* operations [VSW02]. The LIFECYCLE MANAGER will use those operations to notify the servant of

upcoming lifecycle events. This allows servants to prepare for these events accordingly.



A LIFECYCLE MANAGER is typically created by the server application during startup, and is registered with the distributed object middleware's INVOKER. Before an invocation is dispatched, the INVOKER informs the LIFECYCLE MANAGER. If the servant is not active, the LIFECYCLE MANAGER activates it. The INVOKER dispatches the invocation. After the invocation returns, the LIFECYCLE MANAGER is informed again and can possibly deactivate the servant.



The LIFECYCLE MANAGER modularizes the lifecycle strategies including activation, PASSIVATION, POOLING, LEASING and eviction (as explained in the *Lifecycle Patterns* Chapter). Such strategies are important for optimizations regarding performance, stability, and scalability.

The LIFECYCLE MANAGER with its strategies is either implemented as part of the INVOKER, or closely collaborates with it. If multiple different lifecycle strategies have to be provided, different LIFECYCLE MANAGERS can be available in the same server application.

The INVOKER triggers the LIFECYCLE MANAGER before and after each invocation. This allows the LIFECYCLE MANAGER to manage the creation, initialization, and destruction of servants.

For complex remote objects, for example when they hold non-trivial state, it can become necessary to involve the servants into lifecycle management by informing them about upcoming lifecycle events. For this, the LIFECYCLE MANAGER invokes *Lifecycle Callback* operations

implemented by the servant. For example, when using PASSIVATION, a remote object's state has to be saved to persistent storage before the servant is destroyed. After the servant has been resurrected, the servant has to reload its previously saved state. Both events are triggered by the LIFECYCLE MANAGER via *Lifecycle Operations* just before the servant is passivated and just after resurrection.

Triggering the LIFECYCLE MANAGER can be hard-coded inside the INVOKER, but it can also be 'plugged in' using an INVOCATION INTERCEPTOR. Besides the synchronous involvement of the LIFECYCLE MANAGER when triggered by an INVOKER, a LIFECYCLE MANAGER implementation can also become active asynchronously, for example to scan for remote objects that should be destroyed because some LEASE has expired.

To decouple remote objects from servants the LIFECYCLE MANAGER must maintain the association between OBJECT ID and servant. This mapping is either kept separately in the LIFECYCLE MANAGER or is reused from the INVOKER. Additionally, the LIFECYCLE MANAGER also has to store the information which lifecycle state each servant is in.

Besides all the advantages of decoupling lifecycle strategies from the INVOKER, the LIFECYCLE MANAGER also incurs a slight performance overhead, as it has to be invoked on every request of a remote object.

The LIFECYCLE MANAGER pattern applies the *Resource Lifecycle Manager* pattern [KJ04] to the management of remote objects. It integrates several existing patterns, such as *Activator* [Sta00] and *Evictor* [HV99].

Configuration Group

A server application manages remote objects that require similar configurations.



In many applications the remote objects need to be configured with various properties, such as quality of service (QoS), lifecycle management, or protocol support. Configuring such properties per server application is often not flexible enough and configuring them for each individual remote object separately might lead to an implementation overhead.

The implementations of LIFECYCLE MANAGER, PROTOCOL PLUG-INS, MARSHALLER, and INVOCATION INTERCEPTORS, need to be configured with respect to how invocations are actually handled. Especially, some remote objects might require other configurations of these distributed object middleware constituents than other remote objects.

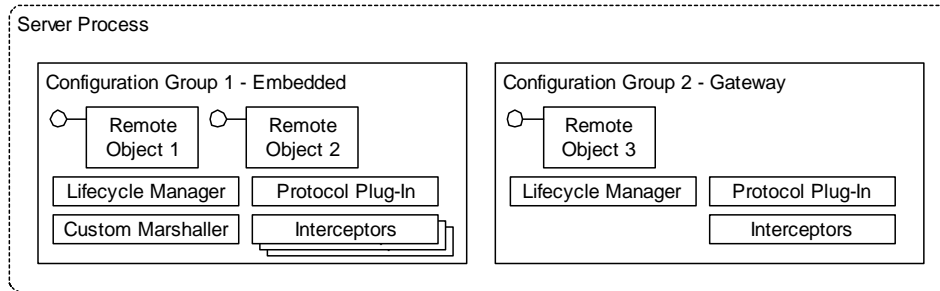
Configuring the framework constituents for each remote object separately is tedious, and incurs implementation and resource-usage overhead. For instance, to configure the priorities of all remote objects in an embedded system and to align those priorities, requires a lot of implementation effort. It is also hard to track which priorities were assigned.

A global configuration for the complete server application is usually not sufficient either: in a vehicle, for instance, a brake control remote object will have to run at a higher priority than an air conditioning remote object, although both might be hosted by the same server application.

Therefore:

Provide CONFIGURATION GROUPS which group remote objects with common properties, for example the same QoS properties or the same lifecycle. A server application can have multiple CONFIGURATION GROUPS at the same time. Configuration properties for the

constituents, such as LIFECYCLE MANAGER, PROTOCOL PLUG-INS, etc., are specified at the CONFIGURATION GROUP level.



The figure depicts two CONFIGURATION GROUPS in a server application: one group for embedded objects and one group for gateway objects. The LIFECYCLE MANAGER, PROTOCOL PLUG-IN, MARSHALLER, and INVOCATION INTERCEPTORS are configured accordingly.



Remote objects requiring the same configuration of the distributed object middleware, such as transaction support, pooling, or security checks, are registered with the same CONFIGURATION GROUP. The CONFIGURATION GROUP is set up in advance, before remote objects register themselves with it, and before invocations are dispatched to them.

The implementation of CONFIGURATION GROUPS is often tightly integrated with the SERVER REQUEST HANDLER and INVOKERS in a distributed object middleware. This is necessary, because many properties, such as execution priority, invocation of INVOCATION INTERCEPTORS, or lifecycle management of remote objects, have to be configured at the proper dispatching level. Due to the configuration differences, a distributed object middleware that supports CONFIGURATION GROUPS might have several differently configured SERVER REQUEST HANDLERS, and INVOKERS.

The configuration of the distributed object middleware constituents, such as INVOKER or LIFECYCLE MANAGER, can be simplified by associating the CONFIGURATION GROUP with each of them. The constituents are either statically configured with a configuration, or retrieve their

configuration from the CONFIGURATION GROUP when the invocation is passed through them. The later approach avoids that multiple instances of the constituents—one for each configuration—are necessary. The dynamic adaptation to configuration properties is especially effective if the constituents are implemented as chained INVOCATION INTERCEPTORS because interceptors can easily be dynamically composed in different chains for different remote objects or groups of objects respectively.

CONFIGURATION GROUPS are typically represented as LOCAL OBJECTS, so that developers can configure them programmatically and eventually change configuration at runtime—disallowing remote access. A good way to keep the overview of the possible configurations, is to organize CONFIGURATION GROUPS into hierarchies of CONFIGURATION GROUPS. To model commonalities, configurations may be inherited by child configurations from parent configurations, unless the child overwrites a configuration parameter with its own, more specific, values.

The collective configuration of remote objects via CONFIGURATION GROUPS eases administration and reduces the risk of incompatible configurations of remote objects.

However, if remote objects substantially differ in their required properties, so that many different CONFIGURATION GROUPS become necessary, CONFIGURATION GROUPS may also cause additional complexity.

Local Object

The distributed object middleware needs to provide interfaces for constituents that need to be configured.

✱ ✱ ✱

To configure policies and parameters in distributed object framework constituents, such as PROTOCOL PLUG-INS, CONFIGURATION GROUPS, or LIFECYCLE MANAGERS, the application programmer must have access to their interfaces. The interfaces must not be accessible remotely, as they should only be configured by the server application developer. However, for consistency reasons, the interfaces should behave similarly as those of remote objects, with respect to parameter passing rules, memory management, and invocation syntax.

Many constituents of a distributed object middleware need to be accessed and/or configured:

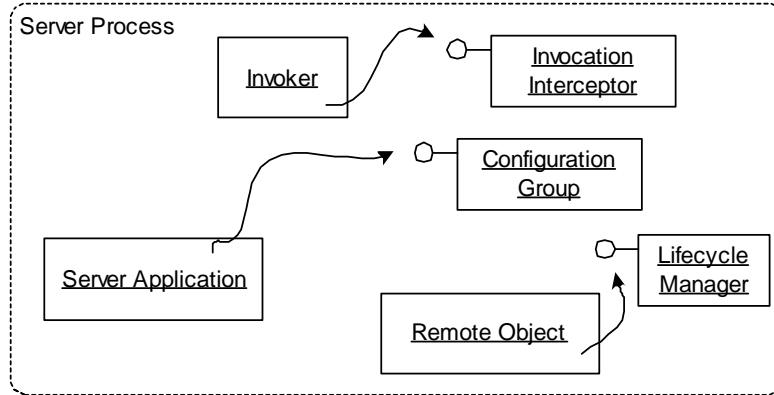
- The LIFECYCLE MANAGER must be configured with activation and eviction strategies.
- The PROTOCOL PLUG-IN must be set and configured.
- CONFIGURATION GROUPS must be set up and remote objects be registered.
- INVOCATION INTERCEPTORS must be created and set up.
- INVOCATION CONTEXT objects, such as the current transaction or a security context, must be accessible.

The interfaces of those constituents must not be accessible remotely, in order to avoid inconsistencies and privacy violations. The interfaces should behave like interfaces of remote objects, so that it is not necessary to deal with two different programming models from the point of view of the server application developer. For example, they should have the same parameter passing semantics, they should be constructed the same way, and maybe even be registered/found using LOOKUP.

Therefore:

Provide LOCAL OBJECTS to allow application developers, on the client- and server-side, to access configuration and status parameters of the

distributed object middleware constituents. Ensure that the LOCAL OBJECTS adhere to the same parameter passing rules, memory management, and invocation syntax as remote objects. However, they must not be accessible remotely.



The server application and the implementations of remote objects access LOCAL OBJECTS as if they were remote objects.



LOCAL OBJECTS make specific constituents of distributed object middleware, such as LIFECYCLE MANAGERS and PROTOCOL PLUG-INS, accessible for server application developers. Other constituents of the distributed object middleware are implemented as regular objects, to which the application programmer will have no access.

To avoid remote access, it has to be ensured that it is not possible to create ABSOLUTE OBJECT REFERENCES for LOCAL OBJECTS. Without the existence of ABSOLUTE OBJECT REFERENCES for LOCAL OBJECTS, the remote clients are prevented from access to a LOCAL OBJECT.

LOCAL OBJECTS allow for a consistent programming model between remote objects and the distributed object middleware's internal constituents. For distributed object middleware systems that have to support several platforms and/or programming languages, LOCAL OBJECTS help to standardize APIs among the different implementations. APIs of LIFECYCLE MANAGERS, PROTOCOL PLUG-INS, etc. can be defined using the INTERFACE DESCRIPTION provided by the distributed object middleware. For instance an interface definition language can be used for defining

| those constituents and the standard language mapping can be used to define the programming language dependent API.

QoS Observer

You want to control application-specific quality of service properties, such as bandwidth, response times, or priorities, when clients access remote objects.



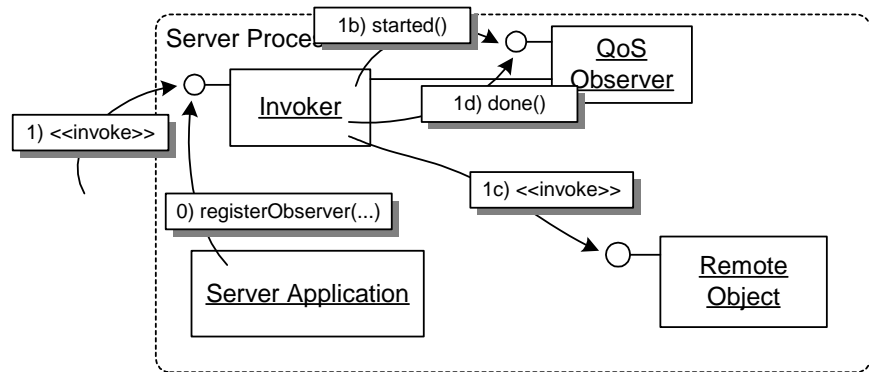
Distributed object middleware constituents, such as the REQUEST HANDLERS, MARSHALLERS, PROTOCOL PLUG-INS, CONFIGURATION GROUPS, provide hooks to implement a wide variety of quality of service characteristics. Applications might want to react on changes in the quality of service currently provided. The application-specific code to react on those changes should be decoupled from the middleware itself.

Consider a situation where response times need to be guaranteed to clients. The response time is determined, for example, by the available network bandwidth, the network traffic, and the load of the server application. To ensure that the response times are actually met, a reasonable strategy is to stop accepting new clients when the response time reaches a certain threshold. To do this, you need to be able to monitor the (average) response time and decline further clients when the quality of service deteriorates.

In general, to react on QoS changes, monitoring of quality of service characteristics is necessary. Because the thresholds and the reactive behavior are typically application-specific, the QoS monitoring functionality should be accessible and configurable by application developers.

Therefore:

Provide hooks in the distributed object middleware constituents where application developers can register QOS OBSERVERS. The observers are informed about relevant quality of service parameters, such as message sizes, invocation counts, or execution times. Since the QOS OBSERVERS are application specific, they can contain code to react appropriately if service quality gets worse.



On startup, QOS OBSERVERS implementations are registered with the INVOKER (or any other relevant framework constituent). The INVOKER notifies the QOS OBSERVERS of the start and end of each invocation. The QOS OBSERVERS can obtain relevant invocation data, such as duration and size, from the constituent it is registered with.



On startup application-specific QOS OBSERVERS are registered with the constituents whose quality of service they should observe. Typical candidates are REQUEST HANDLERS, MARSHALLER, PROTOCOL PLUG-INS, INVOKER, REQUESTOR, and LIFECYCLE MANAGER. The framework constituents reports events, such as invocation start/invoke end, connection establishment, incoming invocations, return of invocation, marshalling started/ended. Configuration of the constituents should be done per CONFIGURATION GROUP. Triggered by the events, the QOS OBSERVERS checks execution time, size, or other QoS characteristics.

The respective middleware constituents need to provide an API so that QOS OBSERVERS can query for additional relevant information. For example, the QOS OBSERVER might need to know which operation has been requested in the current request, or how large the message for this invocation actually was.

To minimize the performance overhead, it is important to make sure the QOS OBSERVERS are only notified about relevant changes they have registered for.

The QOS OBSERVER allows to monitor relevant QoS properties and to react on events triggered by them. Applications benefit from observing QoS properties, as they can better adapt and react on critical situations.

QOS OBSERVERS are typically LOCAL OBJECTS. INVOCATION INTERCEPTORS can be used to implement certain quality of service observations, such as operation invocation times. Be aware that the monitoring and handling of QoS properties might influence the actual observations.

Location Forwarder

Invocations of remote objects arrive at the INVOKER and the INVOKER should dispatch them.

✱ ✱ ✱

Remote objects might be located in a different server application than the one where the invocation arrives. The reasons could be: remote objects are put onto multiple servers to achieve load balancing and fault tolerance, or the remote objects were moved to other server applications. The invocations should transparently reach the correct remote object, even though it lives in another server application.

The lifetime of server applications can be very long, some are expected to never shut down, but the world around them changes. The deployment of remote objects can change, for example because of resource shortage in the original server application, but existing ABSOLUTE OBJECT REFERENCES should stay valid and the invocation should still reach the target remote object.

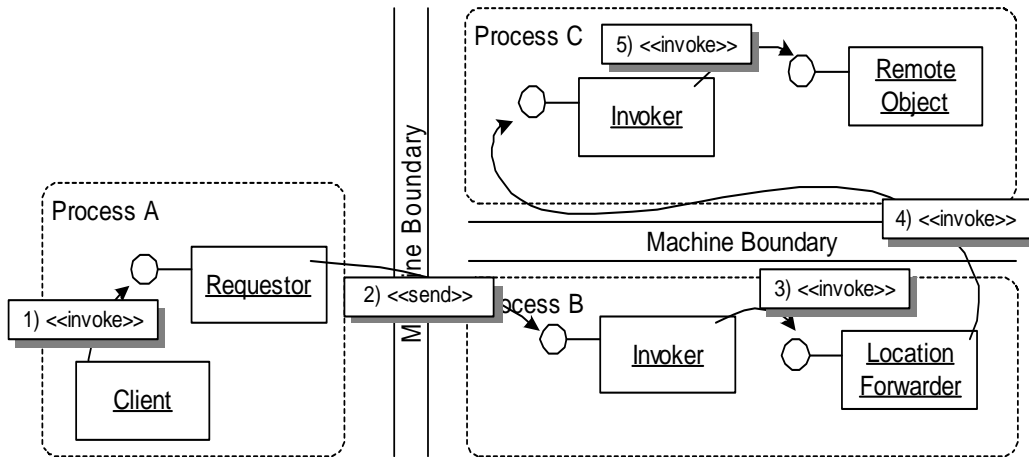
Further, scalable and highly available distributed applications rely on load balancing and fault tolerance mechanisms.

- To achieve load balancing typically an additional layer of indirection is added: a load dispatcher. The load dispatcher is responsible for coordinating the invocations of the target remote objects. How can you delegate requests to multiple instances, while making the client believe it communicates with only one instance?
- One way to reach fault tolerance, especially in the case of hardware faults, is to replicate remote objects and distribute them over several nodes. The replicated objects form a group of remote objects, all implementing the same functionality. How can you make an invocation to such a group look like only one remote object existed?

The general problem is that server applications have to collaborate when remote objects do not reside at the location to which the ABSOLUTE OBJECT REFERENCE points to, because they have been either moved or aggregated.

Therefore:

For remote objects that the INVOKER cannot resolve locally, make the LOCATION FORWARDER forward invocations to a remote object in other server applications. The LOCATION FORWARDER looks up the actual location of the remote object based on the OBJECT ID. The result of this lookup is an ABSOLUTE OBJECT REFERENCES of another remote object. The LOCATION FORWARDER has two options: either it sends the client-side distributed object middleware an update notification about the new location, so the client framework can retry the invocation on the new location, or it transparently forwards the invocation to the new location.



A client invokes an operation on a remote object. Since the instance has moved from one server application to another one, the LOCATION FORWARDER forwards the invocation to the server application where the remote object resides now.



When a client performs an invocation using a particular ABSOLUTE OBJECT REFERENCE in a server application, the invocation will reach the INVOKER. If the INVOKER cannot resolve the target remote object locally, the INVOKER can delegate the invocation to a LOCATION FORWARDER. The LOCATION FORWARDER looks up the new location, represented as an ABSOLUTE OBJECT REFERENCE, based on the OBJECT ID. The behavior that

occurs after the new location has been resolved, depends on the use case.

The following paragraphs explain variants of location forwarding, based on the scenarios mentioned above:

- If remote objects are moved to another server application, let the LOCATION FORWARDER inform the client of the move. Forwarding the invocation transparently to the new location would increase the load of the local server application, as it would have to permanently perform this forwarding. Therefore, it is better to let the client know of the move, so that it can send subsequent invocations directly to the new location.
- For the purpose of load balancing, the LOCATION FORWARDER acts as a load dispatcher. The LOCATION FORWARDER looks up the ABSOLUTE OBJECT REFERENCES to a set of remote objects and redirects the invocation to one of these remote objects. Inside the LOCATION FORWARDER different load balancing algorithms can be applied, such as round robin or smallest number of clients. Again, the LOCATION FORWARDER can become a bottle-neck, if all invocations are routed through it. To avoid this, let the LOCATION FORWARDER update the client to equally distribute the load on the available instances.
- In the case of fault tolerant systems, the LOCATION FORWARDER can act as part of the coordinator of an *Object Group* [Maf96], and thus forward invocations to all remote objects in the *Object Group*. The LOCATION FORWARDER does not update the client location, unless a new coordinator for an *Object Group* is determined, in which case the client is updated with a reference to the new coordinator. In the same way, the LOCATION FORWARDER can be used as a distributor unit in the patterns *Fail-Stop Processor* [Sar02] and *Active Replication* [Sar02] (see the Chapter *Related Concepts, Technologies, and Patterns* for more explanations).
- A special case is an internal failure of a remote object in which case the LOCATION FORWARDER should either update the client with the location of a working remote object of the same type, or forward the invocations until the locally existing remote object is repaired.

Using a LOCATION FORWARDER has the advantage that the deployment of a distributed application can be hidden from clients. This reduces the

complexity from a client perspective and eases overall maintainability. ABSOLUTE OBJECT REFERENCES that clients have once obtained stay valid—even if the deployment of the remote object changes.

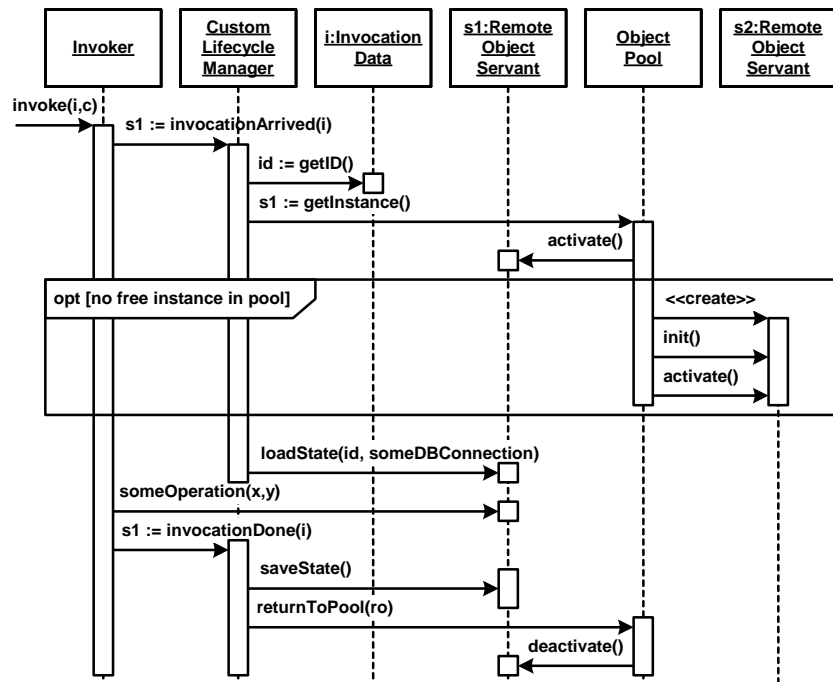
As with every additional level of indirection, the LOCATION FORWARDER incurs a performance overhead and increases the risk of failures. In the case of load balancing, a failed load dispatcher causes the complete distributed application to fail. A typical solution to this is the replication of the load dispatcher using lower-level means, for example at the level of IP addresses.

It can become confusing for developers of client applications, when the location of a remote object is constantly updated. Optimizations that make assumptions about the location of the remote object become useless, as the remote object can have moved anywhere.

This pattern enables fault-tolerant systems; a pattern language on this topic can be found in [Sar03] (see the Chapter *Related Concepts, Technologies, and Patterns*).

Interactions among the Patterns

First, let us have a look at the LIFECYCLE MANAGER. For illustrative purposes, we will show how persistent state can be realized for remote objects in combination with POOLING. In the followig example we assume that the developers of the server application have configured the distributed object middleware to use a custom developed LIFECYCLE MANAGER. Note that these interactions are just examples of how the patterns can be implemented. Many other alternatives exist.

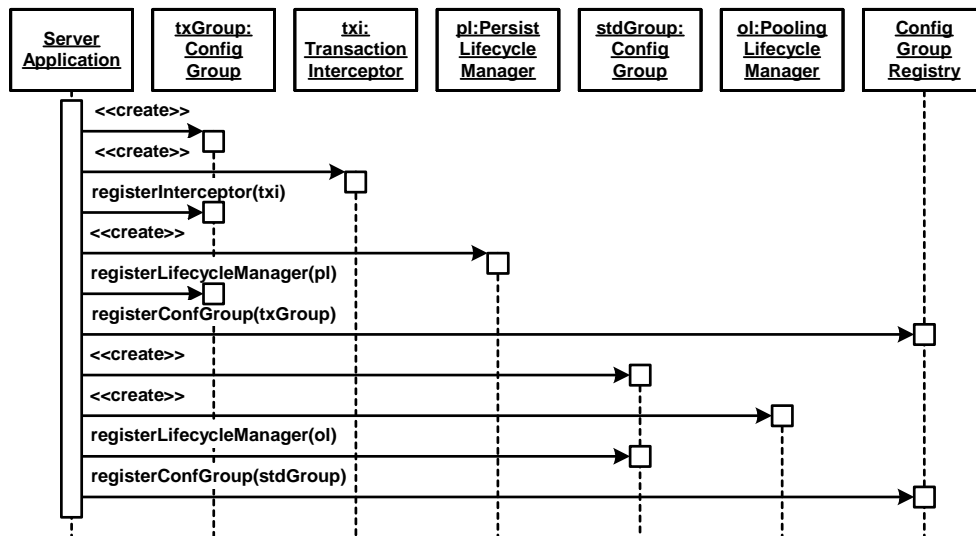


In the figure above, the LIFECYCLE MANAGER obtains a servant from the pool of servants. After the servant is activated (or optionally created lazily), the state, associated with the given OBJECT ID, is loaded from the persistent storage. The operation is executed and finally the state of the remote object is passivated and the servant is put back into the pool.

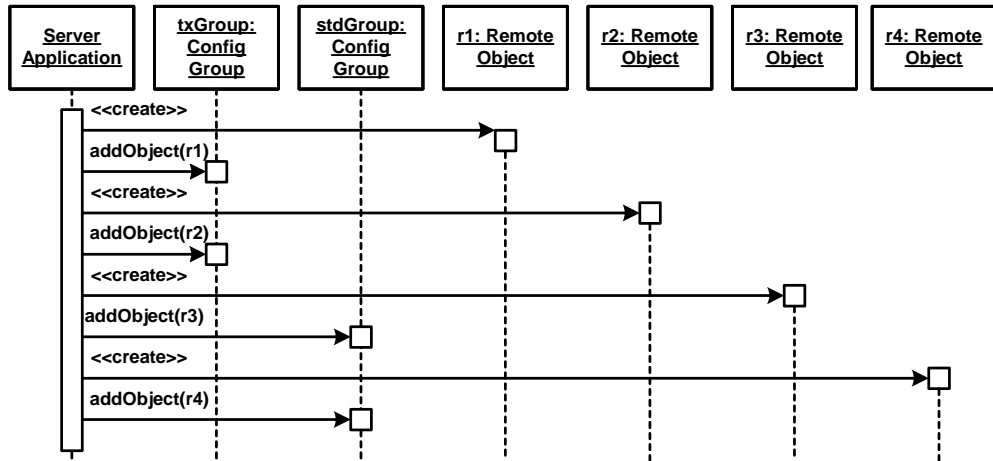
Remote object servants that should be used with this LIFECYCLE MANAGER need to implement up to five lifecycle operations [VSW02]: *init()*, *activate()*, and *deactivate()* to be compatible with POOLING, and

loadState() and *saveState()* to be compatible with the persistent state handling used by PASSIVATION.

Next we will have a look at a possible scenario for CONFIGURATION GROUPS. Let us first look at how a server application sets up two CONFIGURATION GROUPS. One contains a LIFECYCLE MANAGER that uses pooling, whereas the other one contains a persistent LIFECYCLE MANAGER, as well as a transaction INVOCATION INTERCEPTOR to make the persistence aspect transactional. For each of the CONFIGURATION GROUPS, at first, the group object has to be instantiated. Next, INVOCATION INTERCEPTORS and LIFECYCLE MANAGERS for the group have to be created and registered. Finally, the group has to be registered itself with a group registry of the distributed object middleware.

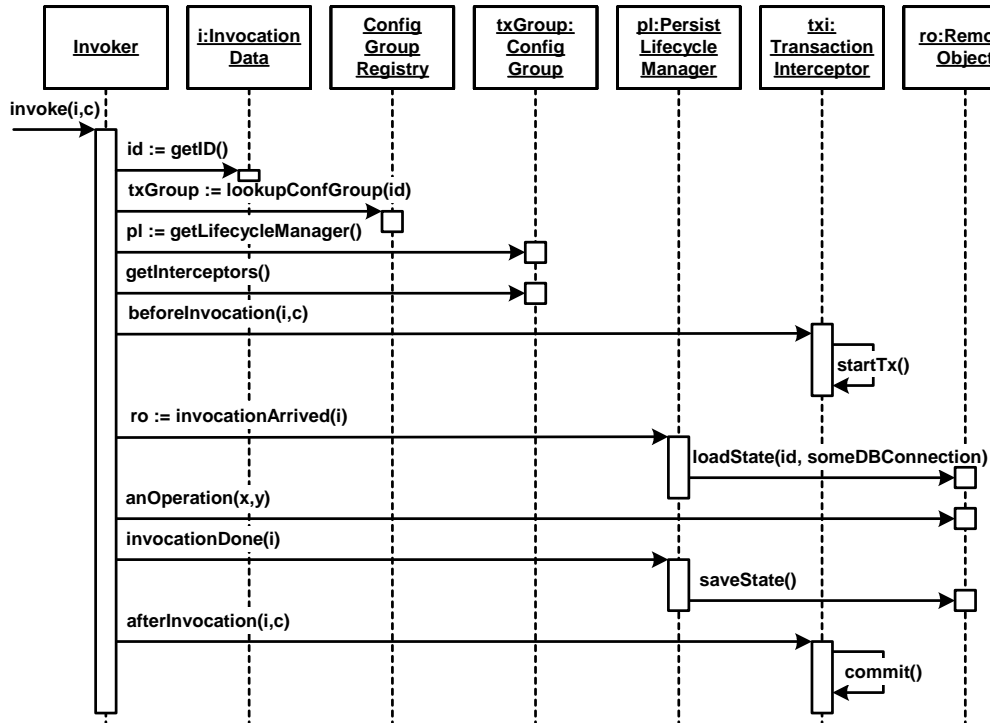


In the next step, we will have to create a set of remote objects and register them with the CONFIGURATION GROUPS.



Now invocations can be handled by objects in the CONFIGURATION GROUP. In the next diagram it is shown how the invocation proceeds when using different CONFIGURATION GROUPS. We can see that the INVOKER first looks up the CONFIGURATION GROUP. According to this group, the LIFECYCLE MANAGER is chosen, as well as the INVOCATION INTERCEPTORS. In the example the persistent group is chosen. This

causes the persistent interceptor and LIFECYCLE MANAGER to manage persistency of the remote object instance.



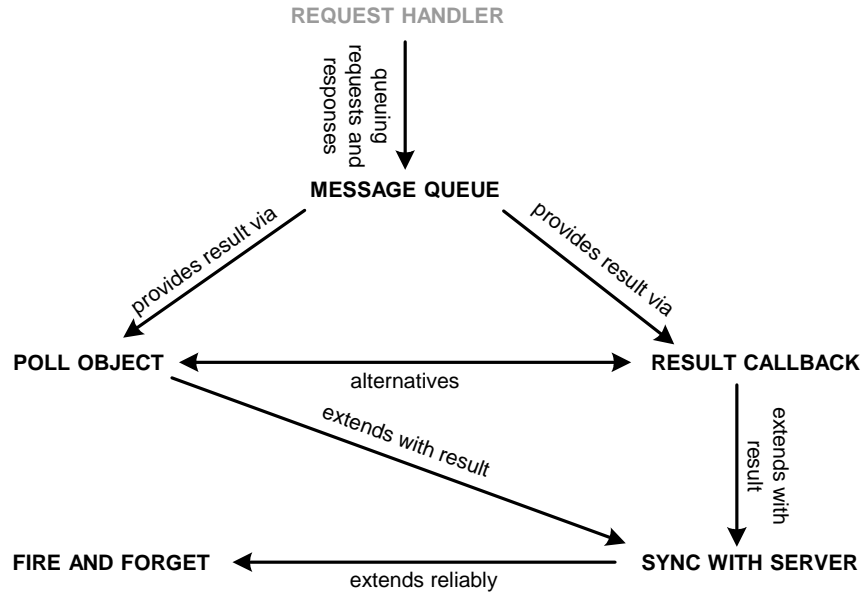
12 Invocation Asynchrony Patterns

This chapter deals with asynchrony issues in a remote invocation between a client and a server. This form of asynchrony has to be contrasted to asynchrony support that is only operating on client side or server side. For instance, asynchronous request handling in the server process, as provided by the patterns *Half-sync/Half-async* [SSRB00] or *Active Object* [SSRB00], is completely independent of the client process. In other words, in this chapter we illustrate patterns that enable a client process to resume its work directly after a remote invocation is sent - without awaiting the result of the server to arrive.

Asynchrony is often motivated by performance and throughput optimizations. The optimizations try to benefit from the inherent asynchrony of the communication channels in a way that the client continues with other work, while waiting for replies of earlier invocations.

FIRE AND FORGET describes best-effort delivery semantics for asynchronous operations that have void return types. SYNC WITH SERVER looks the same from the client's point of view, however, the pattern additionally describes how to notify the client in case the delivery of the invocation to the server application fails (for instance by throwing an exception in the client). POLL OBJECTS provide clients with means to query the distributed object middleware whether an asynchronous reply for the request has arrived yet, and if so, to obtain the return value. RESULT CALLBACK actively notifies the requesting client of the

returning result. The pattern MESSAGE QUEUE can be used to implement a simple form of *Messaging* [HW03] by queueing requests and replies.



The following table illustrates the alternatives for applying the patterns. It distinguishes whether there is a result sent to the client or not, whether the client gets an acknowledgement or not, and, if there is a result sent to the client, it may be the client's burden to obtain the result or it is informed using a callback.

	Acknow- ledgement to client	Result to cli- ent	Responsibility for result
FIRE AND FORGET	no	no	-
SYNC WITH SERVER	yes	no	-
POLL OBJECT	yes	yes	The client is responsible for getting the result.

RESULT CALLBACK	yes	yes	The client is informed via call-back.
MESSAGE QUEUE	yes	yes	The server actively sends back the result. The client can receive it synchronously (by blocking on a MESSAGE QUEUE) or asynchronously using one of the other asynchrony patterns.

SYNC WITH SERVER should be applied when message delivery needs to be more reliable than with FIRE AND FORGET, though message invocation will still be unreliable. This is achieved by sending an acknowledgement for the reception of the request from the server back to the client. The patterns POLL OBJECT and RESULT CALLBACK do not only provide an acknowledgement of the reception of the request, but also send a response back asynchronously. Thus these two patterns implicitly acknowledge that the request has been processed.

POLL OBJECT should be used when the result must be processed

- by the same thread that has send out the message and
- not immediately after arrival.

In contrast, RESULT CALLBACK should be used when it is required to react on the result immediately after arrival of the message.

As a fifth asynchronous invocation pattern, we discuss MESSAGE QUEUES. MESSAGE QUEUES extend the notion of asynchronous message delivery by processing both the request and the result asynchronously. Thus they can be used to tolerate temporal failures of the network connection, the client, or the server. Also they can ensure reliable transmission of requests and results. Using MESSAGE QUEUES the other four invocation asynchrony patterns can be implemented.

Fire and Forget

Your server application provides remote objects with operations that have neither a return value nor report any exceptions.

* * *

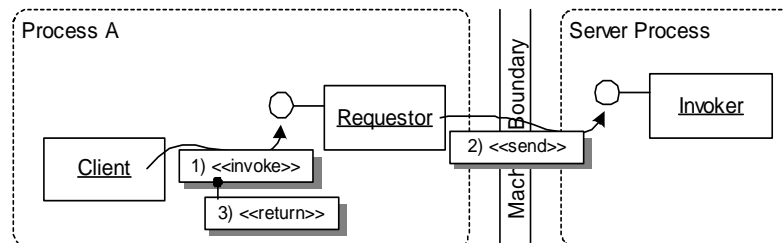
In many situations, a client application needs to invoke an operation on a remote object simply to notify the remote object of an event. The client does not expect any return value. Reliability of the invocation is not critical, as it is just a notification that both client and server do not critically rely on.

Consider a simple logging service implemented as remote object. Clients use it to record log messages. But recording of log messages must not influence the execution of the client. For example, an invocation of the logging service must not block. Loss of single log messages is acceptable.

Note that this scenario is quite typical for distributed implementations of patterns, such as *Model-View-Controller* [BMR+96] or *Observer* [GHJV95], especially if the view or observer is constantly notified and old data is stale data.

Therefore:

Provide FIRE AND FORGET operations. When invoked, the REQUESTOR sends the invocation across the network, returning control to the calling client immediately. The client does not get any acknowledgement from the remote object receiving the invocation in case of success or failure.



When the client invokes a FIRE AND FORGET operation, the REQUESTOR marshals the parameters and sends them to the server.



The implementation of a FIRE AND FORGET operation can be done in multiple ways, specifically:

- The REQUESTOR can simply put the bytes on the wire in the caller's thread, assuming the send operation does not block. Here, asynchronous I/O operations, as supported by some operating systems are of great help to avoid blocking.
- Alternatively, the REQUESTOR can spawn a new thread that puts the bytes on the wire independently from the thread that invoked the remote operation. This variant also works when the send operation temporarily blocks. However, this variant has some drawbacks: it works only as long as the application does not get bogged down from the operating system perspective due to huge numbers of such threads, and the existence of such threads does not overwhelm the underlying marshaling, protocol, and transport implementations due to lock contention, etc. Another drawback of concurrent invocations is that an older invocation may bypass a younger one. This can be avoided by using MESSAGE QUEUES.

As FIRE AND FORGET operations are not considered to be reliably transported, an option is to use unreliable protocols such as UDP for their implementation, which are much cheaper than reliable protocols such as TCP (UDP is connectionless and unreliable. Thus it does not establish a virtual circuit like TCP. Also, it does require an acknowledgement of messages.).

The INVOKER on the server side typically differentiates between FIRE AND FORGET operations and operations returning a result, as it is not necessary to send a reply for a FIRE AND FORGET operation. When the remote invocation is performed in a separate thread, a thread pool will be used instead of spawning a new thread for each invocation to avoid a thread creation overhead.

In cases where the distributed object middleware does not provide FIRE AND FORGET operations, the client application can emulate such behavior by spawning a thread itself and performing the invocation in

that newly created thread. But be aware that such an emulation heavily influences the scalability. In particular, many concurrent requests lead to many concurrent threads, decreasing overall system performance.

The benefit of the FIRE AND FORGET pattern is the asynchrony it provides compared to synchronous invocations. Client and remote object are decoupled, in the sense that the remote object executes independently of the client; the client does not block during the invocation. This means the pattern is very helpful in event-driven applications that do not rely a continuous control flow nor on return values. Further, it is important that the applications do not rely on the successful transmission.

REMOVING ERRORS during sending the invocation to the remote object or errors that were raised during the execution of the remote invocation cannot be reported back to the client. The client is unaware whether the invocation ever got executed successfully by the remote object. Therefore, FIRE AND FORGET usually has only “best effort” semantics. The correctness of the application must not depend on the “reliability” of a FIRE AND FORGET operation invocation. To cope with this uncertainty, especially in situations, where the client expects some kind of action, clients typically use time-outs to trigger compensating actions.

Sync with Server

Your server application provides remote objects with operations that have neither return value nor report any errors.



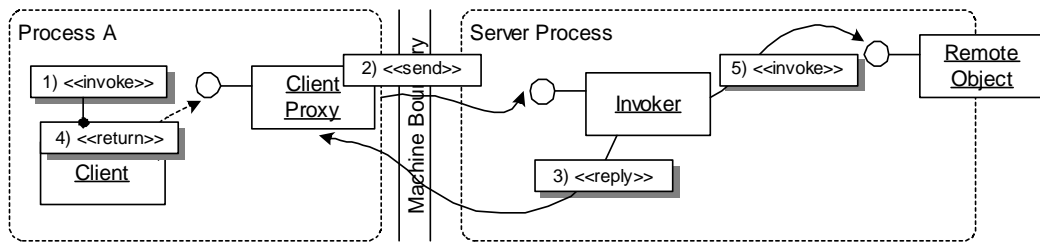
FIRE AND FORGET is a useful but extreme solution in the sense that it can only be used if the client can really afford to take the risk of not noticing when a remote invocation does not reach the targeted remote object. The other extreme is a synchronous call where a client is blocked until the remote method has executed successfully and the result arrives back. Sometimes the middle of both extremes is needed.

Consider a system that stores images in a database. Before the images are actually stored in the database, they are filtered, for example by a Fourier transformation that may take rather long. The client is not interested in the result of the transformation but only in a notification that it is delivered as a message to the server. Thus the client does not need to block and wait for the result; it can continue executing as soon as the invocation has reached the remote object.

In this scenario, the client only has to ensure that the invocation containing the image is transmitted successfully. However, from that point onward it is the responsibility of the server application to make sure the image is processed correctly and then stored safely in the database.

Therefore:

Provide SYNC WITH SERVER semantics for remote invocations. The client sends the invocation, as in FIRE AND FORGET, but waits for a reply from the server application informing it about the successful reception (not the execution!) of the invocation. After the reply is received by the REQUESTOR, it returns control to the client and execution continues. The server application independently executes the invocation.



A client invokes a remote operation. The REQUESTOR puts the bytes of the invocation on the wire, as in FIRE AND FORGET. But then it waits for a reply from the server application as an acknowledgment that the invocation has been received by the server.

* * *

In the SYNC WITH SERVER pattern, as in FIRE AND FORGET, no return value or out parameters of the remote operation can be carried back to the client. The reply sent by the server application is only to inform the REQUESTOR about the successful reception.

If the distributed object middleware supports SYNC WITH SERVER operations, the INVOKER can send the reply message immediately after reception of the invocation. Otherwise, SYNC WITH SERVER can be emulated by hand-coding SYNC WITH SERVER into the respective operations of the remote object. The operation spawns a new thread that performs the remote invocation while the initial thread invoking the remote invocation returns immediately resulting in a reply to the client.

Compared to FIRE AND FORGET, SYNC WITH SERVER operations ensure successful transmission and thus make remote invocations more reliable. However, the SYNC WITH SERVER pattern also incurs additional latency - the client has to wait until the reply from the server application arrives.

Note that the REQUESTOR can inform the client of system errors, such as a failed transmission of the invocation. However, it cannot inform clients about application errors during the execution of the remote invocation in the remote object because this happens asynchronously.

Poll Object

Invocations of remote objects should execute asynchronously, and the client depends on the results for further computations.



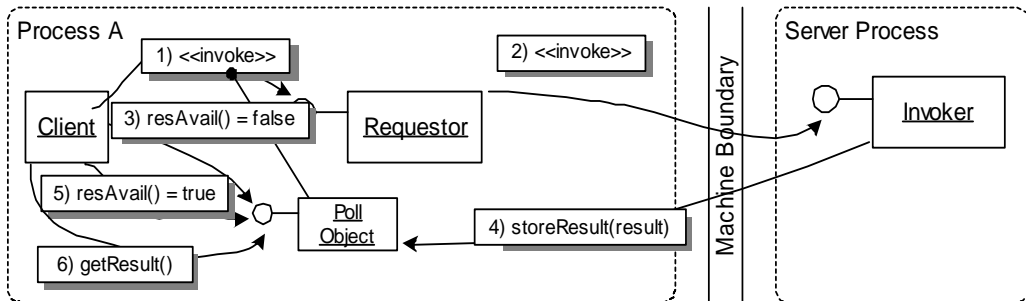
There are situations, when an application needs to invoke an operation asynchronously, but still requires to know the results of the invocation. The client does not necessarily need the results immediately to continue its execution, and it can decide for itself when to use the returned results.

Consider a client that needs to prepare a complex XML document to be stored in a relational database that is accessed through a remote object. The document shall have a unique ID, which is generated by the database system. Typically, a client would request an ID from the database, wait for the result, create the rest of the XML document, and then forward the complete document to the remote object for storage in the database. A more efficient implementation is to first request the ID from the database. Without waiting for the ID, the client can prepare the XML document, receive the result of the ID query, put it into the document, and then forward the whole document to the remote object for storage.

In general, a client application should be able to make use of even short periods of latency, instead of blocking idle until a result arrives.

Therefore:

As part of the distributed object middleware, provide POLL OBJECTS that receive the result of remote invocations on behalf of the client. The client subsequently uses the POLL OBJECT to query the result. It can either just query (“poll”), whether the result is available, or it can block on the POLL OBJECT until the result becomes available. As long as the result is not available on the POLL OBJECT, the client can continue with other tasks asynchronously.



A client invokes a remote operation on the REQUESTOR, which in turn creates a POLL OBJECT to be returned to the client immediately. As long as the remote invocation has not returned, the “result available” operation of the POLL OBJECT returns “false”. When the result becomes available, it is memorized in the POLL OBJECT. When it is polled the next time, it returns “true”, so that the client can fetch the result by calling the “get result” operation.



The POLL OBJECT has to provide at least two operations: one to check if the result is available; the other to actually return the result to the calling client. Besides this client interface, an operation for storing the result as received from the server is needed.

Most POLL OBJECT implementations also provide a blocking operation that allows clients to wait for the availability of the result, once they decide to do so. The core idea of this pattern follows the *Futures* concept described in [Lea99], but POLL OBJECT extends it to distributed settings and allows to query for the availability of the result in a non-blocking fashion, whereas with *Futures* this query would block.

POLL OBJECTS typically depend on the interface of the remote object. To be able to distinguish results of two invocations on the same remote object, either, the two operations “result available” and “get result” must be provided for each of the remote object’s operations, or a separate POLL OBJECT for each operation must exist.

Use POLL OBJECTS when the time until the result is received is expected to be rather short; however, it should be long enough so that the client

can use the time for other computations. For longer waiting periods, especially if the period cannot be pre-estimated, use a RESULT CALLBACKS because it is hard to manage (a potentially huge number) of POLL OBJECTS over a longer time period. A number of “small” programming tasks would be required between the polls, and polling would have to be triggered constantly. This constant triggering can be avoid by using RESULT CALLBACKS - here the callback events handle triggering for you.

POLL OBJECTS offer the benefit that the client application does not have to use an event-driven, completely asynchronous programming model, as in the case with RESULT CALLBACK, while it can still make use of asynchrony to some extent. The server application can stay unaware to client side poll objects. However, the result must be received asynchronously, for instance, by a separate thread or process.

From an implementation perspective, the client framework typically starts a separate thread to “listen” for the result and fill it into the POLL OBJECT. In this thread, the client typically sends a synchronous invocation to the server application.

Some synchronization mechanisms between the client thread and the retrieving thread are needed, for instance by applying mutexes or a *Thread-Safe Interface* [SSRB00].

When using POLL OBJECTS the client has to be changed slightly to do the polling. POLL OBJECTS either need to be generic, which typically requires programming language support, or they have to be specific to the remote object and its interface operations. In the latter case they are typically code generated. More dynamic environments can use runtime means to create the types for POLL OBJECTS.

Result Callback

The server application provides remote objects with operations that have return values and/or may return errors. The result of the invocation is handled asynchronously.



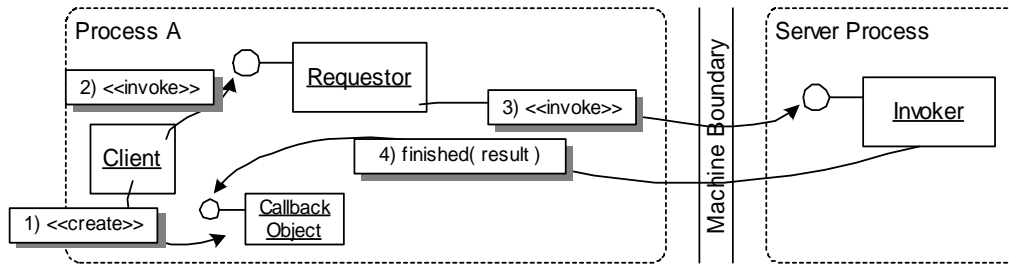
The client needs to be actively informed about results of asynchronously invoked operations on a remote object. That is, if the result becomes available to the REQUESTOR, the client wants to be informed immediately to react on it. In the meantime the client executes concurrently.

Consider an image processing example. A client posts images to a remote object specifying how the images should be processed. Once the remote object has finished processing the image, it is available for download and subsequently displayed on the client. The result of the processing operation is the URL where the image can be downloaded once it is available. A typical client will have several images to process at the same time, and processing will take different periods of time for each image – depending on size and calculations to be done.

In such situations a client does not want to wait until an image has been processed before it submits the next one. However, the client is still interested in the result of the operation to be able to download the result.

Therefore:

Provide a callback-based interface for remote invocations on the client. Upon an invocation, the client passes a RESULT CALLBACK object to the REQUESTOR. The invocation returns immediately after sending the invocation to the server. Once the result is available, the distributed object middleware invokes a predefined operation on the RESULT CALLBACK object, passing it the result of the invocation (this can be triggered by the CLIENT REQUEST HANDLER, for instance).



The client instantiates a callback object and invokes the operation on the REQUESTOR. When the result of the remote invocation returns, it is dispatched by the distributed object middleware to the callback object, calling a pre-defined callback method.



You can use the same or separate callback objects for each invocation of the same type. Somehow the correct callback object has to be identified. There are different variants, and it depends on the application case which variant works better:

- When you reuse a callback object you obviously need an *Asynchronous Completion Token* [SSRB00] to associate the callback with the original invocation. An *Asynchronous Completion Token* contains information that uniquely identifies the callback object and method that is responsible for handling the result message.
- In cases where a callback object is not reused, responses to requests, do not need to be further demultiplexed, which simplifies interaction and no *Asynchronous Completion Token* is necessary.

Using RESULT CALLBACKS the client can immediately react on results of asynchronous invocations. As in the case of POLL OBJECT, the server application has nothing to do with the specifics of result handling in the client. The use of RESULT CALLBACK requires an event-driven application design, whereas POLL OBJECTS allow to keep a synchronous programming model.

The RESULT CALLBACK pattern also incurs the liability that client code, namely the code doing the original asynchronous invocation, and the code associated with the RESULT CALLBACK, is executed in multiple

thread contexts concurrently. The client code therefore needs to be prepared for that, for example when accessing resources. For instance, a separate thread or process can handle asynchronous replies.

The callback itself can be implemented inside the client only; that is, the client executes an ordinary synchronous invocation on a remote object in a separate thread. When the invocation returns, the clients calls back into the provided callback object. Alternatively, a “real” callback from the distributed object middleware in the server application can be used. This requires that the client makes the callback object available as a remote object for the server-side distributed object middleware to invoke. Note that in both cases the implementation of the target remote object is not affected.

In network configurations where a firewall exists between client and server, callback invocations using a new connection for the callback are problematic, as they might get blocked by the firewall. There are two options to solve this problem:

- Use bidirectional connections that allow requests to flow in both directions.
- Let the callback object internally poll the remote object whether the result is available.

As the second option is fairly complex, the first option should be considered in practice.

The major difference between POLL OBJECT and RESULT CALLBACK is that RESULT CALLBACKS require an event-driven design, whereas POLL OBJECTS allow to keep an almost synchronous execution model.

Message Queue

Invocations and/or results of the invocations can be handled asynchronously.



Invoking operations asynchronously, using one of the asynchronous invocation patterns FIRE AND FORGET, SYNC WITH SERVER, POLL OBJECT, or RESULT CALLBACK, allows to avoid blocking clients. However, the patterns POLL OBJECT and RESULT CALLBACK can only be implemented on their own, if there is a separate process or thread for receiving the result asynchronously. Consider further you want to deal with (temporal) problems of the networked environment, such as network latency, network unreliability, or server crashes. Using the asynchronous invocation patterns, the client has to care for dealing with temporal unavailability of remote objects. None of the synchronous or asynchronous invocation variants can handle the order of invoking or receiving messages.

When implementing POLL OBJECT or RESULT CALLBACK in the client, you need some entity that asynchronously receives the acknowledgement or result. Spawning a new thread for receiving the result per POLL OBJECT or RESULT CALLBACK might be problematic, say, because you are running out of threads or you are simply not working in a multi-threaded environment.

A remote invocation consists of three participants: the invoking client process, the receiving server process, and the network connection. Each of these participants has to work reliably (at the same time) to make an invocation successful.

When using synchronous invocations, FIRE AND FORGET, SYNC WITH SERVER, POLL OBJECT, or RESULT CALLBACK, to tolerate temporal failures, a retry loop is required. But consider a lot of such requests and more than one potential failure. Then the retry loop should not retry all the time but only periodically, and should remember which invocation is next.

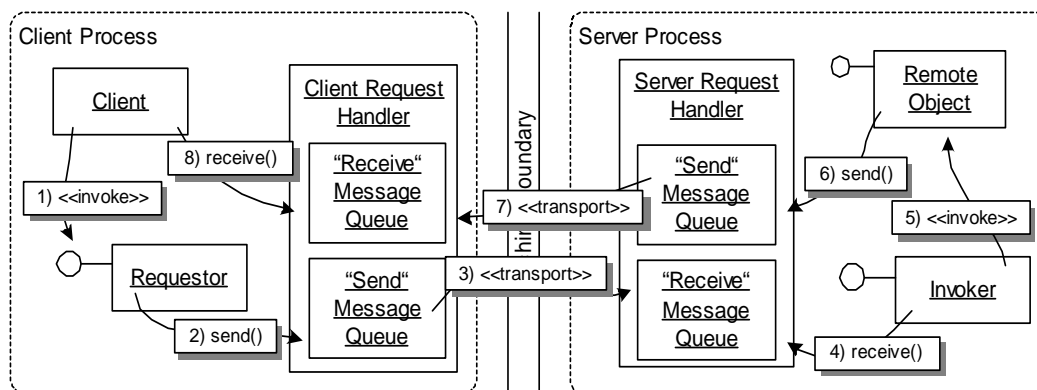
Consider further that only the remote server application fails. If the client has to periodically check, another problem is the use of network

bandwidth for a continuing retry loop, which is unnecessary, as the network and the remote distributed object system are functioning reliably.

The same problems also appear for returning the result.

Therefore:

Add MESSAGE QUEUES to the CLIENT REQUEST HANDLER and the SERVER REQUEST HANDLER, or to their PROTOCOL PLUG-INS respectively (perhaps as part of a larger *Messaging* system [HW03]). When the client sends a request, it is not transmitted immediately, but first put into a MESSAGE QUEUE. It is kept in this queue until it can be transmitted to the server side, and an acknowledgement is received. When a new request arrives at the server side, it is added to a MESSAGE QUEUE as well, and an acknowledgement is sent back to the client. Maybe immediately, maybe later in time, the server process consumes the message and invokes the remote object. Now the result has to be transmitted back to the client. It is sent back using the same scheme. First, it is put into a MESSAGE QUEUE on server side, then it is transmitted and put into the client's MESSAGE QUEUE, and eventually it is consumed by the client process.



A simple messaging system provides a "send" and "receive" MESSAGE QUEUE in both REQUEST HANDLERS. Requests and results are put into these MESSAGE QUEUES. The messages are transport asynchronously across the network. The client and the server process actively consume the messages, whenever they like to.



Custom MESSAGE QUEUES can be implemented in the REQUEST HANDLERS. They can also be part of a PROTOCOL PLUG-IN, for instance, when a dedicated messaging protocol is used - and other protocols without message queuing are supported by the distributed object middleware as well.

MESSAGE QUEUES enable us to handle multiple invocations and/or multiple results in one thread of control asynchronously. When a result needs to be received, the client can either block on the MESSAGE QUEUE, or use one of the two pattern POLL OBJECT or RESULT CALLBACK for retrieving the result asynchronously.

Note that MESSAGE QUEUES enable a receiver, be it a client or a server, to pro-actively process a request or a result. The patterns RESULT CALLBACK and POLL OBJECT do the same - but only for the result processing in the client. With MESSAGE QUEUES, the same two options also have to be considered on server side:

- either the INVOKER can poll the MESSAGE QUEUE (this is also described by the pattern *Polling Consumer* [HW03]), or
- it can receive a callback when a new message has arrived (this is also described by the pattern *Event-Driven Consumer* [HW03]).

To a certain extent, MESSAGE QUEUES can also be used in context of SYNC WITH SERVER for queuing the acknowledgments. However, MESSAGE QUEUES at the communication layer will not work for SYNC WITH SERVER, as long as somebody between client and the communication layer, such as the REQUESTOR, blocks until the acknowledgement is received.

All four invocation asynchrony patterns, FIRE AND FORGET, SYNC WITH SERVER, POLL OBJECT, or RESULT CALLBACK, incur the risk of loss of ordering, when multiple invocations are issued. That is, a “younger” message may bypass some “older” one(s). If the order information is encoded in the message, MESSAGE QUEUES can be used to restore the order.

Compared to pure RPC implementations, MESSAGE QUEUES impose a performance overhead for queueing and dequeuing of messages.

In first place, using MESSAGE QUEUES in a distributed object middleware decouples the client and server processes. Sending a message does not

require both systems to be up and ready at the same time. As a result, we can tolerate temporal failures of the client (for result processing), the network (for transmission), and the server (for processing a request by the remote object).

As a drawback, using MESSAGE QUEUES means that a sender cannot assume that the request will be immediately served by the server because the request is queued up.

Usually MESSAGE QUEUES are of variable size, and thus they require dynamic memory allocations for queued invocations. However, MESSAGE QUEUES should be limited to avoid a memory overflow. More options for implementing MESSAGE QUEUES are discussed in [Her03], including memory management, optimizations (such as pooling or caching), and smart ordering.

We explain the relationships of MESSAGE QUEUES and patterns for messaging systems [HW03] more deeply in the *Related Concepts, Technologies, and Patterns* chapter.

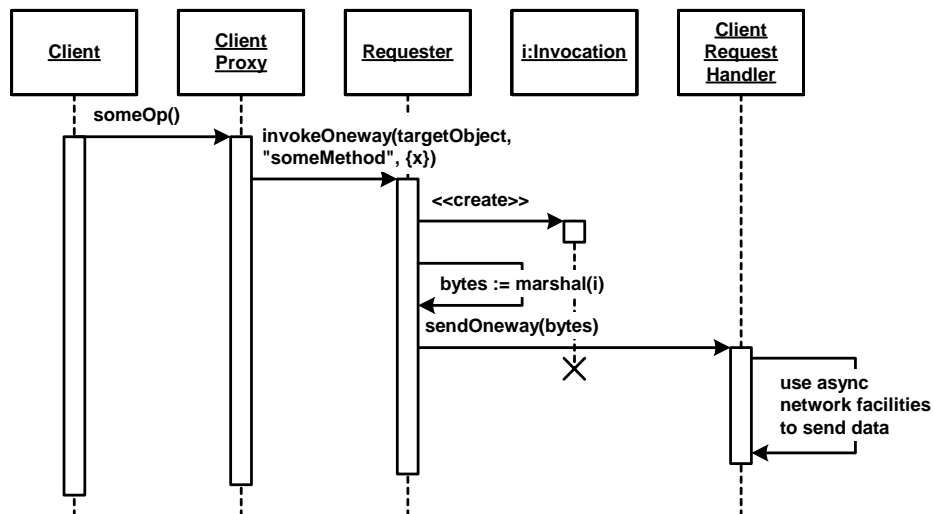
Interactions among the Patterns

The first four patterns described in this chapter (FIRE AND FORGET, SYNC WITH SERVER, POLL OBJECT, and RESULT CALLBACK) are virtually alternatives for each other - and they are all alternatives for synchronous (or blocking) invocations. In many distributed object middleware, more than one of them is supported, and it is the burden of the developer to select for the proper asynchrony pattern for the application task.

Note that changing the invocation variant used in a client application usually requires changing the client code as well. Thus the patterns cannot simply be exchanged as *Strategies* [GHJV95]. This is mainly due to the fact that the different invocation alternatives require different code for result handling (blocking for the result, ignoring the result, polling the result, or event-based reaction on callback events). Such differences require - sometimes severe - differences in the client code. For instance, to convert a complex blocking client to an invocation model based on RESULT CALLBACKS is not a trivial task. It requires implementing an event model, an event loop, and breaking down complex operations into sequences of subsequent callbacks. If the callbacks can execute in parallel, some mechanism for synchronizing incoming results might be required as well. These differences of the event-based programming model to conventional, blocking invocations are one of the motivations for using POLL OBJECTS - they provide some form of asynchrony while keeping the program “sequential” in nature.

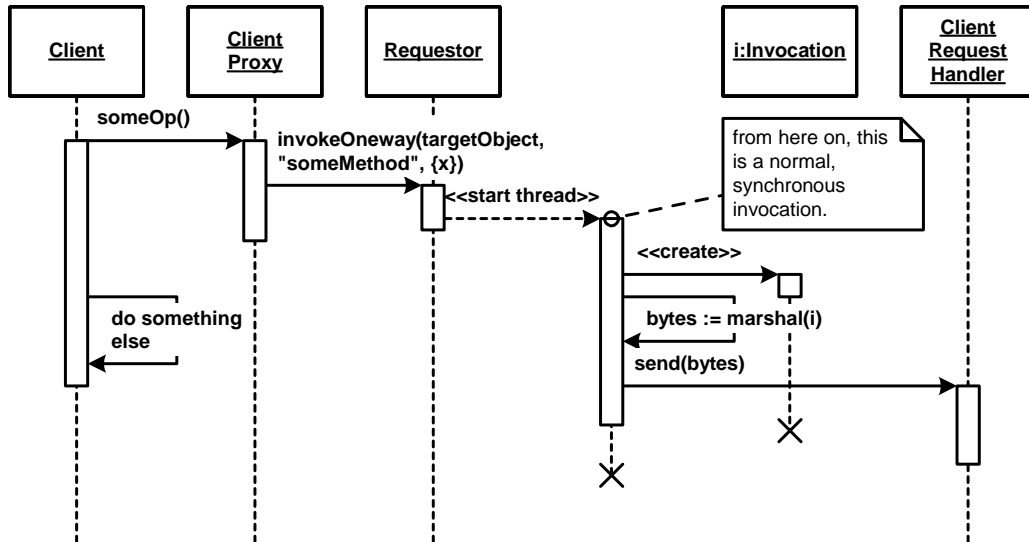
As pointed out in the MESSAGE QUEUE pattern description, MESSAGE QUEUES can be used in the CLIENT and SERVER REQUEST HANDLER, or in their PROTOCOL PLUG-INS respectively. Then MESSAGE QUEUES can be used together with synchronous invocations (blocking on the MESSAGE QUEUE) and all invocation asynchrony patterns (FIRE AND FORGET, SYNC WITH SERVER, POLL OBJECT, and RESULT CALLBACK) for sending requests on client side and receiving them on server side. They can also be used together with either POLL OBJECT or RESULT CALLBACK for queued, asynchronous result reception. If MESSAGE QUEUES are not supported at the REQUEST HANDLER layer, we can introduce queues at the invocation layer to support acknowledgment queuing for SYNC WITH SERVER or result queuing for POLL OBJECT or RESULT CALLBACK.

The following sequence diagrams show the different asynchronous modes in more detail. Let's start with FIRE AND FORGET. The first sequence diagram shows how FIRE AND FORGET can be implemented using asynchronous network facilities. We can see an ordinary client-side invocation - but instead of waiting for the result, the CLIENT-REQUEST HANDLER directly returns to the client.



Alternatively, the client can use multithreading to implement FIRE AND FORGET operations. The following diagram shows this variant. Here, the REQUESTOR returns immediately to the client and the separate thread handling the invocation terminates after the invocation is performed. We show multithreading on the level of the REQUESTOR. Alternatively, this could also be implemented on the level of the CLIENT REQUEST HANDLER. Note that the additional thread incurs an resource

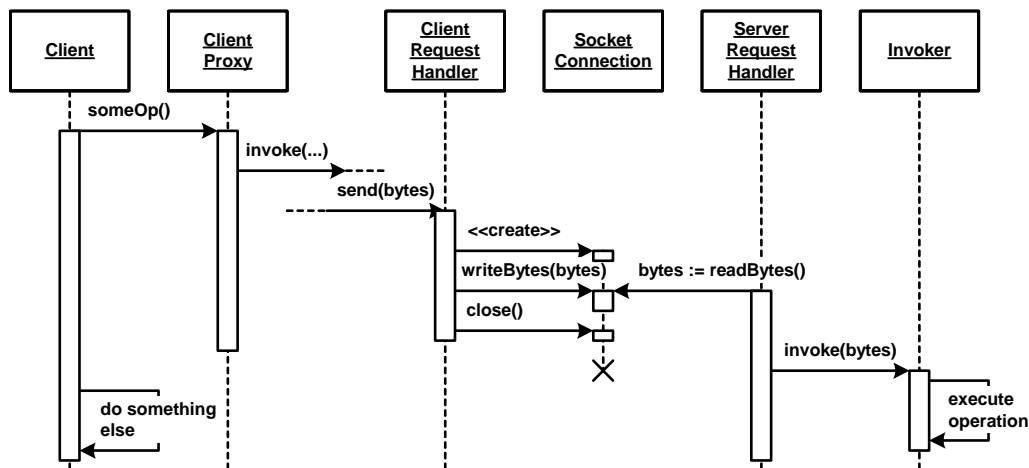
overhead, which might become problematic when a huge number of FIRE AND FORGET operations are sent at the same time.



The next sequence diagram shows an example of SYNC WITH SERVER. Again, there are two variants, one using network facilities and the other one using multithreading, now on the server side.

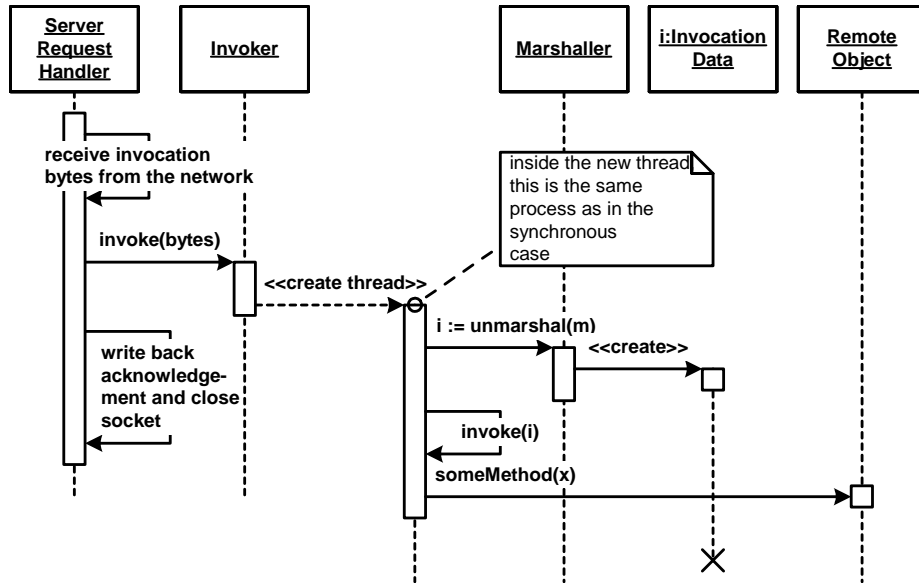
We show both alternatives in turn, starting with the one using network facilities. In this case, we do a normal synchronous method invocation. However, the CLIENT REQUEST HANDLER does not wait until the SERVER REQUEST HANDLER sends a reply, but closes the socket connection as

soon as the invocation bytes have been sent through the connection, and returns to the client.



The second alternative uses a separate thread within the server application, here illustrated inside the INVOKER. In that case, the invocation data sent over the network has to have a flag that instructs the INVOKER to execute the operation asynchronously. The flag is not shown in the diagrams. This way we get a synchronous acknowledgement, but the

invocation is performed asynchronously without having the client wait for a result.



The next couple of diagrams show examples how POLL OBJECT and RESULT CALLBACK can be realized. For each of the patterns, there are several implementation alternatives, which we will discuss briefly. We will then show the sequence diagrams for selected alternatives. Let us look at POLL OBJECT first. There are the following three alternatives:

- The client infrastructure can do a synchronous invocation, albeit in a separate thread. A POLL OBJECT is returned to the client. Once the result is available, it is put into the previously returned POLL OBJECT for client application access.
- The client infrastructure queries the server application each time the client polls the POLL OBJECT, which means a message is sent across the network from the client to the server.
- The server can use a separate invocation on the client once the result becomes available on the server. The client infrastructure can subsequently put the result into the POLL OBJECT to give the client access.

For the RESULT CALLBACK pattern, the same basic alternatives exist, except for the polling variant (the second alternative above):

- A normal synchronous invocation could be made in a separate client thread; when the result is there, the separate thread calls back into the provided callback operation.
- The client could invoke the server and pass along a flag that the call is meant to be asynchronous, and pass in a `RESULT CALLBACK` object. The server infrastructure directly calls back into the supplied object once the reply is available.

So what are the consequences of the different approaches? In the first alternative of both `POLL OBJECT` and `RESULT CALLBACK`, no changes are required in the server infrastructure and the invocation information sent over the network. Only the client needs to be adapted to handle the separate thread. While it is very appealing not to be required to change any of the server and communications infrastructure, there is of course a drawback: in case many parallel asynchronous invocations are made by a single client, many threads are necessary. A large number of threads can become a performance problem; thus, this alternative might not scale well in all scenarios.

The second alternative for the `POLL OBJECT` pattern has the obvious drawback that many network hops might occur; each time the client polls, a message is sent to the server application. Also, the server infrastructure must be changed to allow the client to poll. Some kind of token must be managed to allow the client to query the completion status of a previously invoked operation. While this approach has many obvious drawbacks, it has the advantage that no separate threads are required (on the client or on the server application), and also the client does not have to act as a server to host the callback remote objects. Hosting a remote object for the server application to call in might not seem a problem at the first look, but there is a price to pay: in several distributed object middleware the resources required to be a server application are much higher than for pure clients, both in terms of required program code (size) and runtime overhead. This can be prohibitive of the third alternative.

The third scenario (again for both patterns) has two advantages: First, there is no unnecessary network traffic. Only the callback message that is sent by the server has to be transported. Second, no threading overhead is required in the client. However, there are also obvious drawbacks. The server infrastructure has to be adapted to actively send

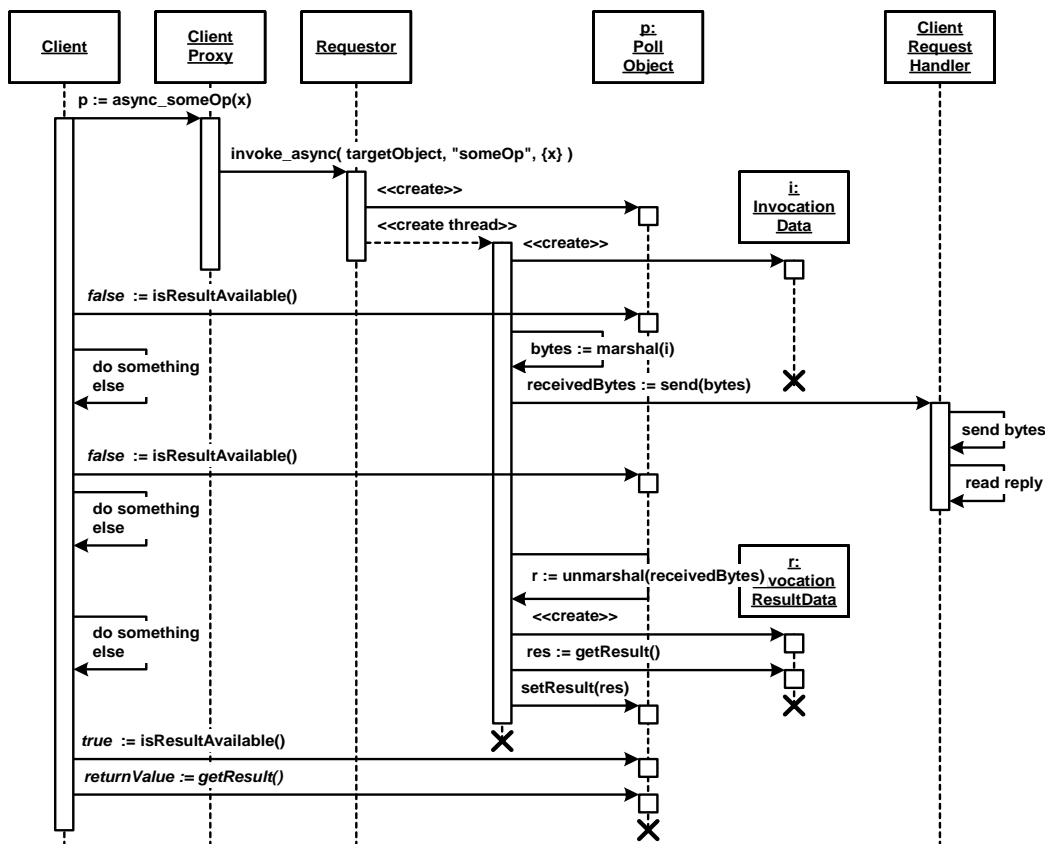
back the callback message (to the correct client's RESULT CALLBACK object) once the operation result is available. And what might be more critical is that the client now becomes a server application, because it has to "serve" the POLL OBJECT. The problems of this approach have been discussed above.

Note that in all the implementation alternatives, the remote object implementation or interface does not change at all. All the asynchrony is handled by some part of the infrastructure.

The point of the above discussion was to show that the programming model of the two alternatives (POLL OBJECT and RESULT CALLBACK) is independent of its implementation. Both patterns can be implemented either way, of course with respective advantages and drawbacks as explained above. As a consequence, we will show the sequence diagrams only for two of the alternatives: We will show the first alternative for POLL OBJECT and the third alternative for RESULT CALLBACK.

We will start with the POLL OBJECT example. We assume that the CLIENT PROXY provides an additional operation for each operation on the remote object, prefixed with the *async_* tag that invokes the remote operation using a POLL OBJECT. The operations have been generated into the proxy after the developer has specified a flag in the proxy generator, indicating that he wants to invoke the operations asynchronously. In this example the REQUESTOR creates a new thread and returns immediately to the client. The thread handles the invocation - as usual - and writes the result to the POLL OBJECT. This object is polled

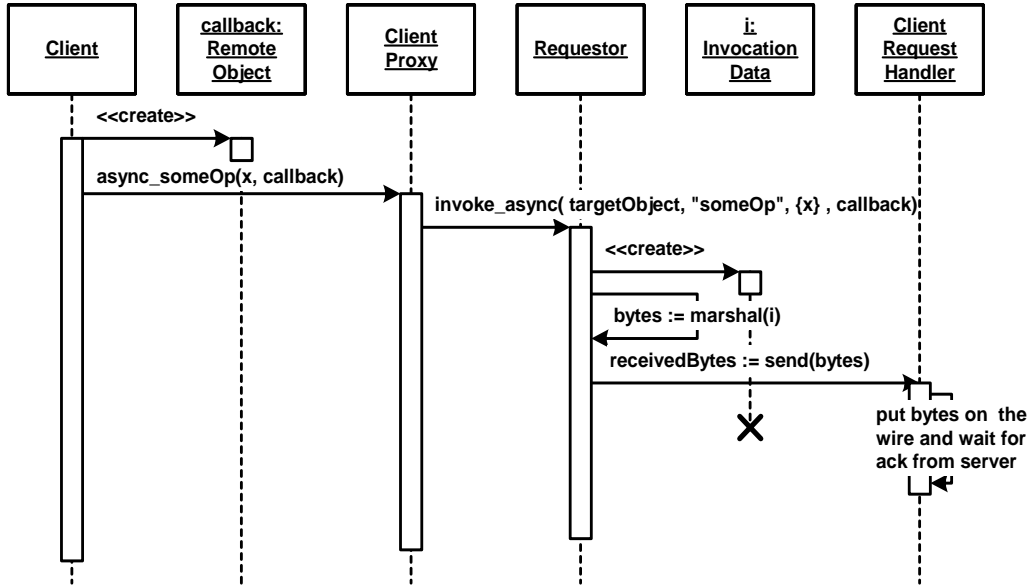
by the client and after the result arrives, the client can obtain it using the operation `getResult()`.



The sequence diagram above shows the POLL OBJECT dynamics in case it is implemented with client-side asynchrony. If we were to implement RESULT CALLBACK that way the only difference would be that the client supplies the callback object to the `async_` operation. The infrastructure would then call the callback operation from the separate thread.

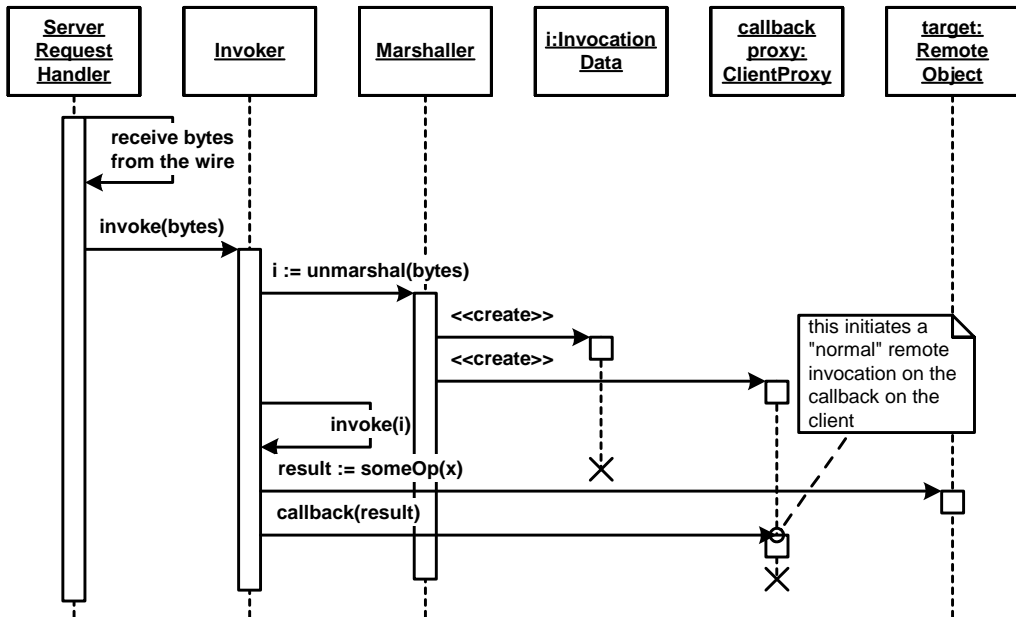
The next section shows the implementation of RESULT CALLBACK using a “real” callback from the server infrastructure. The client instantiates a separate callback object. Alternatively, the client could implement the required callback operation. Also, we assume that there is a second variant of the `async_...` operation that takes the callback object as a

parameter. The following diagram just shows the request sending part of the complete RESULT CALLBACK interaction.



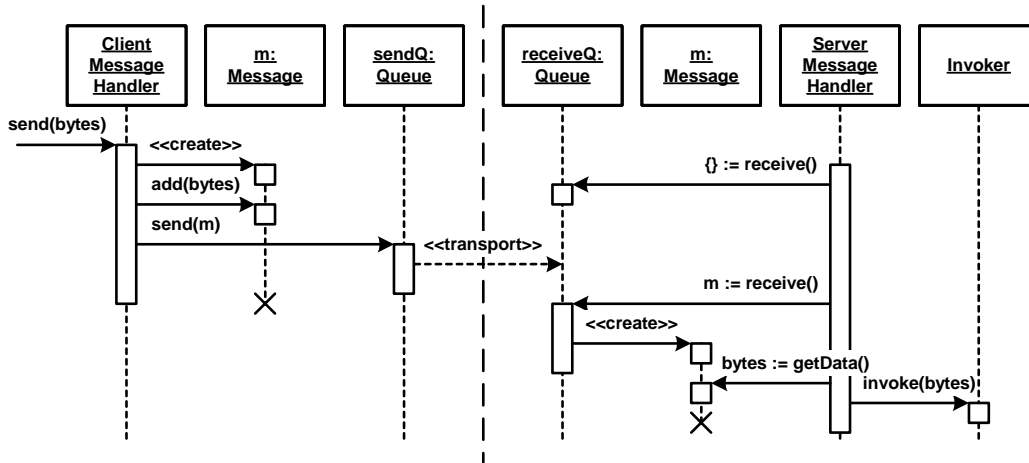
With regards to the request, we assume at least SYNC WITH SERVER semantics; the invocation only returns after the server application has acknowledged the receipt. The callback remote object is now accessible by the server infrastructure. A proxy is constructed in the usual way (using the ABSOLUTE OBJECT REFERENCE passed in the request), and the callback operation is called on it. As a final assumption, we also assume that the SERVER REQUEST HANDLER handles multithreading, so that no

separate thread needs to be created in the INVOKER to handle the callback.



The last part of this section deals with MESSAGE QUEUES. The next sequence diagram shows how MESSAGE QUEUES can be used inside the CLIENT and SERVER REQUEST HANDLER to achieve basic messaging. We

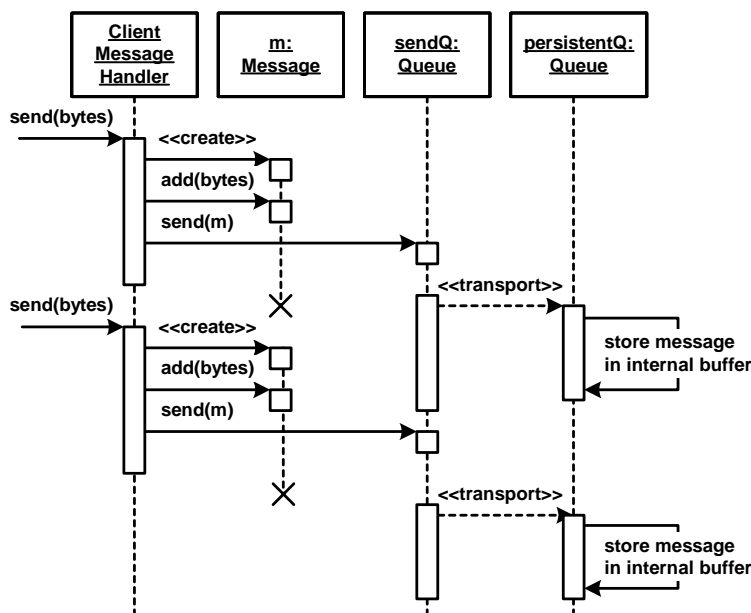
can see that intermediate message queue are added to the REQUEST HANDLERS to queue up requests on client side and server side.



Simple queueing, as the figure above, is only handling synchronization issues. Note that such simple queueing is already very powerful when used together with an event system: it can be used to overcome the problems identified for POLL OBJECTS and RESULT CALLBACKS above. Requests are put into a MESSAGE QUEUE and handled as events by a previously defined number of threads. Incoming results are handled as events again and put into the result MESSAGE QUEUE, again to be processed by a pre-defined number of threads. This variant realizes client-side asynchrony with POLL OBJECTS or RESULT CALLBACKS, but does neither require the client to be a server nor does it require one thread per request. However, MESSAGE QUEUES with an event system are more complex to implemented than using a simple thread per request model.

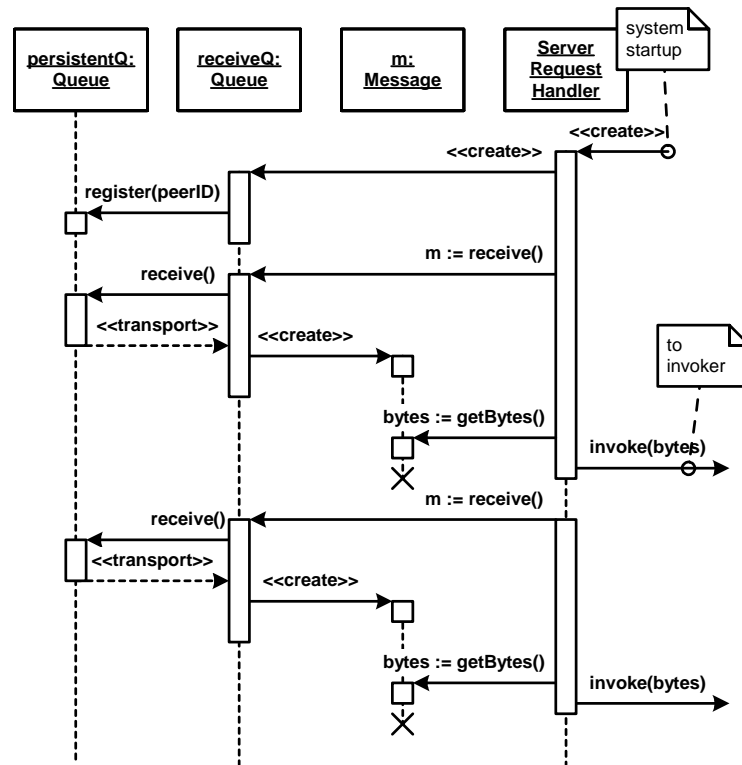
Another reason for introducing MESSAGE QUEUES is temporal decoupling. In the following scenario, we use a persistent intermediate queue that stores messages in case nobody receives them. Once the receiver connects to the queue, the messages are delivered exactly once. The

first diagram shows the sender's part. The most important addition is that messages are stored in a persistent queue - for later processing.



Note that the forwarding of messages to the persistent queue is done asynchronously; the CLIENT REQUEST HANDLER returns as soon as the message is delivered to the send queue in the client. It transports the request asynchronously to the persistent queue.

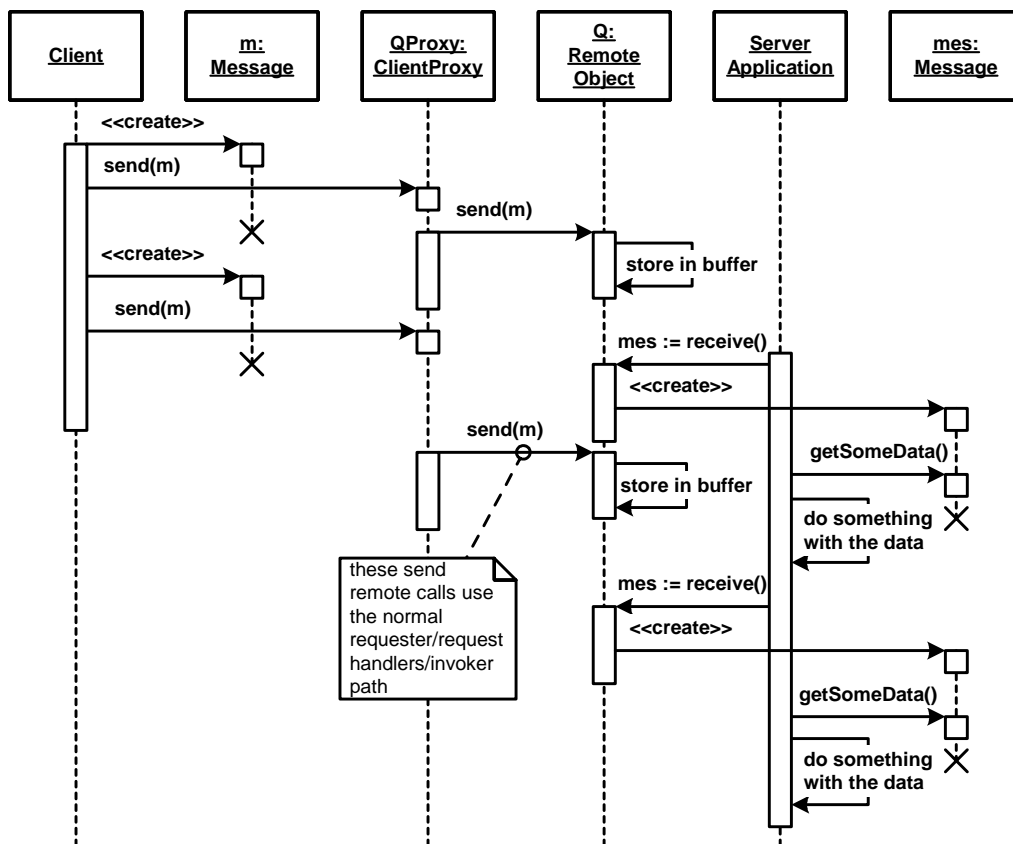
On the server side, upon system startup, the receive queue is created and connects to the persistent intermediary queue to which the client added messages before. When the SERVER REQUEST HANDLER calls the receive operation, the messages are retrieved from the intermediary queue. Note that this happens completely temporarily independent from the client's sending of the messages.



In the above examples, we use the MESSAGE QUEUE as a transport mechanism for remote invocations. As a consequence, there will typically be return values for the operations. These will be sent back as separate messages using the same mechanisms. As a consequence of the temporal decoupling, the response will be sent back whenever the server application starts up and processes the request. Thus, the client has to be prepared to handle the result whenever it becomes available. That is, using MESSAGE QUEUES in the REQUEST HANDLER only makes sense, if clients use one of the previously described asynchronous invocation patterns - either POLL OBJECT, RESULT CALLBACK, or FIRE AND FORGET (where no result is received).

Of course it is also possible to implement a messaging solution on top of (synchronous) remote invocations - that is, within the application layer of a distributed object middleware. The following diagram illus-

trates a possible scenario. In this example, a MESSAGE QUEUE is offered as a remote object.



As a final remark, it is of course also possible to use intermediate queue remote objects that provide persistent storage of messages. For instance, in many enterprise application integration architectures legacy systems have to be integrated. These systems often do only support synchronous processing. A remote object implementing a MESSAGE QUEUE can be used as a wrapper that queues invocations to the legacy system and sends reply messages to the clients once results are available.

Related Patterns: Invocation Asynchrony

When accessed asynchronously, a remote object is very similar to an *Active Object* in [SSRB00]. An *Active Object* decouples method invocation from method execution. The same holds true for remote objects that use any of the above presented patterns for asynchronous communication. However, when remote objects are invoked synchronously the invocation and execution of a method is not decoupled, even though they run in separate threads of control.

Active Objects typically create *Future* [Lea99] objects that clients use to retrieve the result from the method execution. The implementation of such *Future* objects follows the same patterns are presented in RESULT CALLBACK and POLL OBJECT.

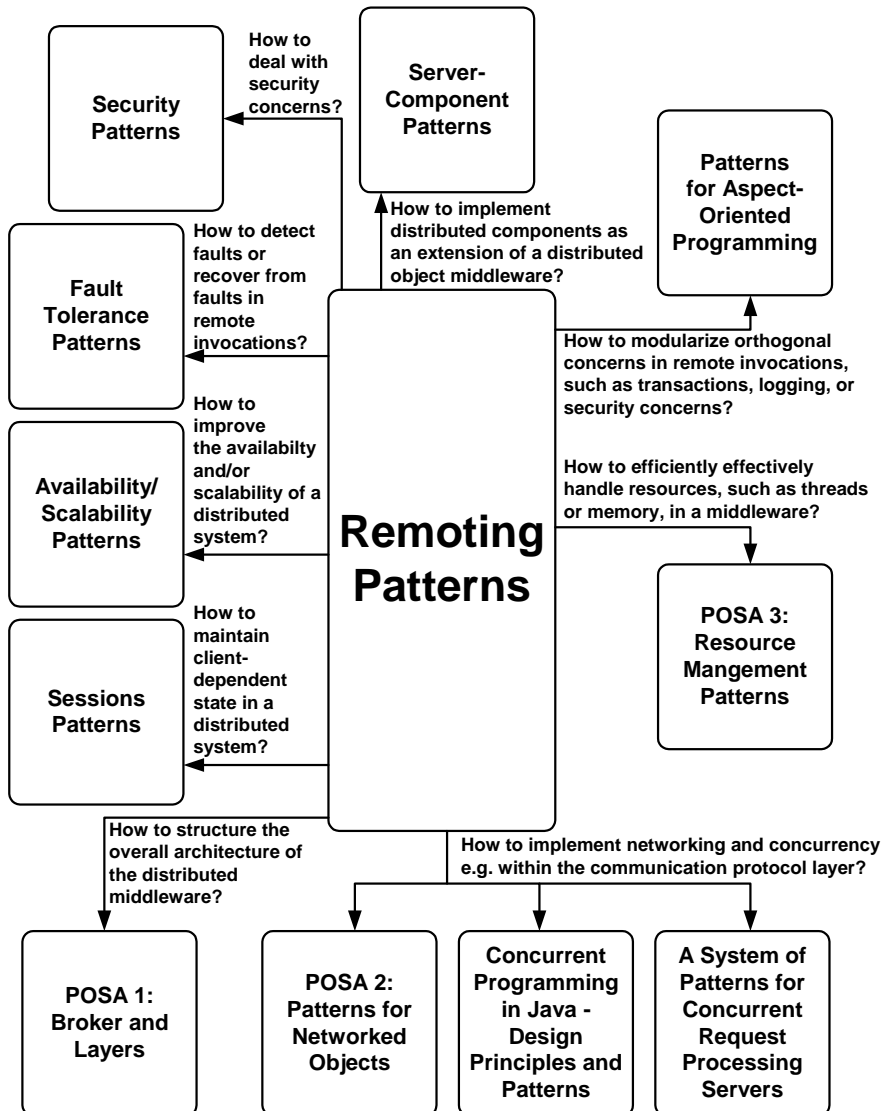
In the case of a RESULT CALLBACK, an *Asynchronous Completion Token* [SSRB00] can be used to allow clients to identify different results of asynchronous invocations to the same remote object.

13 Related Concepts, Technologies, and Patterns

The patterns in this book do not stand alone. There are several related technologies, concepts, and domains, when building distributed object middleware. Many of the best practices in these areas have already been captured in the form of patterns and pattern languages. While the detailed explanation of those would go beyond the scope of this book, this chapter gives a brief overview of some important areas; where possible we reference existing pattern material.

The figure below shows the relations of the Remoting Patterns to patterns from other domains as a guide to this chapter. Many of the related fields are already well captured by other pattern languages - thus the Remoting Patterns are a glue, filling potentially missing links, between these patterns and pattern languages, when they are applied to distributed object middleware or distributed application development. Especially the patterns for the internals of a communication middleware, such as network communication, concurrency, and resource management, are fairly complete. There are many patterns for orthogonal extensions, such as fault tolerance, scalability, and session management existing - however some domains such as security or transactions in distributed systems are not as well captured by patterns yet (a security patterns book is forthcoming but was not available at the time of this writing). Similarly, there is still work to do regarding pattern languages for systems built on top of distributed object middleware. For instance, there are only a few patterns for P2P systems, GRID computing, or aspect-oriented programming (AOP) available yet. It is quite natural that mature patterns are missing for the latter cases because patterns are describing established knowledge and fields like

P2P, GRID, or AOP are still emerging. We expect patterns for these fields to be appearing in the future.



Related Patterns

In this section, we provide a brief overview of patterns and pattern languages that are directly related to patterns in this book.

Networking and Concurrency

Distributed object middleware follows two architectural patterns documented in POSA1 [BMR+96]: The *Broker* pattern, in terms of mediating object invocations between communication participants, and the *Layers* pattern, regarding the separation of responsibilities such as connection handling, marshalling, decomposition, and dispatching of invocations. Distributed object middleware operates on top of network protocols, such as TCP/IP. For this it does not reinvent the wheel, but handles network connections and the inherent concurrency of networked communication using existing implementations and concepts. A number of patterns for networking and concurrency are documented in the following literature:

- POSA2 [SSRB00] contains many patterns that are used to implement distributed systems. This book described the most relevant patterns of this pattern language in a number of patterns presented, for instance in the Chapter *Basic Remoting Patterns* as the constituents of the CLIENT and SERVER REQUEST HANDLER. Among them are *Reactor*, *Half-sync/Half-async*, *Leader/Followers*, *Monitor Object*, and *Active Object*.
- Doug Lea's book *Concurrent Programming in Java - Design Principles and Patterns* [Lea99] describes several concurrency patterns with a special focus on Java. *Guarded Suspension* serializes access to shared objects whose methods can only be executed when certain conditions hold true; the pattern can be used to serialize access to remote objects and other resources that are concurrently accessed. The *Future* pattern provides a generalized form of POLL OBJECT; it provides a *Proxy* that can be used to query for results that are computed asynchronously.
- The patterns in the paper *A System of Patterns for Concurrent Request Processing Servers* [GT03] documents several patterns for concurrent request handling in high-performance servers. The pattern *Forking Server* describes a typical structure of a SERVER REQUEST HANDLER in which one listener process/thread listens to the network port and forks a worker process/thread for each

incoming request. The worker thread may be obtained from a *Worker Pool*. The pattern *Job Queue* applies queueing between listener and worker threads; the resulting architecture is similar to a receiving MESSAGE QUEUE that is part of a SERVER REQUEST HANDLER.

Resource Management

POSA3 [KJ04] deals with patterns for resource management and optimization. It documents a pattern language on how to efficiently and effectively acquire, access, and release resources at different layers of abstraction. That is, the book looks at management of any kind of resource, ranging from typical operating system resources such as threads or connections, up to remote objects or application services.

Acquisition patterns, such as *Lazy Acquisition*, *Eager Acquisition*, and *Partial Acquisition*, document best practices and strategies for resource acquisition. They address non-functional properties, such as scalability, availability, and predictability. The patterns are used in distributed object middleware at several places: *Lazy Acquisition* for remote object instances; *Eager Acquisition* for memory, connection, and thread resources; *Partial Acquisition* for byte streams of large invocations.

The *Lookup* pattern documents how a system can be made flexible by decoupling its elements from each other. It is used to announce and find instances of remote objects.

Patterns for managing the lifecycle of resources, such as *Caching*, *Pooling*, and *Resource Lifecycle Manager*, elaborate on how to increase system performance, while saving the developer from tedious resource management activities. *Caching* is typically used in distributed object middleware, when connections to seldomly used servers are kept available to avoid re-acquisition overhead. *Pooling* is used for managing remote object instances, but it is also applied at lower levels, for example for thread and connection management inside CLIENT and SERVER REQUEST HANDLERS.

The *Coordinator* pattern explains how to ensure consistency among any kind of resource. It is used only in advanced implementations of distributed object middleware, for example when transactions have to be supported as an additional service.

The resource release patterns, *Leasing* and *Evictor*, illustrate how resources can be released without manual intervention from a resource user. *Leasing* is regularly applied to manage resource release of remote objects by clients, whereas *Evictor* is typically used for thread and connection resources inside CLIENT and SERVER REQUEST HANDLERS.

In the patterns LOOKUP, LAZY ACQUISITION, POOLING, and LEASING we focus on the application of resource management to remote objects, even though the patterns are not limited to these issues, as discussed above.

Sessions

Sessions deal with a common problem in the context of distributed object middleware: Client-dependent state must be maintained in the distributed object middleware between individual accesses of the same client. While sessions can exist at any protocol level, they are mostly independent of lower level communication tasks, for example when multiple client objects share the same physical network connection.

The *Session* pattern [Sor02] provides a solution to this problem: State is maintained in sessions, which are maintained between individual client requests, so that new requests can access previously accumulated data. A session identifier lets clients and remote objects be able to refer to a session.

Generally, sessions can be maintained in the server or in the client. If the session is maintained in the server, the session identifier is sent with the each response to the client, and the client refers to it in the next invocation; if it is maintained in the client, the client has to send it to the server, which refers to it in its responses. Though, in distributed object middleware sessions are typically maintained in the server. Clients and remote objects use the same session identifier in requests and responses as part of the INVOCATION CONTEXT.

On the server, sessions can be implemented either on the application level, in the form of CLIENT-DEPENDENT INSTANCES, or as part of the distributed object middleware in the form of actual session objects.

Those session objects must be accessible through the INVOCATION CONTEXT, which is maintained by the INVOKER transparent to the

remote object. The interactions section of the *Extension Patterns* chapter shows an example of its usage.

The lifecycle of session objects and CLIENT-DEPENDENT INSTANCES must be managed by LEASING in order to get rid of sessions that are no longer needed, because the respective client has gone away.

Distribution Infrastructures

There are many distribution infrastructures that are implemented on top of a distributed object middleware. This section takes a look at prominent examples, such as transaction processing monitors, component infrastructures, peer-to-peer computing, grid computing, code mobility, and messaging.

Transaction Processing Monitors

Transaction processing monitors (TP monitors) are one of the oldest kinds of middleware. They provide the infrastructure to efficiently and reliably develop, run, and manage distributed transaction applications. For many decades, TP monitors were the dominant form of middleware, and many other middleware products available today have some link to a TP monitor product.

One of the earliest TP monitors was IBM's Customer and Controller Systems (CICS) [Ibm03], developed in the late 1960's and still in use today. The main purpose of TP monitors is to extend a distributed application with the concept of *transactions*. Transactions were developed in the context of databases, and describe a series of operations that have so-called ACID properties. ACID is an abbreviation that stands for a number of desirable properties of a transaction:

- *Atomicity*: A transaction is treated as an indivisible unit, and it is either entirely completed (committed) or undone (rolled back).
- *Consistency*: When the transaction terminates, the system must be in a consistent state.
- *Isolation*: A transaction's behavior must not be affected by other transactions.

- *Durability*: Changes are permanent after a transaction successfully terminates, and these changes must survive system failures (also called *persistence*).

A distributed transaction involves more than one transactional resource, such as a database, usually residing on more than one machine. Conventional RPC does not support the transaction abstraction. Thus RPC treats all invocations as if they were independent from each other. This makes it hard to recover from partial failure and enforce consistent, complete changes to distributed data. TP monitors, in contrast, allow developers to wrap a series of invocations into a transaction. They can thus guarantee either “at most once”, “at least once”, or “exactly once” semantics for an invocation.

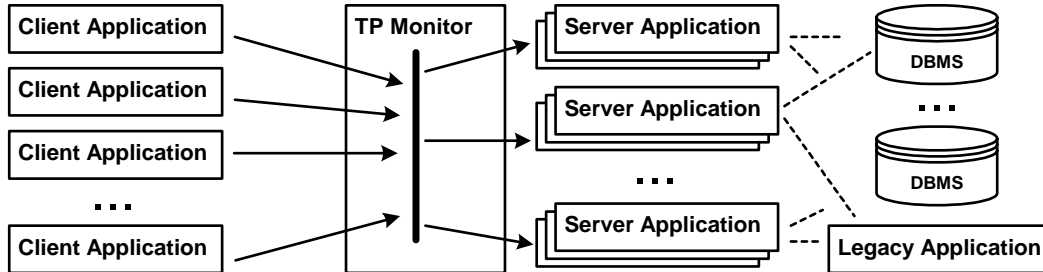
Transactional invocations are typically realized by marking the beginning and the end of a transaction in the client code. At the beginning of the transaction, the client contacts the TP monitor to acquire a transaction ID and context to be used throughout the transaction for the respective remote invocations. At the end of the transaction, the client notifies the TP monitor again, and then the TP monitor runs a commit protocol to determine the outcome of the transaction.

All transactional invocations must be marked to belong to a transaction using a transaction ID. This can be done by the INVOKER and REQUESTOR/CLIENT PROXY. A more elegant way is to use interceptors: INVOCATION INTERCEPTORS can transparently add the transaction ID on client side and read it out on server side. The transaction ID can be stored in an INVOCATION CONTEXT.

In a distributed transaction, a two-phase commit protocol (2PC, see [Gra78]) can be used for distributed transaction processing. Two-phase commit guarantees the ACID properties and supports distributed synchronization of multiple transactional resources. Finally, the client is informed whether the transaction was successfully committed or not.

TP monitors map the request of numerous clients to server applications and/or backend resources. An important goal of commercial TP monitors is to improve the distributed system’s performance. In addition, TP monitors include management features, such as process recovery by restarting, dynamic load balancing, and enforcing consistency of distributed data. Many TP monitors support a number of different communication interfaces to be used by clients and servers, such as

RPC, conversational peer-to-peer communication, message queues, and publish/subscribe.



As shown in the figure above, the TP monitor breaks the direct connection between a potentially huge number of clients and application servers/back-end resources. It orchestrates the actions of all the participants and makes them act as part of a distributed transaction.

Early TP monitors were monolithic, single process systems. Today almost all TP monitor products use a 3-tier architecture. There are many commercial TP monitor products, including IBM's CICS [Ibm03], BEA's Tuxedo [Bea03], Transarc's Encina [Tra00], and Microsoft's MTS [Mic03]. There are also many standards in this area, such as the X/Open (Open Group) Distributed Transaction Processing (DTP) model [Ope03] or the OMG's Object Transaction Service (OTS) [Omg04b].

Server Components

Server-side component infrastructures [VSW02] provide a distributed execution environment for software components. The execution environment is called a component container, or *Container*, for short. *Components* cannot be executed standalone, they require the container to provide essential services to them. These services handle the technical concerns of an application. Technical concerns are typically cross-cutting aspects that are not directly related to the application functionality implemented within the components. What exactly constitutes these technical concerns depends on the application domain. In an enterprise environment (where EJB [Sun03a], CCM [Sev03], or COM+ [Mic02] are typically used), the technical concerns are issues such as transaction management, resource access decision, fail-over, replication, and persistence. In embedded systems technical concerns comprise scheduling, energy management, and diagnostics [Voe03].

The main benefit of a component-based approach is that the component developer does not need to implement the technical concerns over and over again. The developer only specifies the container services required by a component using declarative *Annotations*, and the container makes sure these services are available to the deployed components.

Many containers are implemented for some kind of component standard, such as EJB [Sun03a], CCM [Sev03] or COM+ [Mic02], by professional container vendors, such as Microsoft, IBM, BEA, Oracle, or open source projects such as JBoss. Applications just use the containers as they are. Application developers thus don't need to be experts with respect to the (non-trivial) technical concerns. The following paragraph lists essential building blocks for component infrastructures. For a more detailed explanation see [VSW02] (we also present an overview of this pattern language in the Appendix). Sometimes it is also useful to implement a domain-specific or even project-specific component infrastructure - see [Voe03].

Component infrastructures are often built on top of distributed object middleware as described in the pattern language in this book. They can be seen as an extension. Using INVOCATION INTERCEPTORS, INVOCATION CONTEXTS, as well as suitable LIFECYCLE MANAGERS, most of the functionality of component infrastructures can be built.

The advantage of using component infrastructures is that they already provide useful default implementations of important, recurring features. They also provide easy access and simple configuration - using *Annotations* - for the developers of a specific domain.

In case you want to build your own component infrastructure, starting with a distributed object middleware as a basis is certainly a good idea. The current mainstream component infrastructures have also evolved from (and are built on top of) distributed object middleware systems: EJB is based on Java RMI, CORBA Components are based on CORBA (and specifically, the POA for lifecycle management), and COM+ uses DCOM as a communications mechanism. Using distributed systems as a basis for component infrastructures is also an example of reuse.

Peer-To-Peer Computing

Peer-To-Peer (P2P) systems are different from other distributed systems, as they do not base on a client/server or n-tier model. Instead of having centralized servers offering services, a set of equal peers communicates and coordinates themselves in fulfilling user services. Many P2P systems use remote objects internally for communication between peers.

The most popular P2P system is certainly (the original) Napster [Rox03]. It is a distributed music-sharing system that uses a centralized search index. There are several more or less direct clones, such as OpenNap [Ope01] or Gnutella [Gnu01], not all limited to sharing MP3 files. There are a number of P2P projects. SETI@home [Set03], for instance, uses a world-wide client community of volunteers, who donate the processing power of their machines, to find signs for extra-terrestrial intelligence in radio astronomy data. JXTA [Sun03b] provides a Java implementation of a P2P infrastructure. Jini [Jin03] and UPnP [Upn03] are architectures for spontaneous networking and can be used for building P2P networks.

In pure P2P networks, there is no central server. Each participating node can provide and consume services. The overall state is distributed over the nodes. In other distributed object middleware, LOOKUP is implemented as a centrally provided lookup service with a well-known ABSOLUTE OBJECT REFERENCE. In contrast, in a P2P network, the lookup service is itself provided by each node or by many nodes of the P2P network. Different techniques are applied to obtain an initial ABSOLUTE OBJECT REFERENCE to another node or a number of nodes already linked to the P2P network; examples are broadcast messages and server lists that can be downloaded from the Internet. Once a link to the P2P network is established the lookup service of any node can be queried for resources or services in the network.

A service can be provided by a peer at any time, and they can also be removed; that is, the services offered by the peer are not provided anymore. Clients have to discover at any time, if and where a service is provided. A consequence is that flexible client side invocations are needed. For instance, Jini allows a remote object to publish its CLIENT PROXY in the lookup service. Clients can download CLIENT PROXIES from the lookup service. Thus a remote object can publish code that is later used from within clients to access the remote object. Another option to realize flexible client access is to use the pattern REQUESTOR only,

instead of using CLIENT PROXIES, because REQUESTORS allow to construct invocations dynamically.

In practice, there are several different hybrid systems, with centralized user database, search index, or a “work coordinator,” acting as *Coordinator* [KJ04]. For a thorough discussion of P2P systems see [CV02].

Grid Computing

Grid computing [Gri03, FKT01] is about sharing and aggregation of distributed resources, such as processing time, storage, and information. A Grid consists of multiple computers linked together to one system. Grid computing makes extensive use of remote communication, as the distributed resources need to be managed [KJ03] and computing requests and results need to be transmitted. Here the distributed object middleware plays an important role.

In order to find distributed resources, which are typically represented as remote objects, Grids use the LOOKUP pattern. Depending on the Grid, the role of the lookup service in the system is often referred to as resource broker, resource manager, or lookup service.

The architectures of available Grid computing projects [BBL02] and toolkits vary, but some of the underlying concepts are the same in most of them. Actually, large parts the architectures are very similar to those of ad hoc networks and P2P systems (see the discussion above).

Code Mobility and Agents

As discussed in several places in this book, remote invocations can exhibit a variety of limitations and drawbacks compared to local invocations, for instance, regarding performance, configurability, and scalability. Developers cannot easily get around these problems, because often they are faced with applications that inherently need distribution. Just consider a simple example: a client requests data from remote objects, and evaluates this data locally. Even though the client logic might be rather simple, it cannot be placed on the server because different clients need different information processing, and the server application developers cannot foresee all possible requirements. As a consequence, the client needs to fetch the data from the server and process it at the client side. That is, the client needs to send out a network request every time it needs to access data located at the server

and/or the data needs to be sent across the network. This might be slow and consume much network bandwidth (especially if the data is large). Code mobility resolves such problems by providing the ability of moving code across the nodes of a network. Thus the client code can migrate to the server, execute within the local server context, and return with the result. The custom client code thus executes at the same site where the data is located.

The ability to relocate code is a powerful concept; however, the field is still quite immature. Fuggetta, Picco, and Vigna provide a conceptual framework for understanding code mobility [FPV99]. They distinguish different code mobility paradigms:

- *Remote evaluation:* Code is sent to the server and executed in the server context using the data located at the server.
- *Code on demand:* Code is downloaded by a client and executed in the client context using data located at the client.
- *Mobile agents:* Code and associated data - maybe even the execution state - migrates from host *A* to host *B*, and executes in the context of host *B*.

These code mobility paradigms extend the client/server paradigm used in distributed object middleware concepts. Similar to other high-level remotng paradigms, code mobility can be realized on top of almost any existing distributed object middleware. For instance, in many existing mobile code systems, distributed object middleware, such as CORBA or RMI, or their protocols, such as IIOP, are used for actually send the code across the network.

The ability of a mobile code to change its location means that it is hard to locate the mobile code with a location-dependent ABSOLUTE OBJECT REFERENCE. Thus the LOOKUP and LOCATION FORWARDER patterns play a central role in mobile code scenarios because some entity is needed to inform a client of a new location of a mobile code.

For all code mobility approaches an extension to the MARSHALLER is required because the mobile code must be transferred with the messages in a format appropriate for network transmission. In some approaches native machine-code is transferred. This has the disadvantage that the code can only execute on the one particular platform it is compiled for. As an alternative, byte-code (such as compiled Java

classes) can be transferred that can execute on any platform for which the byte-code's virtual machine (such as the Java Virtual Machine) is implemented. These variants - machine-code or byte-code - are often used in statically compiled languages. In dynamic languages - that allow for behavior modifications at runtime - a *Serializer* [RSB+98] and/or *Introspection Options* [Zdu03] are required to obtain the current definition of the mobile code at runtime. That is, the program code to be transmitted is serialized. Then it is transferred to the remote host, where it gets interpreted and/or compiled at runtime.

Approaches, in which only code can migrate, are called *weak mobility* approaches [FPV99]. In contrast, *strong mobility* [FPV99] is the ability of migrating both the code and the execution state of an object. Strong mobility requires a MARSHALLER that is able to serialize the callstack state relevant for the code to be migrated, as well as the current state of the mobile code.

Mobile code approaches can exploit the security measures taken in most other distributed object middleware, such as encrypted connections and authentication, typically handled with PROTOCOL PLUG-INS or INVOCATION INTERCEPTORS. But there are further security requirements that cannot be simply handled by PROTOCOL PLUG-INS or INVOCATION INTERCEPTORS. If code can migrate to a host, you must secure the host environment and other mobile code from malicious mobile code. A solution is to let the mobile code execute in a restricted environment, such as a sandbox interpreter as in Safe Tcl [OLW98] or using the security mechanisms of a virtual machine, such those of the Java Virtual Machine. A more complex problem - not solved well yet - is how to secure the mobile code from attacks by a malicious host. A simple approach is to establish a net of trusted sites, for instance, with a third-party that acts as a certification authority.

There are numerous implementations of weak code mobility without transferring data. Exemplary approaches are Java applets or Jini Proxies (as explained in the P2P section above). In most scripting languages weak mobility with data transfer is frequently used, due to the *Introspection Options* [Zdu03] provided by those languages and the ability to evaluate code on the fly. Weak mobility with data transfer is also supported by a number of mobile code frameworks, including Aglets [LO98], Grasshopper [BM98], and Voyager [Obj97]. Strong

mobility is for instance supported by D'Agents [GCK+02] and Ara [PS97].

The agent concept, which means a computer program executes actions autonomously on behalf of a human, is often used together with code mobility. However, not all agents are mobile agents - some agent concepts only support stationary agents. In all agent approaches, there is the need for an agent to communicate with other agents. Many agents can thus access other agents in the same server context, and, at the same time, they are remote objects, meaning that they can be reached by agents running in other hosts using RPC invocations. Many agent frameworks support PROTOCOL PLUG-INS for communicating with other agents, using existing distributed object middleware and protocols, such as CORBA or RMI.

If agents need to exchange semantic information or multiple agent platforms need to interoperate, the usage of standardized RPC invocations might not be enough. For such cases, agent communication languages (ACL), such as FIPA ACL [Fip02] or KQML [Arp93], provide INTERFACE DESCRIPTIONS for sophisticated agent communication. ACLs stand a level above INTERFACE DESCRIPTIONS provided by distributed object middleware, such as CORBA or RMI, because they handle semantic rules and actions instead of only remote object invocations, and ACL messages describe a desired state in a declarative language, rather than the invocations used to obtain the desired state.

Messaging

The pattern MESSAGE QUEUE describes one of the key concepts when building a messaging system, such as IBM's WebSphere MQ (formerly MQ Series) [Ibm04], JMS [Sun04c], Microsoft's MSMQ [Mic04a], and Tibco [Tib04]. More fine-grained patterns on how to implement such systems are document in the book *Enterprise Integration Patterns* [HW03]. Messaging systems are related to distributed object middleware: messaging systems can be implemented by extending distributed object middleware, and distributed object middleware may use existing messaging systems inside PROTOCOL PLUG-INS. PROTOCOL PLUG-INS that integrate messaging systems are briefly described in the *Web Services Technology Projection*.

Messaging is inherently asynchronous. Therefore, at least one of the asynchronous invocation patterns FIRE AND FORGET, SYNC WITH SERVER, POLL OBJECT, or RESULT CALLBACK are typically supported in a messaging system. Without support for at least one of these patterns, a client would have to synchronously block on the MESSAGE QUEUE, when waiting for a result. Thus, for asynchronous invocations that return a result, one of the two patterns POLL OBJECT or RESULT CALLBACK is needed. In [HW03] the corresponding patterns are named *Polling Consumer* [HW03] and *Event-Driven Consumer* [HW03]. In our pattern description, we have referred to the *Asynchronous Completion Token* pattern to correlate a result invocation with an invocation; the [HW03] pendant is named *Correlation Identifier*.

Messaging systems use the abstraction of a *Message Channel* [HW03] as connection between a particular sender (client) and a particular receiver (server). When a client sends a message, it puts the message into a particular *Message Channel* and the server application receives messages from a particular *Message Channel* as well. To build a *Message Channel* on top of a distributed object middleware system, the REQUEST HANDLERS can be extended in one of the following ways:

- *Message Channels* can be implemented on top of “send” and “receive” MESSAGE QUEUES that are part of all REQUEST HANDLERS.
- Each *Message Channel* might have its own MESSAGE QUEUE.
- MESSAGE QUEUES are instantiated for each connection between two network endpoints by the respective REQUEST HANDLERS.

Clients and server applications usually do not directly interfere with low-level details of MESSAGE QUEUES and *Message Channels*. Instead the sender and receiver instantiate a *Message Endpoint* [HW03], a piece of custom code to connect to a *Message Channel*. The *Message Endpoint* abstraction handles all interaction with the messaging system. Client and remote object implementations only interfere with the *Message Endpoint*, not with the low-level details of messaging.

Messaging extends distributed object middleware in a number of ways; especially it provides for reliable message transport, order of messages, dealing with duplicated messages, expiration of messages, and different kinds of *Message Channels*. See the book *Enterprise Integration Patterns* [HW03] for more details on messaging systems and the related patterns.

Quality Attributes

There are a number of measures for improving quality attributes of a distributed object middleware. In the following section we discuss security, availability and scalability, and fault tolerance as important examples to be considered in distributed object middleware.

Security

When clients access remote objects over a public network, such as the Internet, they are vulnerable to security threats. The book *Distributed Systems* by Coulouris, Dollimore, and Kindberg [CDK94] enlists the following main security threads in the context of a distributed systems:

- the *leakage* or disclosure of information to non-authorized parties,
- the unauthorized *modification* of information in a distributed system,
- the unauthorized *use of resources* of the distributed system, and
- the *destruction* of information or (parts of) the distributed system.

Attacks are performed by attackers to exercise a thread. Typical attacks include *masquerading* to obtain the identity of legitimate users, *eaves-dropping* to listen to request messages, *interception* and *modification* of request messages, or *replaying* of messages. In order to protect a distributed systems against these and other attacks, there are a number of measures - ranging from organizational to technical measures (see [TS02, Emm00] for a longer discussion).

Technical security measures are pervasive, meaning that they can be applied at all layers of the distributed object middleware—from the REQUEST HANDLERS to the actual application layer.

A typical measure is encryption of request messages or message parts. For example, using a PROTOCOL PLUG-IN that bases on a secure transport protocol, such as SSL. Encryption—like most other security measures—can also be handled at higher-level layers of the distributed object middleware. Higher-level layers often provide and require the keys for encryption and decryption (and other security information).

Authentication techniques are used to establish trust in a remote invocations. This is done using so-called credentials, a number of security attributes that determine what a client (or server) is allowed to do.

Important examples for credentials are user IDs and passwords. Authentication has to be implemented using encryption to avoid sending the credentials in plain text over the network. Encryption and decryption of messages is typically handled by the respective `PROTOCOL PLUG-IN`. Encryption and decryption of further security information, typically contained in the `INVOCATION CONTEXT`, can be done by an `INVOCATION INTERCEPTOR`.

Access control techniques decide whether or not access to a resource, as for instance a remote object, can be granted to a client. The parameters for the server side access decision consist typically of the credentials, `OBJECT ID`, operation name, and parameters. The access control decision can be implemented in a specialized `INVOCATION INTERCEPTOR` or in the `INVOKER`. To make this possible, the credentials, or at least some kind of client authentication token, needs to be made available to the server-side distribution infrastructure. To transfer this an `INVOCATION CONTEXT` is typically used.

If access is not granted, a `REMOTING ERROR` containing the reason of the security error can be sent to the client to signal that access was not granted.

Using digital signatures, the users responsible for a client (or remote object) can be made accountable for their actions. If possible, digital signatures and other additional security information are placed in an extensible `INVOCATION CONTEXT` to transmit this information in a transparent fashion.

Security-relevant events should be logged to persistent storage so that security violations can be analyzed. Auditing policies determine which security information needs to be logged in a persistent storage. To perform this transparently in clients and servers, `INVOCATION INTERCEPTORS` are used.

Some technical security measures are handled outside the distributed object middleware. For instance, firewalls allow tight control of the message traffic between a private and a public network. If the invocations between client and server have to pass the firewalls, the firewall decodes and encodes requests and responses in either direction.

Note that technical measures for security are not enough: it is also necessary to take organizational measures, such as providing security policies and emergency response plans, and updating them regularly.

Improving Availability and Scalability

When services of a distributed application are deployed only on one machine, availability and scalability can become problematic. If the machine fails, the whole distributed system will fail. Also, under increased load conditions, one machine alone might not be able to provide enough performance. To deal with such situations, Dyson and Longshaw introduce patterns for building highly available and scalable distributed systems, especially Internet systems [DL03, DL04]. The targeted Internet systems comprise a broad range of systems: from a small personal web site to multi-national e-commerce sites.

The problem of availability can be resolved by introducing two (or more) *Functionally Identical Servers* that are configured as *Active Redundant Servers*. The two servers offer the same functionality, even though they might have different non-functional characteristics. One of the two servers is deployed as a redundant backup server that is only used in the event of failure or the need to maintain the active server. A switch is used to redirect traffic to the redundant server, if required. Thus clients can still access the same ABSOLUTE OBJECT REFERENCES, even if the main machine serving them is down. The switch in this example uses the pattern LOCATION FORWARDER in its simple form—it forwards invocations to one of the redundant servers.

Replication is simple for interaction with stateless remote objects. However, for stateful interactions (e.g. *Sessions* [Sor03]), the ongoing interaction needs to be maintained. *Session Fail-over* introduces a persistent session information store, available to both of the redundant servers. When the roles of the servers are switched, the redundant server starts using the session information that the previous active server persisted and continues the user interaction. The user will not perceive any interruption in its interaction with the server.

In addition to that, remote objects that read and write persistent state require data consistency as well. *Data Replication* lets the active server write the dynamic data into an associated database, but when one server writes dynamic data, the transaction is not completed until the

data is available on the other server as well. If one server is not available, the data will be written by the available server. When the previously unavailable server restarts, it needs to first synchronize its state.

The measures, mentioned so far, primarily improve availability. How to achieve a balanced load, even when some servers are not available? *Load Balanced Servers* are *Functionally Identical Servers* with a load balancer forwarding the requests to the servers, instead of a simple switch. The task of the load-balancer can be implemented using the LOCATION FORWARDER pattern. But instead of mapping the ABSOLUTE OBJECT REFERENCE to only one server at a time, the load balancer distributes remote invocations to all machines at the same time. The load balancer ensures that the performance is reasonably consistent, if a server fails or needs to be maintained.

Many systems use dedicated hardware for switches and load balancers that are proven, simple, fast, and easy to replace.

Fault Tolerance

Fault tolerance techniques are applied to detect errors of a system and recover from errors or mask errors of a system. To apply fault tolerance techniques it is important to understand the terminology, the underlying assumptions [Lap85, ALR01], and the kinds of faults tolerated by a technique. Saridakis presents basic fault tolerance techniques as a system of patterns [Sar02]. These are explained in this section.

Many fault tolerant systems are inherently distributed systems because the main means for fault tolerance is replication. A computation that should be made fault tolerant is mapped to a number of different units of failure. These units are usually located on different hardware units, such as different machines or processors. In other words, fault tolerance techniques primarily tolerate hardware failures—not software bugs. This is because software—in contrast to hardware—does not “wear out”. Note that there are fault tolerance techniques for tolerating software faults [Lyu95], such as recovery blocks or n-version programming.

Before a fault can be tolerated, it needs to be detected. There are a number of patterns for the detection of errors. Two important patterns are:

- The pattern *Fail-Stop Processor* replicates an input using a distributor and lets two (or more) processors perform the same computations. The distributor can be implemented as a simple kind of LOCATION FORWARDER mapping ABSOLUTE OBJECT REFERENCES to two (or more) replicated remote objects, running on different machines. The two (or more) results are compared. If the results are not the same, a fault has occurred in one of the processors (or during transmission of the message).
- Another way to detecting errors is an *Acknowledgement*. This pattern is implemented by additionally applying the pattern SYNC WITH SERVER that asynchronously invokes a remote object and only receives an *Acknowledgement*. Of course, a result is also implicitly an *Acknowledgement* - thus synchronous invocations contain an acknowledgement implicitly.

A number of patterns are used to detect and tolerate faults. There are two classes: those that recover from an error and those that mask an error.

Saridakis introduces the recovery pattern *Rollback* [Sar02]. The pattern assumes that there is some kind of test that can determine whether an error has occurred or not. If an error occurs, a replica of the system is used to perform the computation again. *Rollback* stores the state of the system at so-called checkpoints in a persistent storage. The pattern can be implemented by a remote object that knows how to split an invocation into a number of steps. This remote object acts as a LOCATION FORWARDER for each of these steps and forwards the invocation to another remote object, located in the active replica. For each step, it tests the result. When all steps are done, the result of the invocation is sent back to the client. When an error occurs, it triggers another replica.

Recovery solutions like Rollback assume that it is possible to test for an error. This, however, is not possible for all kinds of computations. Also, in some cases, such as real-time decisions, the loss of the computation result and a re-computation cannot be accepted. Consider a computation in the computer system of an aircraft, for example, that needs to produce a immediate result. To cope with this problem we can use error masking.

The most prominent error masking pattern is *Active Replication* that extends the configuration of a *Fail-Stop Processor*. To apply error

masking, the computation must produce a deterministic result. The idea is to compare a number of different results, produced by an uneven number of processor units. A distributor sends the input to all processors which perform the same computation. A comparator unit compares the results, and decides for the correct output using a majority vote. This way N simultaneous errors of $2N+1$ processors can be tolerated.

The pattern LOCATION FORWARDER can be used to realize the distributor unit of *Active Replication* (in the same way as explained for *Fail-Stop Processor*). A similar configuration of replicated remote objects with a LOCATION FORWARDER is described by the pattern *Object Group* [Maf96].

So far we have introduced a number of separate fault tolerant measures. Many highly dependable systems, such as aircrafts, however, use a combination of different fault tolerance measures, including redundancy, fail-stop processors, n-version programming, and others. For distributors, voters, and comparators simple (i.e. not complex) hardware units are used to minimize the risk of faults in these units.

Aspect-Orientation and Remoting

Aspect-oriented programming (AOP) [KLM+97] allows to modularize otherwise tangled cross-cutting design concerns. By this AOP becomes an important future trend in the domain of object-oriented remoting.

As an example in the area of remoting, consider a situation in which you want to expose a remote object in a server application, and you are faced with multiple, orthogonal extensions or adaptations of your original task. In the course of this book we have already discussed a number of such extension or adaptation concerns, such as logging, security, activation, lifecycle management, leasing, or QoS monitoring. Simple implementations hard-code these tasks directly in the distributed object middleware, making them tangled with the rest of the code, and scattered across many locations. The result of tangling and scattering is code that is hard to maintain and evolve.

AOP solves this problem by encapsulating the extension or adaptation concern in a separate design unit, called an *aspect*. The classes or components the aspect is applied to do not have to care about the

aspect's existence, they can stay "oblivious" to the aspect [FF00]. The fact that the artefact, to which the aspect is applied does not have to change is called non-invasiveness.

Many distributed object middleware systems use INVOCATION INTERCEPTORS to support extensional concerns. Interceptors are conceptually close to AOP and can be used to implement AOP solutions. A major difference between INVOCATION INTERCEPTOR architectures and AOP solutions is that INVOCATION INTERCEPTORS do not support obliviousness, but non-invasiveness—the application has to provide interception points/hooks, but does not have to care what is done inside those.

AOP can be implemented in different ways. Actually, the term AOP denotes a number of different adaptation techniques, and there are a number of different aspect composition frameworks and languages. In [Zdu03] a pattern language is described that explains how these aspect composition frameworks are realized internally. In [Zdu04a] a projection of this pattern language to a number of popular aspect composition frameworks can be found, including AspectJ [KHH+01], Hyper/J [Tar03], JAC [PSDF01], JBoss AOP [Bur03], XOTcl [NZ00], Axis [Apa03], Demeter/DJ [OL01], and others. These patterns for aspect composition frameworks can also be used to implement aspect solutions for distributed object middleware. Note that JAC and JBoss AOP provide aspect-orientation specifically in the context of distributed object middleware and server side components.

14 Technology Projections

In the pattern language presented in earlier chapters, we did not provide known uses for each of the patterns, but just generic examples. We provide larger examples for the whole pattern language in the next chapters – we take a look at a specific technology and show how the patterns are applied and how they relate to each other. This is done in order to emphasize the focus on the pattern language as a whole – instead of focusing on single patterns within the language. Instead of calling these chapters “Examples for the pattern language” we call them *technology projections*¹.

Each subsequent technology projection is intended to emphasize specific features of the pattern language:

- *.NET Remoting*: .NET Remoting provides a generally usable, flexible and well-defined distributed object middleware. It has a nice and consistent API and can easily be understood even by novices. It is already widely used and thus an important remoting technology. We use C# as the example programming language.
- *Web Services*: Web Services are currently one of the hottest topics in the IT industry. Basically, they provide a HTTP/XML based remoting infrastructure. This technology projection focuses especially on interoperability. Java and Apache Axis are used for most of the examples, but we also discuss other Web Service frameworks, such as .NET Web Services, IONA’s Artix, and GLUE.
- *CORBA and Real-time CORBA*: CORBA (and RT CORBA) is certainly the most complex, but also the most powerful and sophisticated remoting architecture available today. In addition to being language independent and platform interoperable, there are also implementations for embedded and real-time applications.

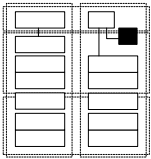
1. The term “technology projection” is according to a term that we first heard from Ulrich Eisenecker.

This technology projection will focus especially on these quality of service aspects. C++ is used in the examples.

Please note that these chapters cannot serve as complete and full tutorials for the respective technologies. They are really just meant to help you understand the patterns, as well as the commonalities and trade-offs of the respective technologies.

Reading the respective technology projections will of course give you a basic understanding for the respective technology, but in order to use one of the described technologies in practice you will probably still have to read a dedicated tutorial.

At the beginning of each of the three technology projections, we present a *pattern map*, illustrating the relationships between the Remoting Patterns and the architecture of the respective technology. To give the reader a guide where we are in this overall architecture, we display little boxes as zoomed-out versions of the pattern map at the column. The box here on the left shows an example that denotes that “we are in the server application”- taken from the .NET technology projection.



15 .NET Remoting Technology Projection

.NET provides an easily usable and powerful distributed object middleware which follows very closely the patterns described in the first part of this book. It is probably one of the more powerful and flexible distributed object middleware available today. For the examples, we will use the C# programming language exclusively, although we could also use other .NET languages such as VB.NET or C++.NET.

Note that we cannot go into every detail of .NET Remoting in this chapter. You can find more information for example in [Bar02], [Ram02], and [Sta03].

This chapter is structured as follows. First we give a brief history of .NET Remoting. Second, some core concepts of .NET and .NET Remoting are explained, and we provide a pattern map which we will use throughout the chapter to show “where we are” when explaining concepts. We will then show a simple example, a kind of “Hello Remoting”. Then, we explain some concepts about .NET Remoting boundaries, application domains, and contexts. Then follows a discussion of some first internal workings of .NET Remoting - mainly explaining how the basic remoting patterns are realized. A discussion of error handling follows. The different activation strategies and their implementation in .NET Remoting make up the next sections. Some more advanced lifecycle management techniques, and ideas how they can be emulated in .NET, follow. The next section looks at some internals of .NET, specifically, request handlers as well as proxies. How to extend .NET Remoting is the focus of the subsequent section. Finally, we take a look at asynchronous communications in .NET.

Brief History of .NET Remoting

.NET Remoting has been introduced as part of Microsoft’s .NET platform. From the developer’s perspective, .NET replaces the “old” Windows programming APIs such as the Windows 32 bit API (Win32

API), the Microsoft Foundation Classes (MFC) or, with regards to remoting, DCOM. Still, DCOM will live on as part of COM+ [Mic02]. We don't want to compare DCOM and .NET Remoting here – except for saying that they have almost nothing in common, and developing with .NET remoting is much simpler and more straightforward than DCOM [Gri97].

Note that technically, .NET does not replace the “old” APIs, but it is built on top of them. This, however, is irrelevant for developers.

.NET Remoting is an infrastructure for distributed *objects*. It is not ideally suited to build service-oriented systems. Microsoft will release a new product (currently named Indigo) that provides a new programming model for Web Service based, service oriented systems on top of .NET. It will be possible to migrate .NET Remoting applications as well as applications using .NET Enterprise services to Indigo. At the time of this writing, more information on Indigo can be found at [Mic04b]; we also summarize some key concepts in the later section *Outlook to the Next Generation*.

.NET Concepts - A Brief Introduction

The goal of the technology projections is not to serve as a complete tutorial for the respective technologies. The projections are written in a way that people who do not understand the underlying platform can nevertheless see how the patterns are implemented. The technology projections do assume some previous knowledge: the CORBA projection assumes knowledge of C++, the Web Services projections assumes knowledge of Java. In the same way, the .NET technology projection expects people to understand the basics of .NET, specifically C#.

However, since .NET features a couple of important concepts - and also introduces a couple of new ones - this section briefly looks at some basics that should be roughly understood before reading on.

Just as Java, .NET is based on a virtual machine architecture. The virtual machine is called *Common Language Runtime* or CLR (we also frequently use the term *runtime*). The runtime runs programs written in “.NET assembler”, the *Microsoft Intermediate Language*, or MSIL. Many source languages can be compiled into MSIL, including C#, C++, VB.NET, Eiffel, and many more. Since everything is ultimately repre-

sented as MSIL, a great deal of language interoperability is possible. For example, you can let a C++ class inherit from a C# class.

To avoid name clashes, .NET supports namespaces. A namespace, just as in C++, is logically a prefix to a name. Several classes can be in the same namespace. In contrast to Java, there is no relationship between namespaces and the file system location of a class. Completely independent of namespaces are assemblies; assemblies are the packaging format, a kind of archive, for a number of .NET artifacts, such as types, resources, and metadata (see below). The elements of a namespace can be scattered over several assemblies, and an assembly can contain elements from any number of namespaces. So, namespaces are a means to logically structure the names, whereas assemblies are used to combine things that should be deployed together. It is of course good practice to establish some kind of relationship between namespaces and assemblies to avoid confusion, for instance: put all the elements of one namespace into the same assembly.

.NET provides many features that are known from scripting or interpreted languages. For example, it provides reflection. It is possible to query an assembly for its contained artifacts or to introspect a type to find out its attributes, operations, superclasses, etc. It is also possible to on-the-fly create .NET types and assemblies. *CodeDOM* and the *Reflection.Emit* namespace provides facilities to define types as well as their complete implementation.

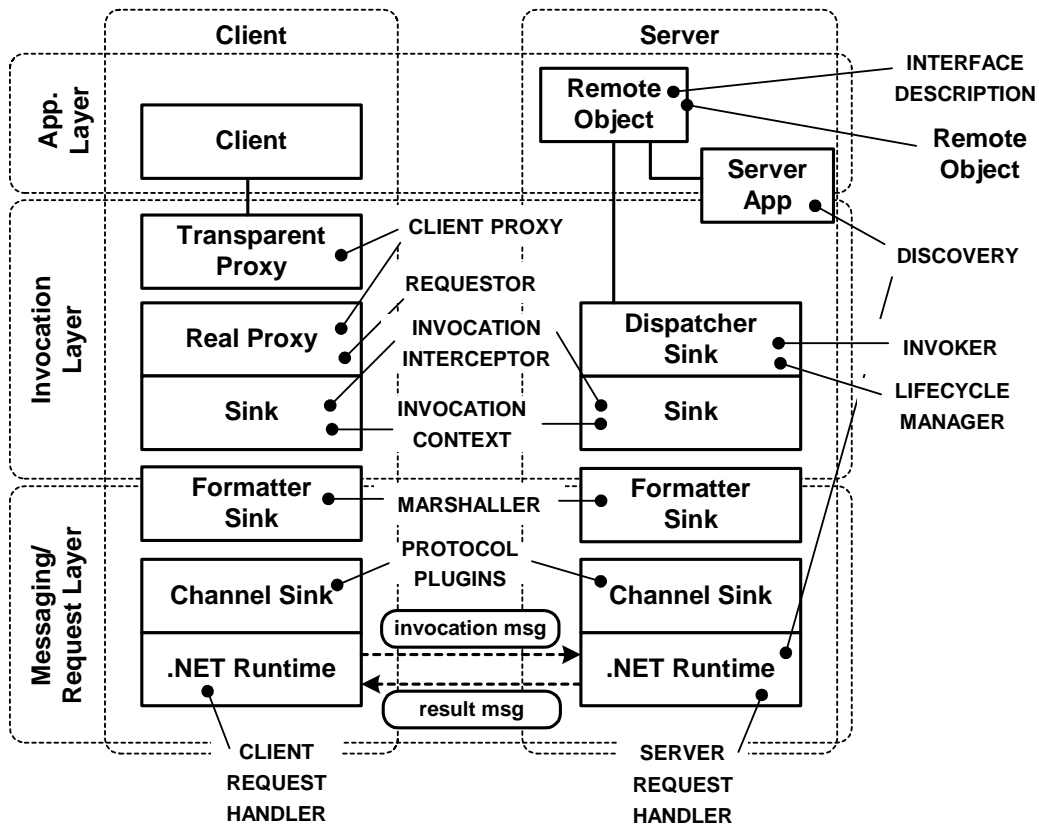
Another very interesting feature is that of attributes. Many .NET elements, such as types, member definitions, operations, etc., can be annotated with attributes in the source code. For example the `[Serializable]` attribute specifies that the annotated type should be marshalled by-value in case of a remote invocation. Developers can define their own attributes which are .NET types themselves; the compiler instantiates them and serializes them into the assembly, together with the compiled MSIL code. At runtime, it is possible to use reflection to find out about the attributes of a .NET element and react accordingly.

Finally, it is worth mentioning that in addition to processes and threads there are additional concepts in .NET. While threads only define a separate, concurrent execution path, processes define a protection domain in addition. If one process crashes, other processes stay unaf-

affected. As a consequence, communicating between processes involves a significant overhead due to context switching. .NET Application Domains provide the context of a protection domain independent of the separate, concurrent execution path without the context switching overhead.

Pattern Map

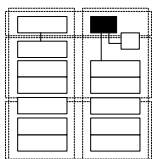
The following illustration shows the basic structure of the .NET Remoting framework. It also contains annotations of pattern names showing which component is responsible for realizing which pattern. Of course this overview does show the behavioral patterns (*Lifecycle Patterns* and *Client Asynchrony Patterns*).



A zoomed-out version of this diagram will serve as an orientation map during the course of this chapter. It highlights the area of the above diagram which a particular paragraph or section relates to.

An Simple .NET Remoting Example

In order to introduce .NET Remoting, we will start out with a simple example. Instead of using the stereotypical Hello World, we use a different (but comparably simple) example, in which we build a distributed system in the medical domain. Our introductory example will comprise remotely accessible *PatientManager* object that provides access to the patients in a hospital's IT system.

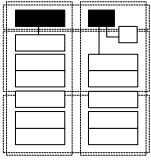


Let us look at the remote object first. In .NET, a remotely accessible object must extend the *MarshalByRefObject* class as shown in the following example. *MarshalByRefObject* is thus the base class for all remotely accessible objects in .NET. The name is an allusion to the fact that the object will not be marshalled (i.e. serialized) to a remote machine. Instead only the remote object's reference will be marshalled to the caller in case an instance is passed as an operation parameter or as a return value.

```
using System;
namespace PatientManagementServer
{
    public class PatientManager: MarshalByRefObject
    {
        public PatientManager()
        {}
        public Patient getPatientInfo( String id )
        {
            return new Patient( id );
        }
    }
}
```

Here we define the *PatientManager* class which provides a single operation to retrieve a *Patient* object. The class resides in the *PatientManagementServer* namespace.

.NET does not require a separate interface for remote objects - by default, the public operations of the implementation class are available remotely. But to make sure we do not need the remote object class definition (*PatientManager*) in the client process, we provide an



interface that defines the publicly available operations. According to .NET naming conventions, this interface is called `IPatientManager`. It is located in a namespace `PatientManagementShared` which we will use for the code that is required by client and server. `PatientManagementServer` and `PatientManagementClient` will be used for server-only and client-only code, respectively.

```
namespace PatientManagementShared
{
    public interface IPatientManager
    {
        Patient getPatientInfo( String patientID );
    }
}
```

The implementation must now of course implement this interface to ensure that the `PatientManager` is-a `IPatientManager`. This is necessary because the client will use the `IPatientManager` interface to declare references to remote `PatientManager` objects.

```
namespace PatientManagementServer
{
    public class PatientManager: MarshalByRefObject, IPatientManager
    {
        public PatientManager()
        {}
        public Patient getPatientInfo( String id )
        {
            return new Patient( id );
        }
    }
}
```

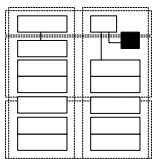
The operation `getPatientInfo` returns an instance of `Patient`. This class is a typical example of a *Data Transfer Object* [Fow03, Mic04c, Sun04a], which is serialized “by value” when transported between server and client (and vice versa). In .NET such objects need to contain the `[Serializable]` attribute in their class definition, as shown below. Note that, just as the interface, the `Patient` class is also defined in the `PatientManagementShared` assembly because it is used by client and server.

```
namespace PatientManagementShared
{
    [Serializable]
    public class Patient
    {
        private String id = null;
    }
}
```

```

    public Patient( String _id )
    {
        id = _id;
    }
    public String getID()
    {
        return id;
    }
}

```



To continue, we need to get a suitable server application running for our initial example. It needs to set up the remoting framework and publish the remote object(s). We postpone the details of setting up the Remoting framework, and thus just use a “black box” helper class¹ to do this for us, `RemotingSupport`. With this, the server application becomes extremely simple:

```

using System;
using RemotingHelper;
using System.Runtime.Remoting;
using PatientManagementShared;

namespace PatientManagementServer
{
    class Server
    {
        static void Main(string[] args)
        {
            RemotingSupport.setupServer();
            RemotingConfiguration.RegisterWellKnownServiceType(
                typeof(PatientManager), "PatientManager",
                WellKnownObjectMode.Singleton );
            RemotingSupport.waitForKeyPress();
        }
    }
}

```

The second line of the main method is the interesting part: Here we publish the `PatientManager` remote object for access from clients.

This also includes a definition of a name under which the remote object will be available to clients. The following URL will serve the purpose of a ABSOLUTE OBJECT REFERENCE:

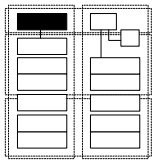
```
http://<the serverHost>:<the port>/PatientManager
```

-
1. This class is not part of the .NET Remoting framework, we have implemented it for the sake of this example. The class itself consists just of a couple of lines, so it does not hide huge amounts of complex code.

Note that there is no central LOOKUP system in .NET, clients have to know the host on which an object can be looked up by its name. They have to use the URL above. Note that the URL contains information about the transport protocol used to access the object. Since .NET supports different communication *channels* (see below), it is possible, after appropriate configuration, to reach the same object using a different URL. An example using the TCP channel is:

```
tcp://<the serverHost>:<another port>/PatientManager
```

Note also that we use LAZY ACQUISITION here. The `WellKnownObjectMode.Singleton` specifies that there is only one shared instance remotely accessible at any time. This instance is created only when the first request for the `PatientManager` remote object arrives. For details of the activation mode see below.

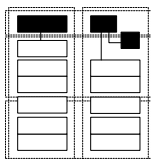


Last but not least, we need a client. It is also relatively straightforward:

```
namespace PatientManagementClient
{
    class Client
    {
        static void Main(string[] args)
        {
            RemotingSupport.setupClient();
            IPatientManager patientManager =
                (IPatientManager)Activator.
                GetObject(typeof(IPatientManager),
                "http://localhost:6642/PatientManager" );
            Patient patient = patientManager.getPatientInfo("42");
            Console.WriteLine("ID of the "+
                "retrieved Patient:"+patient.getID());
        }
    }
}
```

First we invoke the `setupClient` operation on the `RemotingSupport` class. Then we look up the remote object using the URL scheme mentioned above. We use the `Activator` class provided by the .NET framework as a generic *factory* for all kinds of remote objects. The port 6642 is defined as part of the `RemotingSupport.setupServer` operation called by the `Server's Main` operation. Note that we use the interface to declare and downcast the returned object, not the implementation class. We then finally retrieve a `Patient` from the remote `PatientManager` and, for the sake of the example, ask it for its ID.

Setting up the Framework



As mentioned, we use the `RemotingSupport` helper class to set up the Remoting framework. Internally, this class uses a .NET API to do the real work. Alternatively, it is also possible to set up Remoting using certain elements in the application configuration XML file. Each .NET application can have an associated configuration file that controls various aspects of the application's use of the .NET framework. The file must be loaded by the program manually. Examples include the versions of assemblies that should be used, security information as well as setup information for Remoting. Although we will show a snippet of the XML file later in this chapter, we will not look at it in very much detail in this book.

Assemblies for the Simple Example

Assemblies in .NET are binary files that are used for deployment. An assembly is represented by a DLL or an executable. Note that assemblies have no direct relationship to namespaces. A namespace can be scattered over several assemblies, and a single assembly can contain several namespaces. It is, however, practical in most cases to use only one assembly per namespace. This is what we will do here. We use the following four assemblies in this example:

- `PMShared` contains the things that are needed by client and server.
- `RemotingHelper` contains stuff needed to set up the .NET Remoting framework.
- `PMServer` contains all the code for the server.
- `PMClient` contains all the code for the client.

are detected. As a consequence, application domains provide the same isolation features as processes but with the additional benefit that communication between application domains need not cross process boundaries. This can significantly improve performance and scalability since developers can use application domains instead of processes if they need protection, paying a smaller penalty in communication performance.

However, in order to communicate between two application domains, .NET Remoting has to be used. Since two processes automatically constitute two application domains, .NET Remoting has to be used whenever application domain boundaries have to be crossed.

Note that application domains are not related to threads. An application domain can contain several threads, and a thread can cross application domains over time.

There is a distinct relationship between assemblies and application domains, though. A specific assembly is always executed in one application domain. For instance, you can run the same assembly in several application domains in which case the code is shared, but the data is not.

Application domains can be set up manually by using the `AppDomain.CreateDomain()` operation. It takes a set of parameters such as the domain's working directory, the configuration file for the domain, as well as the search path the runtime uses to find assemblies. Once an application domain is set up, you can use the `Load()` operation to load a specific assembly into this domain. `Unload()` allows you to unload the assembly (without requiring to stop the process). Finally, you can create instances of a `Type` in a certain application domain using the `CreateInstanceFrom()` operation.

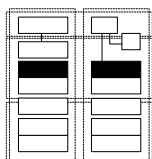
In the remainder of this chapter, we will, however, look at Remoting from the perspective of crossing process boundaries.

Contexts

Contexts provide a way to add `INVOCATION INTERCEPTORS` to objects running in the same application domain. Again, .NET Remoting is used to insert proxies that handle interception; thus, context can be

seen as a form of CONFIGURATION GROUPS. Details are explained later in the section on INVOCATION INTERCEPTORS or in [Low03].

Basic Internals



In contrast to other platforms, such as natively compiled C++, the .NET platform is a relatively dynamic environment. This means that reflective information is available, instances can be asked for their types, and types can also be modified or even created at runtime. As a consequence, many classes for which you would have to generate and compile source code manually on a native platform, can be generated on the fly by the .NET runtime. There is no separate source code generation or compilation step required.

An example for such an automatically generated class is the CLIENT PROXY. You never see any source code for these classes. There is also no code-generated server-side skeleton as part of the INVOKER. The INVOKER - called *dispatcher* in .NET - is a .NET framework component that uses reflection as well as the information passed in remote method invocation requests to dynamically invoke the target operation on the remote object.

As usual, the communication framework is an implementation of the BROKER pattern [BMR+96]. This, as well as the CLIENT REQUEST HANDLER and SERVER REQUEST HANDLER, form an integral part of the .NET framework and are also supported by the .NET runtime itself.

Each remote object instance has a unique OBJECT ID. Using Visual Studio.NET's debugger, we can look into the state of a remote object instance and see the unique OBJECT ID, called `_ObjURL` here:

Name	Value	Type
this	{RemotingServer.PatientManager}	RemotingServer.PatientManager
System.MarshalByRefObject	{RemotingServer.PatientManager}	System.MarshalByRefObject
System.Object	{RemotingServer.PatientManager}	System.Object
identity	{System.Runtime.Remoting.ServerIdentity}	System.Runtime.Remoting.ServerIdentity
[System.Runtime.Remoting.ServerIdentity]	{System.Runtime.Remoting.ServerIdentity}	System.Runtime.Remoting.ServerIdentity
System.Runtime.Remoting.Identity	{System.Runtime.Remoting.ServerIdentity}	System.Runtime.Remoting.ServerIdentity
System.Object	{System.Runtime.Remoting.ServerIdentity}	System.Runtime.Remoting.ServerIdentity
_ObjURL	"e478bad4_a6c0_43a1_ae5e_4f7f9bd2c644/PatientManager"	string
_URL	null	string

Another attribute of the remote object instance currently under introspection is the ABSOLUTE OBJECT REFERENCE. This reference contains

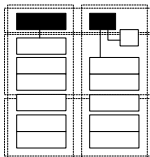
several attributes (highlighted below): among others, it contains the OBJECT ID and the communication information, here, the *tcp* URL. Note

Name	Value	T
- objRef	{System.Runtime.Remoting.ObjRef}	S
- [System.Runtime.Remoting.ObjRef]	{System.Runtime.Remoting.ObjRef}	S
- System.Object	{System.Runtime.Remoting.ObjRef}	S
- FLG_MARSHALED_OBJECT	1	in
- FLG_WELLKNOWN_OBJREF	2	in
- FLG_LITE_OBJREF	4	in
- uri	"/e478bad4_a6c0_43a1_ae5e_4f7f9bd2c644/PatientManager"	st
+ typeInfo	{System.Runtime.Remoting.TypeInfo}	S
- envoyInfo	null	S
- channelInfo	{System.Runtime.Remoting.ChannelInfo}	S
+ [System.Runtime.Remoting.ChannelInfo]	{System.Runtime.Remoting.ChannelInfo}	S
+ ChannelData	{Length=2}	S
- objrefFlags	0	in
+ orType	{System.RuntimeType}	S
- URI	"/e478bad4_a6c0_43a1_ae5e_4f7f9bd2c644/PatientManager"	st
+ TypeInfo	{System.Runtime.Remoting.TypeInfo}	S
- EnvoyInfo	null	S
- ChannelInfo	{System.Runtime.Remoting.ChannelInfo}	S
+ [System.Runtime.Remoting.ChannelInfo]	{System.Runtime.Remoting.ChannelInfo}	S
- ChannelData	{Length=2}	S
- [0]	{System.Runtime.Remoting.Channels.ChannelDataStore}	S
- System.Object	{System.Runtime.Remoting.Channels.ChannelDataStore}	S
- _channelURLs	{Length=1}	st
- [0]	"tcp://172.20.2.13:6642"	st
- _extraData	null	S
+ ChannelUri	{Length=1}	st
- Item	<cannot view indexed property>	S
+ [1]	{System.Runtime.Remoting.Channels.CrossAppDomainData}	S

the similarity between the information here and the URL we introduced before that allows clients to access remote objects: the URL contains the same information as is displayed here.

As we shall see later, a remote object can be accessed remotely through several different *channels*. Channels are communication paths that consist of a protocol and a serialization format, as well as some other things (see below). A channel is connected to a network endpoint with a specific configuration. In case of TCP this would be the IP address and the port, here 6642. Channels have to be configured when the server application is started.

Error Handling



REMTING ERRORS are reported using subclasses of `System.SystemException`. For example, in case the CLIENT REQUEST HANDLER is not able to contact the server application because of network problems, it throws a `WebException` or a `SocketException` (both extending

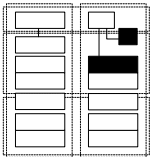
SystemException) to the client, depending on whether a TCP or an HTTP channel is used to access the remote object. To clearly distinguish application-specific exceptions from REMOTING ERRORS, a convention says that application exceptions must not subclass SystemException, instead it is recommended that you use System.ApplicationException as a base class. An example of a user-defined exception follows:

```
using System;
using System.Runtime.Serialization;

namespace PatientManagementShared
{
    [Serializable]
    public class InvalidPatientID: ApplicationException
    {
        public InvalidPatientID( String _message ) : base(_message)
        {
        }
        public InvalidPatientID(SerializationInfo info,
                                StreamingContext context):
            base(info, context)
        {}
    }
}
```

Note that the class has to provide a so-called deserialization constructor (the one with the `SerializationInfo` and `StreamingContext` parameters) and it has to be marked `[Serializable]` otherwise the marshalling will not work. The exception's body is empty, because we have no additional parameters compared to `Exception`.

Server-Activated Instances



.NET provides two fundamentally different options with regards to activation of remote objects:

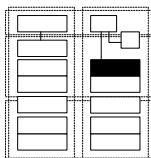
- Remote objects can be activated by the server. In this case, a client just contacts a remote object and does not care for its creation and destruction.
- Remote objects can also be explicitly created and destroyed by a client. The lifecycle of these CLIENT-DEPENDENT INSTANCES is thus controlled by a specific client and not by the server.

This section looks at alternatives for server-side activation; the next section looks at CLIENT-DEPENDENT INSTANCES more extensively.

.NET Remoting provides the following activation options for server activated objects. We will look at each of those in detail in the following sections:

- There are PER-REQUEST INSTANCES: a new instance will be created for each and every method invocation.
- STATIC INSTANCES can also be used; they have singleton semantics, which means there is always at most one instance in each .NET runtime.
- LAZY ACQUISITION is similar to STATIC INSTANCES, but the instances will be initialized on demand, whereas STATIC INSTANCES are instantiated manually, typically when the server application starts up.

Per-Request Instances



As described in the pattern text, using PER-REQUEST INSTANCES requires the remote objects to be stateless, because a new instance is created upon each invocation. Note that you as the developer have to be aware of this restriction and make sure your remote objects are actually stateless - there is nothing in .NET that prevents developers from making PER-REQUEST INSTANCES stateful - possibly resulting in unexpected program behavior.

To implement a PER-REQUEST INSTANCE, all you have to do is to specify the `SingleCall` activation mode when you register the remote object with the Remoting framework. The code to do so in the server application is shown in the following piece of code:

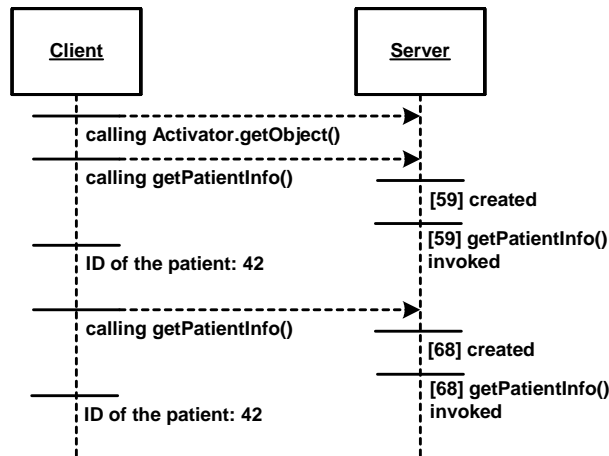
```
namespace PatientManagementServer
{
    class Server
    {
        static void Main(string[] args)
        {
            RemotingSupport.SetupServer();
            RemotingConfiguration.RegisterWellKnownServiceType(
                typeof(PatientManager), "PatientManager",
                WellKnownObjectMode.SingleCall );
            RemotingSupport.WaitForKeyPress();
        }
    }
}
```

```

    }

```

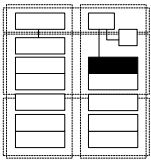
To illustrate the effect of using PER-REQUEST INSTANCES, we have added some text output messages to the `PatientManager` remote object and the client. The next diagram shows the output from client and server, using a sequence diagram-like graphical syntax.



Two things can be observed: First of all, an instance is really only created when an operation is invoked, not when the object is acquired by the client calling `Activator.GetObject` or when the object is registered with the remoting framework. And second, for each invocation a new instance is actually created, no POOLING is used here. This can be observed from the numbers printed as part of the server's output, they are a short form of the current OBJECT ID.

As each request has its own instance, you don't need to care about thread-safety - as long as you don't access shared resources.

Lazy Acquisition

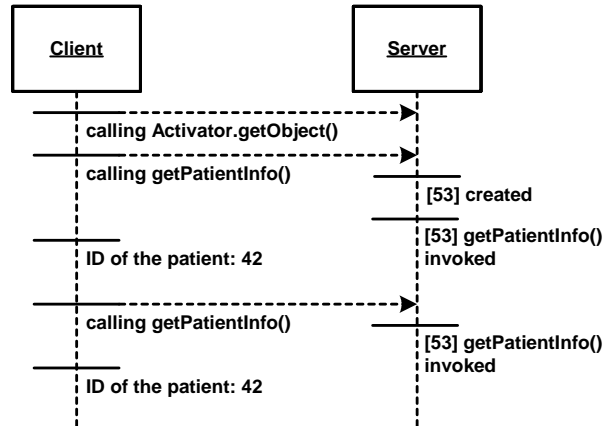


LAZY ACQUISITION activates an instance when a request arrives, and keeps it alive until its lease time expires (more on LEASING below). When several requests arrive for the same remote object, all are handled by the same instance (per CLR instance; this is why we say it has *Singleton* semantics). The .NET Remoting framework will potentially handle concurrent requests concurrently in several threads, so the remote object implementation has to be thread safe. Configuring the framework to behave accordingly is easy again, just pass

WellKnownObjectMode.Singleton as the parameter to the RegisterWellKnownServiceType operation.

```
RemotingConfiguration.RegisterWellKnownServiceType(
    typeof(PatientManager), "PatientManager",
    WellKnownObjectMode.Singleton );
```

The output is visualized by the following diagram.

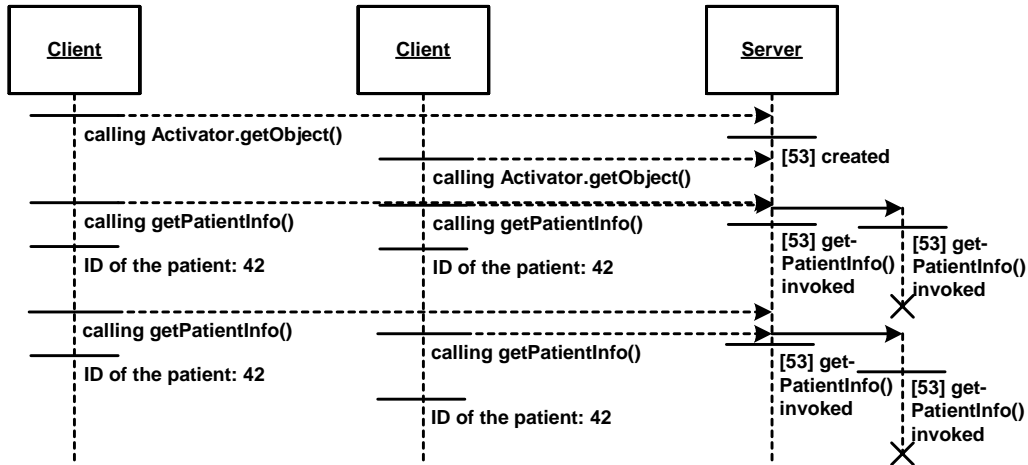


As you can see, only one instance is created on the server, even for multiple subsequent requests. If the lease time expires, the instance is discarded. When a new invocation arrives, a new instance is created that will handle all requests until its own lease expires. If you want to avoid this behavior, you have to make sure the instance lives forever. To do this, you have to return null in InitializeLifetimeService of the particular remote object type. This is typically done in case of STATIC INSTANCES:

```
public class PatientManager: MarshalByRefObject, IPatientManager
{
    // all other operations as before
    public override object InitializeLifetimeService()
    {
        return null;
    }
}
```

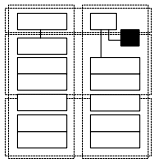
Take a look at the CLIENT-DEPENDENT INSTANCES section below to learn more details about InitializeLifetimeService and leasing.

A more interesting situation occurs, however, when two clients concurrently call the `PatientManager` on the same server application. This situation is depicted in the following diagram.



In this situation, the operation is invoked from within several threads (as illustrated by the two object lifelines of the server), but on the same instance. The constructor is called only once. Here, thread-safe programming is required as soon as shared resources require coordinated access from within the remote object.

Static Instances



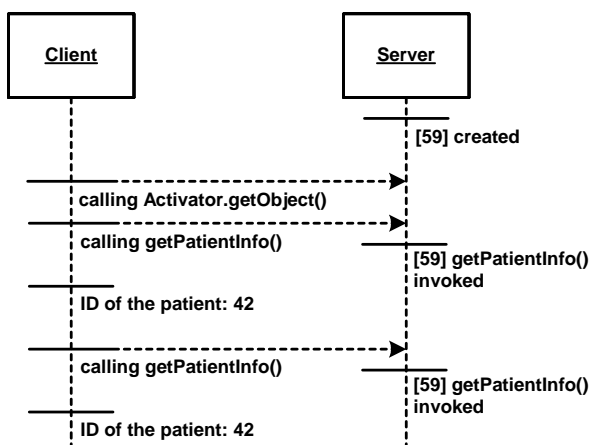
.NET also provides the possibility to use a `STATIC INSTANCE`. Instead of just registering a remote object type with the framework and letting the framework do the instantiation on demand, you instantiate the remote object manually and then publish the instance explicitly.

```

namespace PatientManagementServer
{
    class Server
    {
        static void Main(string[] args)
        {
            RemotingSupport.setupServer();
            PatientManager pm = new PatientManager();
            RemotingServices.Marshal(pm, "PatientManager");
            RemotingSupport.waitForKeyPress();
        }
    }
}
  
```

Here, the object is instantiated manually. After the instantiation in the second line of the Main operation, the remote object is published using the `RemotingServices.Marshal` operation. Note that this approach is a use of the *Eager Acquisition* pattern [KJ04].

Using `STATIC INSTANCES` has a number of additional advantages: You can call arbitrary constructors, whereas the other means always use the Remoting framework's invocation of the remote object's default constructor. Also, you can control if and when instances are created, and clients' access to instances will happen without a potential creation overhead. The following diagram shows the output of running the program.



Note that just as with `LAZY ACQUISITION`, you have to add your own synchronization constructs when accessing shared resources.

You can publish several distinct instances of a particular remote object type with different names:

```

PatientManager pm = new PatientManager();
RemotingServices.Marshal(pm, "PatientManager");
PatientManager pm2 = new PatientManager();
RemotingServices.Marshal(pm2, "PatientManager2");
  
```

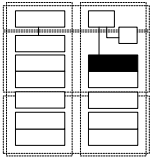
`pm` and `pm2` are distinct instances, accessible through a different name, and both have their own, separate state (in case they are not stateless). Of course you can also publish several different instances when using the other server-based activation techniques, you just have to register the same remote object type with different names – these are then different remote objects from the framework's perspective, both of

them happen to use the same implementation class and interface. The following code shows this for PER-REQUEST INSTANCES:

```
namespace PatientManagementServer
{
    class Server
    {
        static void Main(string[] args)
        {
            RemotingSupport.setupServer();
            RemotingConfiguration.RegisterWellKnownServiceType(
                typeof(PatientManager), "PatientManager",
                WellKnownObjectMode.SingleCall );
            RemotingConfiguration.RegisterWellKnownServiceType(
                typeof(PatientManager), "PatientManager2",
                WellKnownObjectMode.SingleCall );
            RemotingSupport.waitForKeyPress();
        }
    }
}
```

As usual, a client can access any of the remote objects by using the `Activator.GetObject` operation passing in the URL containing the respective object's name.

Lifetime of the Instances



First of all, we have to define what we mean by “lifetime of instances”. There are two different “lifetimes” of remote objects:

- One is the time the programming language class's instance exists, i.e. the time during which the object is managed by the .NET runtime. This is not related to the Remoting framework.
- The other lifetime is the time during which the instance is managed by the Remoting framework, and is remotely accessible.

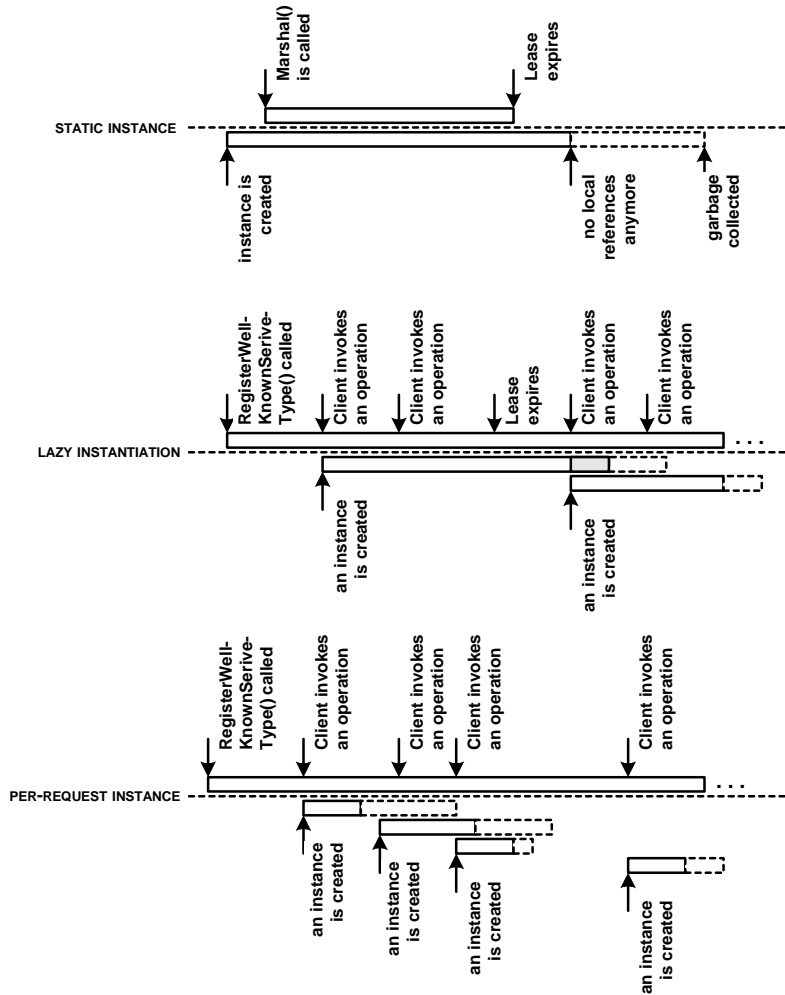
Depending on the activation mode, the object either physically exists first and is then managed by the remoting framework (as in case of STATIC INSTANCES) or the remote object type is managed by the framework first (no instances yet existing), and instances are created on demand (as in case of LAZY ACQUISITION, PER-REQUEST INSTANCES and CLIENT-DEPENDENT ACTIVATION).

In the context of this section, we are mainly interested in the period of time during which an object is managed by the Remoting framework. Let's look at the different activation modes:

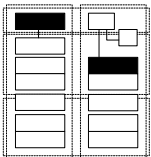
- In case of a PER-REQUEST INSTANCE the situation is rather simple: The instance is created upon request, and then immediately discarded again. Eventually the object is garbage-collected by the .NET runtime.
- In the other cases the object lives until its lease expires. It is then discarded by the Remoting framework, and eventually garbage collected. Usually, you want to make sure the instance lives forever by returning null in `InitializeLifetimeService`. We will look into LEASING in more detail later, when we discuss CLIENT-DEPENDENT INSTANCES.

The following diagram illustrates the discussion. It shows the lifetime of the remote object (above the dotted line) and the lifetime of real

implementation objects (below the dotted line) for each of the activation modes.



Client-Dependent Instances and Leasing



CLIENT-DEPENDENT INSTANCES are necessary if you need to have remote objects that are specific to a certain client and can keep (conversational) state on the client's behalf. There are two specifically interesting aspects of CLIENT-DEPENDENT INSTANCE:

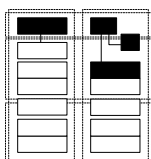
- how they are created and accessed by a client, and
- how long they live.

These two aspects are trivial for the other activation modes, because in all cases a generic factory can be used for activation (.NET's *Activator* class) and the lifetime of the instances is also non-critical:

- PER-REQUEST INSTANCES die just after serving a request
- and singletons are typically made to live forever.

The next sections look at these aspects in the context of CLIENT-DEPENDENT INSTANCES, where especially the determination of when an instance is allowed to die is non-trivial.

Activation and Bootstrapping



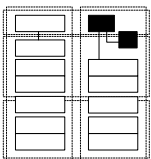
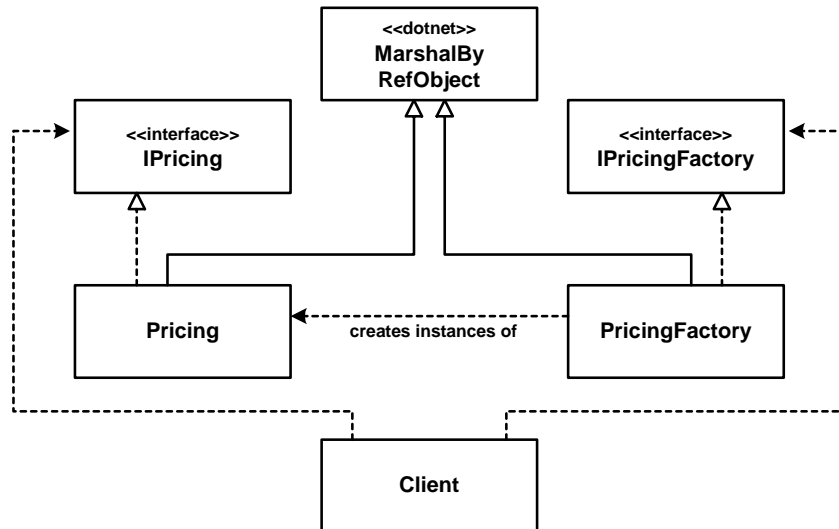
In .NET there are several ways how a client can obtain a reference to a CLIENT-DEPENDENT INSTANCE:

- One is to call `Activator.GetObject`. The *Activator* class is a generic factory provided as part of the .NET framework. It can be used to access any kind of remote object. This also works for CLIENT-DEPENDENT INSTANCES, but allows only to instantiate remote objects using their default constructor. This is impractical, since a client typically wants to initialize “his personal instance” by passing parameters to a constructor.
- Another alternative is to use the `new` operator that seamlessly, after correct configuration of the framework, instantiates a remote object instance *on the server*, although called in a client application. However, for this to work, you need the remote object’s implementation class on the client as opposed to just an interface, as the compiler will not allow you to call the `new` operator on an interface type. This is a bad choice, since clients should never depend on remote object implementation classes.
- A third option is to use a *factory* [GHJV95] to instantiate these remote objects on the server application. In this approach, the factory is a STATIC INSTANCE or PER-REQUEST INSTANCE and thus needs no additional bootstrapping. The factory can call whatever constructor is useful, the parameters can be passed as parameters to the factory operation. The factory operation’s return type is the

an interface type that is implemented by the remote object implementation class, and defines its remotely available operations.

Since we consider the third option the most versatile and powerful, let us look at it a bit more closely. The CLIENT-DEPENDENT INSTANCE we want to create is an object that allows us to calculate the total amount of money that will be billed to the health insurance company for a patient's stay in a hospital. Because such calculations can become rather complex (at least in Germany, as a result of the intricacies of the german health system), it is not possible to do this within one stateless remote operation call. We need CLIENT-DEPENDENT INSTANCES, one for each pending calculation. So we introduce a new example, the Pricing remote object, that will be stateful and client-activated. The interface of the Pricing remote object, is called IPricing.

As mentioned before, we want to use a factory to allow the client to create new Pricing instances. This factory is called PricingFactory. We also introduce the interface (IPricingFactory) that will be used by the client to access the Pricing remote object. The following UML diagram shows the setup.



We'll start with the interface of the factory:

```

namespace PatientManagementShared
{
    public interface IPricingFactory
    {

```



```

        IPricing getPricingInstance( Patient patient );
    }
}

```

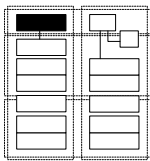
The factory operation returns a new instance of a class that implements the `IPricing` interface, such as a `Pricing`. The instance is initialized with the `Patient` that has been passed to the factory. The implementation for the factory interface is shown next; it is rather simple:

```

namespace PatientManagementServer
{
    public class PricingFactory : MarshalByRefObject,
                                IPricingFactory
    {
        public IPricing getPricingInstance( Patient patient )
        {
            return new Pricing( patient );
        }
    }
}

```

The implementation simply returns a new instance of the `Pricing` remote object. The `PricingFactory` will be registered as a LAZILY INSTANTIATED object, just as described above. This approach permits the client to only work with the interface (`IPricing`) and we can call a non-default constructor (the one with the `Patient` parameter). The only drawback is the additional “complexity” for client and server to handle the *factory* object, and the development of the *factory* class by the developer.



The only additional code required on the client is the factory’s interface. So how does the code look from the client’s perspective? The following piece of code shows an example.

```

// retrieve a Patient object from a PatientManager
RemotingSupport.setupClient();
IPatientManager patientManager =
    (IPatientManager)Activator.GetObject(
        typeof( IPatientManager ),
        "tcp://localhost:6642/PatientManager" );
Patient patient = patientManager.getPatientInfo("42");

// retrieve the PricingFactory
IPricingFactory pf = (IPricingFactory)Activator.GetObject(
    typeof( IPricingFactory ),
    "tcp://localhost:6642/PricingFactory" );

// get an instance of a pricing object and work with it
IPricing p = pf.getPricingInstance(patient);

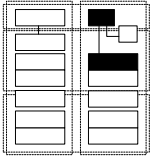
```

```
p.addDiagnosticKey( new Key("09098") );
p.addDiagnosticKey( new Key("23473") );
double total = p. getTotalPrice();
```

In the code above you can see how the client retrieves a *Patient* object in the first section using the *Activator*. Section two retrieves the factory using the same approach (because it is also stateless) and the third section uses the factory to obtain an instance of *IPricing*.

Until here there is no difference comparing *CLIENT-DEPENDENT INSTANCES* and the other activation modes. It gets interesting, however, when we consider how long an instance actually lives before it is considered “invalid” by the framework. The following discussion is also important for *LAZY ACQUISITION* and *STATIC INSTANCES*, because they might also fade once their lease expires. This is typically not what is intended, so they have to be made to live forever.

Implementing Lease Management

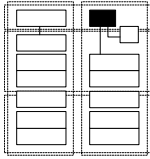


.NET remoting uses the *LEASING* pattern. A lease is basically a temporarily limited permission for a remote object instance to exist, and to be accessed by clients. Each instance has a counter (called *Time To Live*, or *TTL*) that determines how long the instance is allowed to live. .NET uses the following policies to determine the lifetime of an instance:

- An instance has an *InitialLeaseTime* that determines how long the instance lives initially after creation.
- Upon each method invocation, the remaining *TTL* is set to the specified *RenewalOnCallTime*.
- When the instance's *TTL* reaches zero (i.e. when no method has been invoked for the remaining *TTL*) then a so-called *sponsor* will be contacted by the *LEASING* manager, if one is registered for the respective instance. The sponsor can increase the *TTL* to make sure the instance does not die although the lease had expired.
- If the sponsor does not increase the *TTL* or in case no sponsor is registered for the instance, the instance is discarded by the Remoting framework and eventually garbage-collected.

The default values for the initial lease time is five minutes, the renewal on call time defaults to two minutes. For many application scenarios this is not suitable, and shorter or longer timeouts might be appropriate. It is easily possible to change these settings – either on an

application-wide level, on a remote object type level, or per remote object instance. Let's look at changing the LEASING settings for a particular remote object type.



The Pricing implementation could look something like the following:

```
namespace PatientManagementServer
{
    public class Pricing : MarshalByRefObject, IPricing
    {
        private Patient patient = null;
        private ArrayList keys = new ArrayList();

        public Pricing( Patient _patient ) {
            patient = _patient;
        }

        public void addKey( DiagonsticKey key ) {
            keys.Add( key );
        }

        public double getTotalPrice() {
            return keys.Count;
        }

        public Patient getPatient() {
            return patient;
        }
    }
}
```

This implementation uses the system-wide defaults. To change these for this particular remote object type, we have to override the InitializeLifetimeService operation in the following way:

```
namespace PatientManagementServer
{
    public class Pricing : MarshalByRefObject, IPricing
    {
        // the other operations as before

        public override object InitializeLifetimeService()
        {
            ILease lease = (ILease)base.InitializeLifetimeService();
            if ( lease.CurrentState == LeaseState.Initial )
            {
                lease.InitialLeaseTime = TimeSpan.FromSeconds(10*60);
                lease.RenewOnCallTime = TimeSpan.FromSeconds(5*60);
            }
            return lease;
        }
    }
}
```

```
}
```

As you can see, we set the initial lease time to 10 minutes and the renewal interval at each method call to 5 minutes.

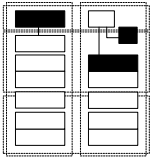
If you want to make sure your instance lives forever (i.e. until program termination) then you just return null in `InitializeLifetimeService`. This is typically done in case of STATIC INSTANCES:

```
public class PreconfiguredEx : MarshalByRefObject, IPricing
{
    public override object InitializeLifetimeService()
    {
        return null;
    }
}
```

When a client accesses an instance that is no longer accessible because its lease has timed out, the client receives a `RemotingException`:

```
Unhandled Exception: System.Runtime.Remoting.RemotingException:
Object </6ddffc7/1.rem> has been disconnected or does not exist at
the server.
```

Last Hope: Sponsors



We mentioned the concept of a *sponsor* before. Sponsors are the “last hope” for a remote object in case its lease expires. The lease manager contacts the sponsor and asks it whether it should renew the lease for the ready-to-die instance. Sponsors must implement the `ISponsor` interface which is very simple:

```
public interface ISponsor {
    TimeSpan Renewal( ILease lease );
}
```

The following is a simple example of a sponsor that allows its associated instance an extended lifetime of 20 minutes:

```
namespace RemotingTestApp
{
    public class Sponsor : ISponsor
    {
        public TimeSpan Renewal( ILease lease )
        {
            return TimeSpan.FromSeconds(20);
        }
    }
}
```

The respective instance would have to die immediately if the sponsor returns `TimeSpan.Zero`.

Sponsors can be remote objects themselves. This means that the client of a CLIENT-DEPENDENT object can host a sponsor object that determines the lifetime of the associated CLIENT-DEPENDENT object.

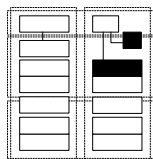
In addition, the lease manager on the server application will only try to contact the sponsors (there can be several of them per remote object) for a specific period of time, which can be defined in the `InitializeLifetimeService` operation. After not reaching the sponsor(s) for the specified amount of time, the lease manager gives up and the instance is finally discarded.

Remote sponsors and the sponsor timeout together provide a flexible and practical scheme for distributed garbage collection. Two examples:

- If all remote sponsors for CLIENT-DEPENDENT INSTANCES are located in the respective client processes, then, should this client go away (e.g. terminate without logging off, or just crash) the client-side sponsor will go away with it and no new lease renewals will be sponsored. The CLIENT-DEPENDENT instance on the server application will be discarded and eventually garbage-collected.
- Alternatively, you can deploy a sponsor on the server application, make it remotely accessible, and use keep-alive pings from the client to determine its own lifetime – thus determining the time period during which it is available for the lease manager.

Depending on the implemented strategy, there might be a performance penalty because of messages being sent between the lease manager and the sponsor.

More Advanced Lifecycle Management

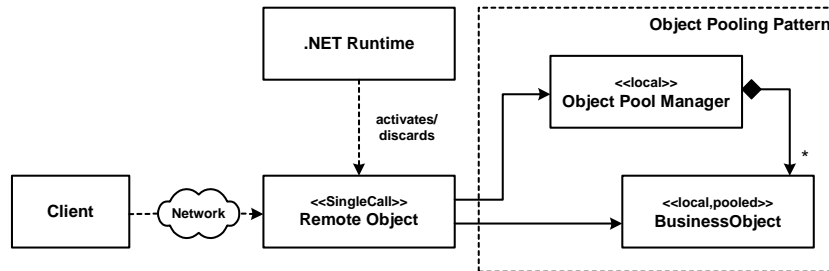


More advanced lifecycle management options are not directly available as of version 1.x of the .NET framework. Specifically, POOLING of remote objects is not available. In the case of the `SingleCall` activation strategy a new instance is created for each invocation. After the invocation, it is discarded and eventually garbage-collected. This strategy assumes that garbage collection does not impose an unacceptable performance penalty. Of course it is possible to deploy your

remote objects as COM+ [Mic02] components where POOLING is supported. Visual Studio provides tool-support for this and other forms of deployment.

In case of the *Singleton* activation strategy, the instance lives until its lease expires (and no sponsor votes for keeping the instance alive). There is no passivation of state if an instance is not accessed for a while.

In cases where the optimizations such as POOLING are necessary, they have to be built manually. How can this be achieved? Let us take a closer look at POOLING. This strategy is obviously only necessary, if the objects that are pooled are either expensive to create or to destroy or if the resources they represent are limited. The following diagram shows the basic constituents of a self-made pooling infrastructure.



In this example, we use a *SingleCall* remote object which is remotely accessible and provides the interface to the client. Its lifecycle is managed by the .NET runtime. Nothing special so far. This remote object, however, does not do very much though, it merely delegates to another object (we called it *business object* here) which does the real work. The remote object acquires an instance of the business object through an object pool manager. So the resources that should be pooled are kept by the business object which itself is pooled. The remote object, with its framework-managed *SingleCall* lifecycle, merely acts as a kind of proxy delegating to the pooled business object. Of course you have to develop your own synchronization and lifecycle code. This is non-trivial and it would be nice if the .NET framework did this for us.

For *PASSIVATION* the same discussion applies. .NET Remoting does not support it directly, however, COM+ provides support for this feature. Again, a *PASSIVATION* infrastructure can be implemented along the same lines. It can even be seen as an extension of the *POOLING* infrastructure we just looked at. In this case, the pool manager would also

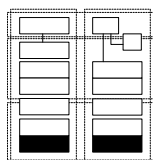
be responsible for passivating/activating objects as they go into/from the pool by persisting the objects' state.

Note that in effect we are building our own LIFECYCLE MANAGER here.

Internals of .NET Remoting

Many aspects of the framework we describe with our patterns in part one of the book are completely invisible to the .NET Remoting developer. This is because these features are supported by the .NET runtime and thus provided to the developer transparently. We discuss these internals in this section.

Request Handlers



CLIENT REQUEST HANDLER and SERVER REQUEST HANDLER are provided by the .NET runtime. Every CLR instance is capable of receiving remote invocations from other runtimes. However, this does not imply that runtimes that currently do not host any remote objects consumes resources for the provisioning of the remoting infrastructure. The necessary infrastructure is only started up once remote objects are actually hosted by a CLR instance. Also, network resources (such as sockets) are only acquired when they are really needed.

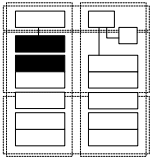
The integration of remoting as an integral part of the .NET framework and runtime has its advantages as well as liabilities. On the plus side, the developer does not have to care about the details of how remoting is technically handled - it is part of the platform. Also, the API for accessing remote objects is the same in every .NET environment - there is no great variability of distributed object libraries.

On the down side, the problem of this approach is that the developer does not have very much control over how .NET manages threads, network resources (such as sockets), memory, activation and deactivation of objects, etc. Compare this to CORBA where the developer is in full control of these aspects - if needed. .NET Remoting is a typical example where this kind of flexibility is sacrificed for ease of use. The learning curve should be as shallow as possible making Remoting technology available to as many people as possible. Performance-critical applications that do not fit the access patterns assumed by the .NET framework developers at Microsoft are hard to realize in such an envi-

ronment. In the typical office/enterprise world this should only seldom be a real problem. In distributed embedded real-time systems such an approach usually does not work - but that is not the typical usage scenario for the .NET framework.

Note that this does not mean that .NET Remoting is not flexible: to the contrary, it is *very* flexible with regards to application-level/infrastructure-level adaptations: leases, sponsors, message sinks, and custom channels provide an excellent way of adapting .NET Remoting to specific application requirements.

Interface Descriptions and Proxies



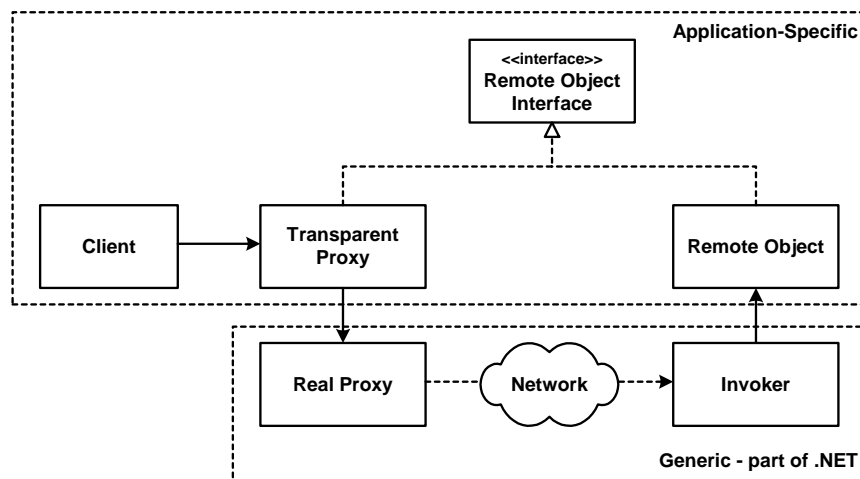
As with any distributed object middleware, a CLIENT PROXY is used by the client to invoke operations. The CLIENT PROXY is subsequently responsible for forwarding the request over the network to the remote object. The CLIENT PROXY is a local object in the client's address space and thus can receive ordinary, local method invocations.

In .NET Remoting, the CLIENT PROXY is separated into two parts:

- The so-called *transparent proxy* is the CLIENT PROXY in the sense of this pattern language. It has the same interface as the remote object and forwards invocations across the network.
- The *real proxy* is actually a part of the client-side facilities of the .NET Remoting infrastructure. It receives the invocation data from the transparent proxy and forwards it over the network. Thus, it plays the role of the REQUESTOR. However, it is not accessible by the client program directly.

As explained in the client proxy and requestor patterns, this separation into transparent and real proxy does make sense because only the transparent proxy is specific to the interface of the remote object, the real proxy is independent of this interface. The same real proxy class is common to all kinds of remote objects, which is the reason why it is

actually a part of the .NET framework. The following illustration shows this:



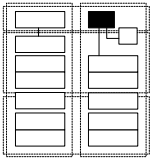
The interesting thing about the transparent proxy is that you never really see its implementation code. There is no explicit code generation step involved here. While in CORBA, for example, you have to manually code-generate the **CLIENT PROXY**, this is not necessary in .NET. .NET Remoting takes advantage of the features provided by the .NET runtime which transparently generates the proxy on-the-fly. Let's look at this a little bit closer.

If a client wants to communicate with a remote object, it typically uses the `Activator.GetObject` operation to obtain a reference. This operation returns a **CLIENT PROXY** for the requested remote object, optionally contacting the server application to physically create the instance (depending on the activation mode). The client specifies two parameters: the type of the remote object (as an instance of the `Type` type) as well as a URL that points to the required object. Since the client passes the type object to the operation, this approach requires the remote object type to be available on the client already. There is no need to transfer the interface to the client as part of the `Activator.GetObject` call.

The returned transparent proxy implements the interface of the remote object by forwarding each method invocation to the real proxy that is connected to the server. The on-the-fly code generation is made

possible by the `Reflection.Emit` namespace's classes, which can be used to generate MSIL code at runtime, packaging it into an in-memory assembly for direct execution in the running CLR instance.

To find out about the interface of the remote object (the interface that must also be used for the generated transparent proxy), the client runtime uses reflection on the type parameter given in the `Activator.GetObject()` call.



So what exactly is the interface of a remote object? .NET does not require developers to define an explicit interface for remote objects. By default, the public methods of the remote object class constitute the interface of the remote object. To allow the client to use reflection in order to build the transparent proxy, this implies that the implementation of the remote object is actually available to the client. However, it is not very good design practice to transfer the implementation class's code to the client, just to invoke remote operation invocations.

It is therefore good practice to define an explicit interface for remote objects, as illustrated in the next code snippets.

```
namespace PatientManagementShared
{
    public interface IPatientManager
    {
        Patient getPatientInfo( String id );
    }
}
```

The remote object hosted by the server then has to implement this interface:

```
namespace PatientManagementServer
{
    public class PatientManager :
        MarshalByRefObject, IPatientManager
    {
        public Patient getPatientInfo( String id )
        {
            return new Patient( id );
        }
    }
}
```

Using this idiom, only the code of the interface `IPatientManager` is needs to be available on the client, the implementation class is used only on the server. Note that the implementation code is not necessary

at the client anyway - the implementation on the client side must only package the invocation and forward it to the real proxy.

As a consequence of this process, we can write code that uses only the interface type on the client, as opposed to the remote object implementation class:

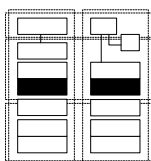
```
IPricingFactory pf = (IPricingFactory)Activator.GetObject(
    typeof( IPricingFactory ),
    "tcp://localhost:6642/PricingFactory" );
```

Extensibility of .NET Remoting

In this section we look at some advanced features of .NET Remoting, which are mostly concerned with extensibility. These features include:

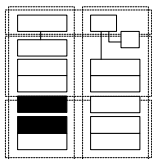
- INVOCATION INTERCEPTORS provide hooks into .NET Remoting's message handling architecture.
- INVOCATION CONTEXTS allow additional data to be sent with remote method invocations.
- PROTOCOL PLUG-INS allow for adapting the communication protocol.
- Custom MARSHALLERS allow adapting the marshalling format for specific needs.

Intercepting the Invocation Stream



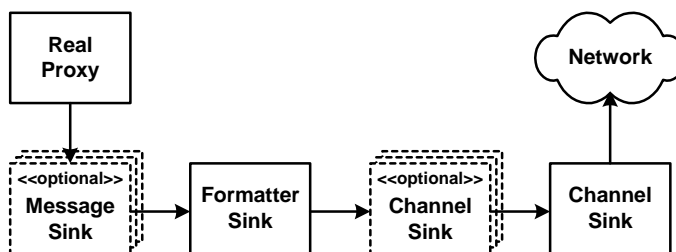
This section will provide some details on how .NET Remoting works internally and how developers can extend some of the functionality. Note, however, that for reasons of brevity we cannot delve into too much detail here. Please refer to [Ram02] for more information.

.NET can use several communication *channels* for access to remote objects. A channel provides a certain configuration for accessing remote objects. This includes protocols, the endpoint configuration, a message serializer and, as we shall see, INVOCATION INTERCEPTORS, if required. The Remoting framework in .NET is built as a layered system. Let's look at the following illustration first.



The message sink chain on the server and the client side are conceptually similar, but not identical. Let's look at the client side. There are two very important sinks that play a vital role in the framework:

- The *formatter sink* serialize the message objects into a format suitable for transfer over the network.
- The *channel sink* handle the actual transfer of the serialized message over the network.



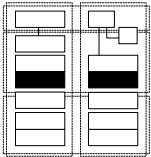
The extensibility of the .NET framework is largely due to the fact that it is possible to register application-specific sinks and thereby provide INVOCATION INTERCEPTION. There are sinks that have to be configured statically (i.e. when a channel is created, either using the API or using the XML configuration file) and others, the dynamic sinks, that can be configured into the chain at runtime. The static sinks can be further grouped into two kinds:

- *Message Sinks* intercept the stream when the message object is still in object form and not yet serialized.
- *Channel Sinks* operate on the already serialized messages.

As the two kinds of sinks work with different data (message object vs. serialized data stream) their interfaces are different. The following interface is the one used for those sinks working on the message level (i.e. before the message has been marshalled):

```
public interface IMessageSink
{
    IMessageSink NextSink{ get; }
    IMessageCtrl AsyncProcessMessage(IMessage msg,
                                     IMessageSink replySink);
    IMessage SyncProcessMessage(IMessage msg);
}
```

The two kinds of sinks are marked <<optional>> in the above illustration because you need not necessarily to have them in a system; the minimum system configuration needs to have only a formatter sink and a channel sink (as well as some additional sinks that are beyond the scope of this book).



Sinks can either just inspect messages or modify them. For example, a channel sink can encrypt the message stream. The piece of code below shows the interface of a dynamic sink:

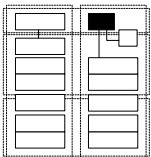
```
public interface IDynamicMessageSink {
    void processMessageStart( IMessage request,
                             bool clientSide, bool isAsync );
    void processMessageFinish( IMessage reply,
                              bool clientSide, bool isAsync );
}
```

The processMessageStart operation is called by the framework when an request message is sent to the server. The message is passed as an argument. processMessageFinish is called for the reply.

On the server side, there is also a formatter sink (which deserializes a request and serializes the reply) as well as a channel sink (that handles network communication). For many tasks, the client- and server-side sinks must be compatible, for example we need to use the same formatters on both sides. The same is true for custom sinks. If you plug-in an encryption sink on the client, you need a compatible decryption sink on the server. Note that there are also sinks that make do not need collaboration from „the other side“; an example could be a logging sink.

It is also worth noting that the .NET framework itself uses sinks extensively. For example, there is an SDLChannelSink that creates WSDL [CCM+01] INTERFACE DESCRIPTIONS for a .NET Remoting interface.

Invocation Interception using Contexts



As mentioned before, it is possible to associate objects running in the same (or different) application domains to different contexts. A context basically defines the “services” the runtime should apply to the objects within the context. Developers can explicitly associate objects with a context, making this feature an implementation of CONFIGURATION GROUPS. By associating a remote object with a specific context, you can define how the framework handles the object.

These services provided by a context

- are implemented using `INVOCATION INTERCEPTORS`, basically in the same way as explained in the section above, and
- are “requested” for a certain type by adding certain `ContextAttributes` to the type.

While the purpose of this facility is to allow framework developers to add their own interception-based services, there are some services provided by the .NET framework directly. For example, you can make sure concurrent access to instances of the `SomeClass` class is synchronized by applying the `Synchronization` attribute to a `ContextBoundObject`:

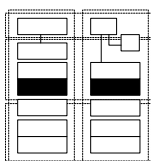
```
using System.Runtime.Remoting.Contexts;
[Synchronization]
public class SomeClass: ContextBoundObject
{...}
```

The implementation follows the same approach as in case of remoting. A transparent proxy is generated; the real proxy forwards the invocation message (an instance of `IMessage`) to the first sink in the chain which, after doing its business, forwards the message to the next sink until it reaches the `StackBuilderSink`, and finally, the destination object.

To implement a sink that can be used in such a scenario, developers have to implement the `IMessageSink` interface, which has been shown before.

Of course, since communication between the client and the target object does not have to cross application domain (or even process) boundaries, no expensive marshalling is required. There is an efficient `CrossContextChannel` provided by the .NET framework. More details are beyond the scope of this book and can be found at [Low03].

Note that the context feature provides only a implementation of `CONFIGURATION GROUPS` since many key characteristics cannot be defined using contexts, such as activation modes, associations between channels and contexts as well as (thread) priorities.



Invocation Contexts

`INVOCATION CONTEXTS`, called `CallContexts` in .NET, can be used to transport information from a client to a remote object (and back) that is

not part of the invocation data (target object, operation name, and parameters). Examples include security credentials, transaction information, or session IDs. The main motivation for using `INVOCATION CONTEXTS` is that the signature of the remote object operations should not need to feature these additional parameters and that the client does not need to explicitly provide them when invoking an operation. This allows adding context data without changing clients or remote objects, and to develop frameworks that handle this kind of data transparently for the developer. So how do these additional information items get inserted into the `INVOCATION CONTEXT`? Typically they are inserted by `INVOCATION INTERCEPTORS` (aka sinks) on the client and the server.

Let's first look at the API that allows passing of context data from client to server. In principle, the data is an associative array that contains name-value pairs.

```
SomeObject someObject = new SomeObject();
CallContext.setData( "itemName", someObject );
```

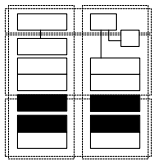
On the server, you can obtain the data item again:

```
SomeObject someObject =
    (SomeObject)CallContext.getData( "itemName");
```

The only precondition that must hold is that all objects that should be transferred as part of a `CallContext` must implement the `ILogicalThreadAffinitive` interface - a simple marker interface. The following is a legal class for `CallContext` transfers:

```
public class SomeObject : ILogicalThreadAffinitive {
}
```

Protocol Plug-ins and Custom Marshallers



.NET provides pluggable message formats as well as exchangeable network stacks. As you have seen above, this variability is realized using sinks, which are implementations of the `INVOCATION INTERCEPTOR` pattern. .NET provides two ways of configuring the channels with respect to sinks, and specifically, with respect to message formats and network stack adapters: using an API from within the application or using an XML configuration file. We will look at both alternatives in turn.

In previous examples, you have typically seen the statement `RemotingSupport.setupClient` and

`RemotingSupport.setupServer`. These are utility functions that hide the setup of formatters and network stacks. The following is the code from the `RemotingSupport` class:

```
namespace RemotingHelper
{
    public class RemotingSupport
    {
        public static void setupServer()
        {
            TcpChannel channel = new TcpChannel(6642);
            ChannelServices.RegisterChannel(channel);
        }
        public static void setupClient()
        {
            TcpChannel channel = new TcpChannel();
            ChannelServices.RegisterChannel(channel);
        }
    }
}
```

The `setupServer` instantiates a `TcpChannel` and listens on port 6642. The client also defines a `TcpChannel`, however, we do not specify the port, because the client sockets will connect to whatever server port specified in the parameters of the `Activator.GetObject` operation.

The same configuration data can be specified in a configuration file.

```
<!-- file myapp.exe.config -->
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel ref="tcp" port="6642"/>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

In order to configure an application, the configuration file must be loaded from within the application:

```
RemotingConfiguration.Configure( "myapp.exe.config" );
```

If you want to specify a certain formatter to be used in the server, you can add the following elements to your configuration file:

```
<channels>
  <channel ref="tcp" port="6642">
    <formatter ref="binary"/>
  </channel>
```

```
</channels>
```

In order to provide application-specific PROTOCOL PLUG-INS or custom MARSHALLERS, developers can implement their own classes for these two tasks and register them with the .NET Remoting framework. Details are beyond the scope of this text.

Note that there is no association between a channel and a particular remote object. All remote objects hosted by a server can be reached by all channels that are configured for this server.

As an example of a custom PROTOCOL PLUG-IN for .NET you could take a look at an IIOP implementation for .NET, described in [OG03], or at Borland's Janeva [Bor04].

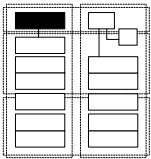
QoS Observers and Location Forwarders

QOS OBSERVERS are not provided by .NET remoting out of the box. However, it is rather simply to build your own facilities here, using sinks. You can define a message sink on the server side where developers can register QOS OBSERVERS that are notified if messages arrive for a specific remote object.

Also, LOCATION FORWARDERS are not provided by .NET Remoting.

Asynchronous Communication

.NET provides a comfortable API for asynchronous remote communication. Actually, the facilities are not only usable for remote invocations, but also for local invocations. As you will see, the runtime with its dynamic features provides the basis for the functionality.



Asynchronous invocations are handled exclusively by the client of an invocation. This means that the remote object as well as the server runtime is not affected at all. While this is an advantage, there is also a serious problem with this approach: All the asynchrony is handled by executing code in a separate thread on the client side. This approach is not always the most efficient one, since threads are potentially expensive. In some cases, you could use network characteristics (such as the UDP protocol) or a different server implementation to implement asynchrony. As all this happens internally within the Remoting framework, you cannot easily adapt the behavior to your own requirements.

When reading the following explanations of .NET Remoting's asynchronous communication features, you should bear in mind that you can of course also implement your own approach. For example, it you can implement server-side asynchrony using an asynchronous handler as a server-side message sink. However, this requires some more intricate programming as well as changes to the server application configuration (adding a certain message sink). The target remote object need not be changed.

The asynchronous features in .NET are implemented using so-called *delegates*. Since they play a central role in understanding asynchrony, we will start with an introduction of delegates. Consider, the class `SimpleCalculator`:

```
public class SimpleCalculator {
    public int add( int a, int b ) {
        return a+b;
    }
}
```

A delegate can be seen as a kind of "object-oriented function pointer." In our example, we want to be able to define a "function pointer" to the `add` method. As with function pointers, say, in C, a delegate is typed by the method signature and the return type. So, let's define a delegate that can point to methods that have two ints as arguments and also returns an int.

```
public class ThisIsAnotherClass {
    delegate int ExampleDelegate( int, int );
}
```

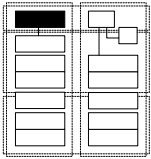
This delegate is called `ExampleDelegate` and can point to methods that have two integers as arguments and also return an int. A delegate is a type declaration (actually, it's a reference type) and instances can be created. The instance takes the actual method to which the instance should point as an argument. This target operation needs to be compatible regarding its signature.

```
public class ThisIsAnotherClass {
    delegate int ExampleDelegate( int, int );
    public void ExampleOperation() {
        SimpleCalculator sc = new SimpleCalculator();
        ExampleDelegate d = new ExampleDelegate( sc.add );
        int sum = d( 3, 7 ); // d calls sc.add(), sum is now 7
    }
}
```

`d(...)` indirectly calls the `add` operation of the `sc` object. This is because the constructor of a delegate takes the target operation that should be invoked when an invocation is done on the delegate (the `d(...)` statement).

Delegates in .NET can also point to multiple methods at the same time. In that case, .NET builds a linked list of method pointers internally and calls each method in turn when the delegate is invoked. This feature is heavily used in .NET's event mechanism but is not relevant to the following discussion.

Poll Objects



Now, what is the relationship of this to asynchronous remote method invocations? Let us start by looking at the POLL OBJECT pattern and its implementation in .NET. Consider again the remote version of the `SimpleCalculator`. We will first look up an instance of the class:

```
ISimpleCalculator sc = (ISimpleCalculator)Activator.GetObject(
    typeof( ISimpleCalculator ),
    "tcp://localhost:6642/SimpleCalculator" );
```

We can then declare a delegate to the `add` operation of this object. Actually, the delegate is defined for the transparent proxy instance that is returned from the `Activator.GetObject` operation:

```
delegate int RemoteAddCallDelegate( int, int );
```

We now instantiate the delegate, pointing to the `add` operation of `sc`.

```
RemoteAddCallDelegate racd = new RemoteAddCallDelegate( sc.add );
```

You can now call an operation `BeginInvoke` on this delegate instance that will invoke the delegate's target operation asynchronously:

```
racd.BeginInvoke( 3, 4, null, null );
```

`BeginInvoke` is code-generated automatically by the compiler when it comes across the delegate declaration. It has (in this case) four parameters. The first two are the two `ints` of the `add` operation. Other operations have other parameters, depending on the signature of the target operation. The other two parameters which have `null` values in the example above. These are used for callbacks, more details will be provided below.

Invoking the operation executes the operation asynchronously. Of course, there is no POLL OBJECT yet. However, the `BeginInvoke` oper-

ation returns an instance of `IAsyncResult` which serves as the POLL OBJECT:

```
IAsyncResult ares = racd.BeginInvoke( 3, 4, null, null );
```

We can now do several things with this `IAsyncResult`. First of all, we can ask the object whether the result of the operation is available yet. This is a non-blocking operation and returns a `bool`:

```
bool isResultAvailable = ares.IsCompleted;
```

We can also do a blocking wait until the result arrives:

```
ares.AsyncWaitHandle.WaitOne();
```

The main point of the POLL OBJECT, however, is to provide access to the return value of the asynchronously invoked operation. The `EndInvoke` operation of the delegate allows access to it:

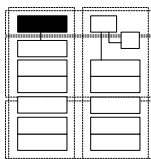
```
int sum = racd.EndInvoke( ares );
```

We have now invoked the operation asynchronously and received the result back to the calling thread. Note that no explicit downcast of the result was necessary. The compiler knows that the delegate returns an `int`.

It is also possible to catch exceptions thrown by the asynchronously invoked operation. To do this, simply surround the `EndInvoke` operation in a try/catch clause.

```
try {
    int sum = racd.EndInvoke( ares );
    Console.WriteLine( "result:" + sum );
} catch ( Exception e ) {
    Console.WriteLine( "oops, exception:" + e );
}
```

Result Callbacks



RESULT CALLBACKS are implemented on top of what we have seen above, namely delegates and the `IAsyncResult` interface. In order to provide a RESULT CALLBACK, we need a callback operation that is called by the .NET framework once the result of an asynchronous operation becomes available. Such a callback operation must be a void operation that takes exactly one argument of type `IAsyncResult`:

```
public class ThisIsAnotherClass {
    public void TheResultIsHere( IAsyncResult res ) {
        // ...
    }
}
```

```
// ...
}
```

The invocation of the asynchronous operation works as before, except that the first null argument is replaced by a `AsyncCallback` delegate instance pointing to the callback operation:

```
public class ThisIsAnotherClass {
    // ...
    public void ExampleOperation() {
        // obtain a RemoteAddCallDelegate instance as above; racd
        racd.BeginInvoke( 3, 4,
            new AsyncCallback(this.TheResultIsHere), null );
    }
}
```

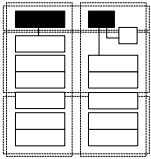
When the result is available, the .NET framework invokes the delegate instance, that is, the `TheResultIsHere` operation of the current instance of the `ThisIsAnotherClass`. In this operation we can now access the result in the same way as before:

```
public class ThisIsAnotherClass {
    public void TheResultIsHere( IAsyncResult res ) {
        RemoteAddCallDelegate racd = (RemoteAddCallDelegate)
            ((AsyncResult)res).AsyncDelegate;
        Console.WriteLine( "result: " + racd.EndInvoke(res) );
    }
}
```

The compiler creates the `EndInvoke` operation in a way that returns the result of the target operation - without a manual downcast.

.NET also supports the *Asynchronous Completion Token* (ACT) pattern [SSRB00]. The last parameter to `BeginInvoke`, null in the example, can be used to supply an arbitrary value (the ACT) that can be accessed from the callback method in order to identify the asynchronous request.

Oneway Operations



Oneway operations are the .NET implementation of FIRE AND FORGET; that is, the .NET framework provides best-effort semantics. In other words, nobody knows if the operation is actually executed. Oneway operations - when used in the simplest way - provide no feedback about the result of an invocation execution, be it a return value or an exception. This is the reason why oneway operations must have a `void` return value.

To make an operation a oneway, the operation declaration needs to specify this. Oneway operations have to have the `[OneWay()]` attribute. The following example shows this:

```
public interface ILogger {
    [OneWay()]
    void logMessage( string message );
    void store();
}
```

The `logMessage` operation is declared to be oneway. So, invocations of this operations have FIRE AND FORGET semantics, while the `store` operation is not oneway - it is invoked synchronously. The client can now invoke the operation without the `BeginInvoke/EndInvoke` construct, as the following source code example shows:

```
ILogger logger = (ILogger)Activator.GetObject(
    typeof( ILogger ), "tcp://localhost:6642/logger" );
logger.logMessage( "hello!" );
```

This invocation is handled asynchronously, although it looks like a normal synchronous method call. There is an important point to make here, though: Although we have modified the interface definition of the remote object by adding the `[OneWay()]` attribute, the asynchrony is still handled on the client! When the transparent proxy is generated by the runtime, its implementation contains the necessary code to invoke the operation in a separate thread. Note the difference to the other asynchronous invocation techniques explained in this chapter: in case of POLL OBJECTS and RESULT CALLBACKS, it is the caller of the operation who decides whether the operation should be called synchronously or asynchronously. In case of oneway operations, it is defined as part of the operations interface: so the server (the provider of the interface) decides that the operation should always be invoked asynchronously.

Let's finally look at exceptions. Exceptions thrown by the remote object's oneway operation are ignored. They are not reported back to the client. The same is true for exceptions thrown by the client proxy or framework. For example, if the server application is down or unreachable, the operation *still succeeds* from a client's point of view. No exception is thrown, although the invocation never reaches the server. Note that this is true even if you invoke the oneway operation using delegates plus `BeginInvoke/EndInvoke` and enclose the `EndInvoke` call into a try/catch clause.

Other Asynchronous Invocation Modes

In .NET, SYNC WITH SERVER semantics are not possible out of the box. As mentioned above, all the described asynchrony is implemented with the help of threads on the client-side. If we need SYNC WITH SERVER, we have to implement it on our own. This means, however, the server needs to be changed. The simplest way to achieve this is to execute the business logic on the server in a separate thread. This could be implemented in the operation itself, or with the help of a suitable message sink.

MESSAGE QUEUES are also not provided as part of .NET Remoting. However, Microsoft's MSMQ messaging queue can be integrated into .NET applications.

Outlook to the Next Generation

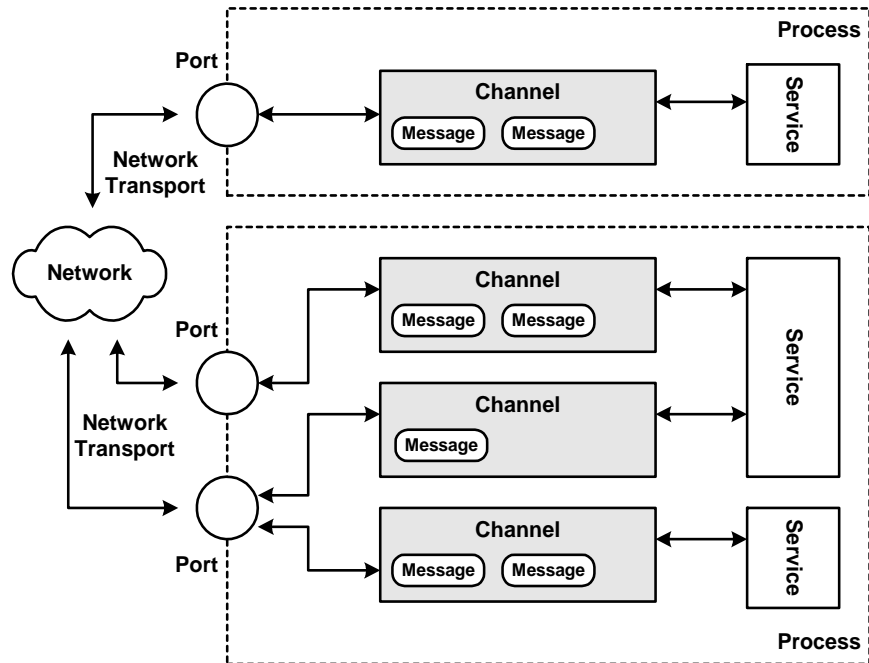
Indigo is Microsoft's upcoming technology for distributed ("connected") systems. Indigo subsumes several current APIs and technologies, such as .NET Remoting, ASMX and COM+. It is a service-oriented framework that makes communication boundaries explicit, as opposed to trying to hide these as in most other current remoting frameworks. Indigo is based on a messaging abstraction, but in contrast to current messaging infrastructures, however it provides convenient support for request/reply style programming and other higher level programming models as well as additional services such as transactions or security. Indigo also provides transport over several protocols, among them the well-known Web Service infrastructure based on SOAP. There are several hosting environments. Indigo components can be run inside normal applications, as a Windows service, or in the context of the IIS.

Core Components

At its core, Indigo consists of the following components:

- *Ports* constitute network endpoints that can be addressed remotely, they also define the supported protocol.
- *Message Handlers* contain the actual application logic to handle the messages.

- *Channels* connect ports with message handlers. Channels are used to configure additional services, such as transactions and security, and to configure the programming model, such as reliable or unreliable messaging.



Application developers first have to define ports, then configure the channels that provide access to the ports (including a channel's services), and finally register message handlers with the ports. Note that there is no notion of a server or a client, since any process can receive and send messages.

Pattern Mapping

Ports implement the CLIENT REQUEST HANDLER and SERVER REQUEST HANDLER patterns. Channels fulfill multiple roles:

- they implement the role of a REQUESTOR, accepting Message objects,
- they are part of the INVOKER, dispatching incoming Message objects to registered Message Handlers, and

- they are INVOCATION INTERCEPTORS, in the sense that additional services can be “plugged in” to the channel.

Regarding identification, OBJECT IDS are strings, such as “simple”. ABSOLUTE OBJECT REFERENCES further define the protocol and the host and port at which the service is provided, so that a complete ABSOLUTE OBJECT REFERENCE looks like “soap://localhost:50000/simple”.

As messaging is by nature asynchronous, a specific setup is needed to use it for synchronous invocations. For client-side synchronous request/reply style, a separate RequestReplyManager, provided by the framework, must be used to send requests and receive results synchronously. For server-side synchronous request/reply style, a SyncMessageHandler must be used.

16 Web Services Technology Projection

Web Services provide a standardized means of service-based interoperation between different, distributed software applications. These applications might run on a variety of platforms, programming languages, and/or frameworks. The use of Web Services on the World Wide Web is expanding rapidly as the need for application-to-application communication and interoperability grows [BHC+03]. The goals of Web Services go beyond those of classical middleware frameworks, such as CORBA, DCOM, or RMI: they aim at standardized support for higher-level goals such as service and process flow orchestration, enterprise application integration (EAI), and providing a “middleware for middleware” [Vin03].

Web Services are defined by a number of protocols and standards. The implementations of these protocols and standards in different Web Services frameworks vary, but share many characteristics. For instance, the APIs, used in Web Service frameworks, are not standardized but are typically quite simple. We use Apache Axis [Apa03] as a representative and well-known Web Service framework in the main part of this technology projection. This chapter shows how Axis is designed using the patterns presented in this book. We also discuss other Web Service frameworks towards the end of this chapter, namely Glue [Min03], Microsoft’s .NET Web Services, IONA’s Artix, and Leela [Zdu04c]. These frameworks are only discussed briefly to show remarkable characteristics of how the patterns are used by these frameworks.

Brief History of Web Services

Before we take an in-depth look under the hood of current Web Service frameworks, let us take a brief look at the bigger picture and understand the motivation for Web Services.

Consider a situation in which multiple business processes are co-existing and cutting across multiple departments or even companies.

Each of these parties has its own IT infrastructure, including many different existing applications, programming languages, component and middleware platforms (for instance J2EE, .NET, CORBA, or MQSeries), backends, and third-party systems (such as ERP or CRM systems). Somehow this heterogeneous IT landscape has to be integrated.

The goal of Web Services is to integrate such heterogeneous environments by enabling a standardized, *service-oriented architecture* (SOA). The basic concept of a SOA is quite trivial: A service is offered using a message-oriented remote interface using some kind of a well-defined INTERFACE DESCRIPTION. The service can be implemented, for instance, as a remote object or as any other kind of service implementation, such as a set of procedures. The service advertises itself within another central service, the lookup service. This service realizes the pattern LOOKUP, and it is itself offered as a STATIC INSTANCE remotely. Thus applications can look up the advertised services by name or properties to find the details for interacting with the service. As we have seen in the other technology projections, service-oriented architectures can be built with most modern middleware systems, not only with Web Services.

There are many things that are called “Web Services”. For the purpose of this chapter, and without prejudice toward other definitions, we will use the following definition similar to that of [BHC+03]:

A Web Service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web Service in a manner prescribed by its description using messages (specifically SOAP messages). Messages are typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards, but any other communication protocol can be used for message delivery as well.

Web Services emerged from the World Wide Web (WWW). The WWW was initially designed for unstructured information exchange. The information in HTML text is primarily designed to be consumed by human beings. Over time the Web got used for machine-to-machine

communication as well, such as e-commerce applications and business-to-business (B2B) communication. XML [BPS98] and many XML-based standards have emerged to facilitate structured information exchange on the Web. Early e-services, however, have exposed their interfaces in an ad-hoc manner, meaning that there was no standardized interoperability between services.

The XML-RPC specification was released in 1999 [Win99] as a simple remote procedure call specification. To further facilitate automated access to complex services, a group of companies including IBM and Microsoft (and now being handled by the W3C) have released a number of Web Service standards, including SOAP [BEK+00], WSDL [CCM+01], and UDDI [Oas02]. SOAP is the successor of XML-RPC.

Today's Web Services are distributed services with WSDL interfaces. They are more often message-oriented than they are RPC-oriented. WSDL allows for a variety of transports, protocols, and message formats. SOAP over HTTP is just one possible Web Services protocol.

A number of successful middleware frameworks, such as CORBA, DCOM, .NET Remoting, or RMI, have been around for a while and can also be used to build complex services. The question arises, aren't Web Services just reinventing the wheel? In the middleware integration world, however, interoperability problems occurred that are similar to the early interoperation problems of e-service. As Steve Vinoski puts it [Vin03]: "Unfortunately, middleware's success and proliferation has recreated — at a higher level — the very problem it was designed to address. Rather than having to deal with multiple different operating systems, today's distributed-application developers face multiple middleware approaches". Among other things, a goal of Web Services is to provide a "middleware for middleware".

In this section, we explain some central differences of the Web Service approach compared to other middleware approaches. Internally, Web Services are implemented using the same remoting patterns as the other distributed object middleware systems - which is what we will illustrate in the remainder of this technology projection.

Web Services are designed around the stateless exchange of messages. This design is originally due to the stateless nature of the HTTP protocol. In HTTP, stateful interactions are implemented as extensions on top of HTTP, such as cookies or sessions. Web Services use a similar

stateless request/response scheme for a message-oriented interaction, leading to a loose coupling of clients and servers.

SOAP is an XML-based message exchange format that is used in almost all Web Service frameworks available to date. It can be extended in various ways, including secure, reliable, multi-part, multi-party, and/or multi-network transport of messages. This also allows the messaging infrastructure to provide authentication, encryption, access control, transaction processing, routing, delivery confirmation, etc. SOAP has quickly become a de-facto standard in the Web Service domain. It is often equaled to “Web Service” in general, which is not quite correct, as there are also other Web Service protocols around and most applications of Web Service use more technical infrastructure than just a message exchange protocol. As shown below, there is a “stack” of different components (often different in different implementations) that together provide a Web Service infrastructure.

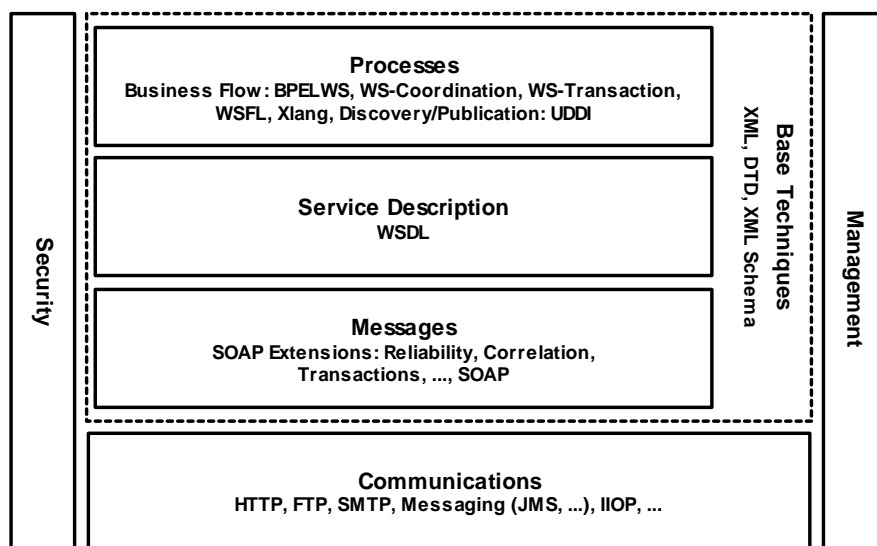
WSDL is used for XML-based service descriptions (among other information, it contains INTERFACE DESCRIPTIONS) commonly understood by Web Services producers and consumers. It facilitates interoperability across heterogeneous systems by providing the precise structure and data types of the messages.

Web Services do not require any particular communication protocol. For instance, HTTP, other Internet protocols such as SMTP and FTP, generic interface APIs such as JMS, other distributed object protocols such as IIOP, or almost any other communication protocol can be used. Most Web Service implementations today, however, use HTTP as their communication protocol.

Above the level of individual message exchanges, Web Services standardize process descriptions. This includes UDDI as a lookup service, a kind of service that can be found in many other distributed object middleware as well. But it also includes XML-based standards for higher-level business transactions, multi-part and stateful sequences of messages, and the aggregation of elementary processes into composite processes. For instance, the Business Process Execution Language for Web Services (BPEL4WS) [ACD+03] is a XML-based workflow definition language that allows businesses to describe business processes that can both consume and provide Web Services. BPEL emerged from IBM’s WSFL and Microsoft’s XLANG. The WS-Coordination [CCC+03]

and WS-Transaction [CCC+02] specifications complement BPEL4WS in that they provide a Web Services-based approach to improving the dependability of automated, long running business transactions in an extensible and interoperable way. The Business Process Management Initiative specifies the Business Process Modelling Language (BPML) [Bpm02]. BPML provides an abstracted execution model for collaborative and transactional business processes based on the concept of transactional finite-state machines.

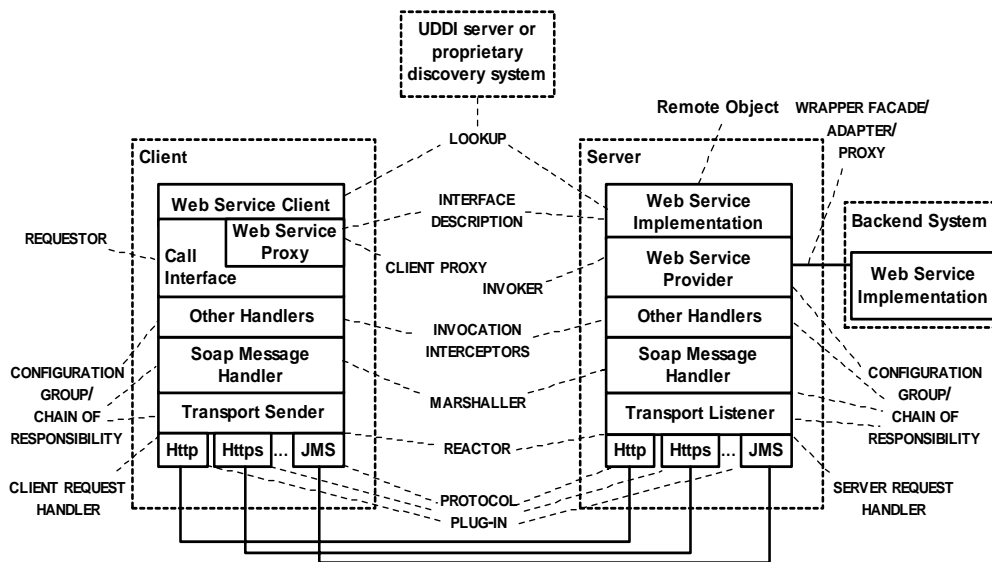
In addition to specific messaging and description technologies, the Web Service architecture also provides for security and management. These are complex areas that are applied on multiple levels of the Web Service architecture.



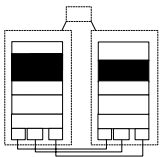
The figure above shows a high level architecture for Web Services [BHC+03]. Note that XML is the basics of the Web Service architecture. One can provide a Web Service framework that works without SOAP or WSDL, but XML is seen as more fundamental for two reasons [BHC+03]. Firstly, XML provides the extensibility and vendor, platform, and language neutrality that is the key to loosely-coupled, standards-based interoperability. Secondly, XML helps to blur the distinction between “payload” data and “protocol” data, allowing easier mapping and bridging across different communications protocols.

Pattern Map

The following diagram summarizes the usage of the Remoting Patterns within the presented Web Service architectures. In the figure, we use the terms from Apache Axis, but this general static architecture is very similar in the other Web Service frameworks, discussed towards the end of this chapter. Note that purely dynamic patterns, such as those in the *Client Asynchrony* or the *Lifecycle Management* chapters, are not visualized in the figure.



SOAP Messages

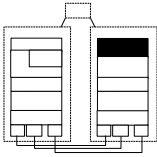


SOAP [BEK+00] is a standard, XML-based protocol used for invoking Web Service via messages. It supports RPC-style invocations (request/response) and other distribution paradigms, such as asynchronous messaging or one-way messages. From a developer's perspective a SOAP implementation provides the *Layer* [BMR+96] of REQUESTOR, CLIENT PROXY, and INVOKER.

In this section, we discuss two ways of using SOAP from the perspective of developers of Web Service clients or server applications. Firstly, we show how to dynamically construct an invocation of remote objects

using a REQUESTOR and INVOKER. Secondly, we discuss how to use the INTERFACE DESCRIPTION language WSDL to generate CLIENT PROXY and INVOKER. Axis supports the client invocation models defined by the Sun's JAX-RPC API standard [Sun04b].

Dynamic Invocation of Remote Objects on Server Side



Let us jump right into an Apache Axis example to explain how the *Basic Remoting Patterns* are realized for Web Services. We will see that Web Service frameworks offer quite simple APIs. Even though these are not standardized, most other Web Service frameworks offer simple interfaces to provide or access a Web Service (see the end of this chapter for examples from other frameworks than Axis).

As an introductory example, let us consider using a SOAP-based Web Service that provides a simple date service in Java with Apache Axis. First we have to implement the service provider as an ordinary Java class:

```
package simpleDateService;
import java.util.*;
import java.text.SimpleDateFormat;

public class DateService {
    public String getDate (String format) {
        Date today = new Date();
        SimpleDateFormat formatter =
            new SimpleDateFormat(format);
        String result = formatter.format(today);
        return result;
    }
    public String getDate () {
        return getDate("EEE d MMM yyyy");
    }
}
```

This simple date service class provides two methods for querying the current date (and time), one using a default format (without parameter) and one using the given format parameter. Instances of the `DateService` class are the remote objects offered by our Web Service application.

Note that a Web Service remote object does not have to be implemented as an object internally. For instance, it can instead be implemented by

a set of procedures. This way, Web Services can be used to provide an object-oriented “view” onto a procedural application.

In the Axis framework, the most simple way to provide a Web Service in a server is to use a SOAP INVOKER. The INVOKER is set up, when the object is deployed to the Web Server. This INVOKER is running in the Web Server. We have to tell the INVOKER which services it provides and which operations are exported by each service. This is done in an XML file, called the *deployment descriptor*:

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
  xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
  <service name="DateService" provider="java:RPC">
    <parameter name="className"
      value="simpleDateService.DateService"/>
    <parameter name="allowedMethods" value="getDate"/>
  </service>
</deployment>
```

When this deployment descriptor is sent to the Axis engine running in the Web application server, the INVOKER is informed that it supports a new Web Service with the symbolic ID “DateService.” This ID is used as an OBJECT ID for the Web Service. Whenever the corresponding URL, such as:

```
http://localhost:8080/axis/services/DateService
```

is accessed by a SOAP client, the INVOKER knows that an instance of the class `simpleDateService.DateService` is responsible for handling the invocation. Thus, when an invocation reaches the server, the INVOKER looks up this class, creates a PER-REQUEST INSTANCE of this class, and lets the instance handle the request.

In the SOAP request the operation name and parameters of the invocation are encoded. Before the invocation is performed, it is checked that the invocation is an “allowed method” (here only the operation `getDate` is valid). It is additionally checked that the parameter number and types are also valid. If one of these invocation preliminaries is not valid, a SOAP REMOTING ERROR is returned to the client, otherwise the result of the operation invocation.

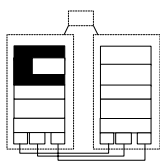
During the deployment we have announced to the server that the OBJECT ID “DateService” has to be mapped to an instance (here a PER-REQUEST INSTANCE) of the class `simpleDateService.DateService`.

Almost all Web Service frameworks provide some dynamic form of deployment. That is, remote objects can be dynamically registered and de-registered with the INVOKER. However, not all frameworks use an XML-based deployment descriptor. This XML-based deployment scheme is flexible and common for Java application servers. It decouples deployment code from application code. Compared to deployment code that is hard-coded within the application code, as in GLUE (discussed below), the downside is an additional level of complexity for developers and an overhead for XML processing. XML can only define the deployment in a declarative manner, whereas programmatic deployment specifications can also define a deployment behavior such as a condition.

Note that URLs are used as ABSOLUTE OBJECT REFERENCES by a client to access a Web Service. Using URLs as ABSOLUTE OBJECT REFERENCES is simple and eases integration of Web Services with existing Web protocol, such as HTTP, FTP, or DNS. However, URLs do not necessarily stay stable over the time; for instance, when the server is started on another machine or when the machine's host name changes, the ABSOLUTE OBJECT REFERENCES have to be updated in all clients.

The Internet's Domain Name System (DNS) [Moc87] realizes a variant of the pattern LOOKUP that is used for finding the host which belongs to an URL. By changing the DNS entry we can update the information provided to clients - thus the DNS can be used as a LOCATION FORWARDER when URLs are used as ABSOLUTE OBJECT REFERENCES. Note that DNS is not primarily designed for this task. For instance, it takes relatively long to update DNS entries. A better solution to this problem might be to use HTTP Redirect [FGM+97] or other forwarding approaches. These approaches realize a LOCATION FORWARDER for Web Services directly on the original server, and thus have an immediate effect.

Constructing an Invocation with a Requestor on Client Side



On the client side, we have to write a Web Service client that accesses the SOAP INVOKER via the network. The Axis framework provides a REQUESTOR that enables users to dynamically construct requests. The details of transporting the invocation across the network using SOAP are then handled by the Axis framework:

```

public class DateClient {
    public static void main(String [] args) {
        try {
            Options options = new Options(args);
            String endpointURL = options.getURL();
            String formatString;
            Object[] arguments = null;

            args = options.getRemainingArgs();
            if ((args == null) || (args.length < 1)) {
                formatString = null;
            } else {
                formatString = args[0];
            }

            Service service = new Service();
            Call call = (Call) service.createCall();

            call.setTargetEndpointAddress(
                new java.net.URL(endpointURL));
            call.setOperationName("getDate");
            if (formatString != null) {
                call.addParameter("format",
                    XMLType.XSD_STRING, ParameterMode.IN);
                arguments = new Object[] { formatString };
            }
            call.setReturnType(
                org.apache.axis.encoding.XMLType.XSD_STRING);
            String result =
                (String) call.invoke(arguments);
            System.out.println("Date: " + result);
        } catch (Exception e) {
            System.err.println(e.toString());
        }
    }
}

```

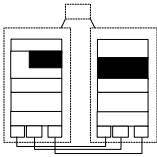
In this example, the client takes the command line arguments provided by the user and accesses one of the two operations provided by the remote object. The framework-provided `Call` class implements the pattern REQUESTOR that lets us construct an invocation from a given URL, an operation name, parameters, and a return type. Example invocations of the above client program are:

```

% java simpleDateService.DateClient
-lhttp://localhost:8080/axis/services/DateService
% java simpleDateService.DateClient
-lhttp://localhost:8080/axis/services/DateService "dd.MM.yy"

```

Generating Invoker and Client Proxy Code with WSDL



The dynamic invocation variant of REQUESTOR and INVOKER, used above, avoid static stubs for clients and so-called skeletons for the server side. The goal is to configure service information on-the-fly, what is very flexible and generic. Another benefit is that the client has full control over the invocation. As the INVOKER handles all details of the invocation, it takes only a proper deployment descriptor to deploy or un-deploy any Java class as a Web Service into the Web Service engine running in a Web Server. However, the disadvantage of this approach is that the client has to deal with remoting details directly, such as constructing the `Call` object and handling argument types. Sometimes this level of detail is required, for instance, when we want to control how `Call` objects are constructed. In other cases we can use an `INTERFACE DESCRIPTION` to hide the details of the remote invocation: from the `INTERFACE DESCRIPTION` we can generate `CLIENT PROXY` and `INVOKER` code so that the client developer does not have to deal with the details of the SOAP invocation.

The Web Services Description Language (WSDL) [CCM+01] is a standard, XML based language for `INTERFACE DESCRIPTIONS` of Web Services. It is based on the following kinds of abstract definitions and bindings:

- *Types* contain data type definitions that are relevant for later use in the messages.
- *Messages* contain the data being exchanged, in one or more logical parts. Each part has a type, and the types are extensible.
- A *port type* is a named set of abstract operations and the abstract messages involved. *Operations* to be remotely invoked can be either in-out, in-only, out-only, or out-in.
- A *binding* defines message format and protocol details for operations and messages defined by a particular port type.
- A *port* defines an individual endpoint for a binding. That is, it maps the abstract communication scheme of the binding to a concrete communication protocol.
- *Services* are defined as a collection of ports.

Each port has one binding, and each binding has a single port type. Each service, in contrast, can have multiple ports. Each of these ports is

an alternative way to access the service. Note that WSDL describes the concept of PROTOCOL PLUG-INS this way (we will see in later sections how the Web Service framework architecture mirrors this extensibility with communication protocols).

In contrast to other INTERFACE DESCRIPTIONS in other distributed object middleware, this scheme is relatively complex. The reason for this complex design is that the INTERFACE DESCRIPTION of a Web Service does not only describe the remote object's interfaces, but also the protocol binding how it can be reached and its location. Because the used protocols might be quite diverse, we require a very generic binding scheme. Therefore, almost all Web Service frameworks provide some means to generate the WSDL code for deployed Web Services or Web Service classes automatically. In many cases, this generated WSDL code can be accessed or downloaded remotely. Thus clients can download the INTERFACE DESCRIPTION how to access a Web Service, and can then generate a CLIENT PROXY (on the fly).

In Axis there are two generation tools provided: one to generate a WSDL description from an existing Java class (Java2WSDL), and one to generate CLIENT PROXY and INVOKER code from WSDL (WSDL2Java). Axis follows the JAX-RPC specification [Sun04b] when generating Java client bindings from WSDL. The main advantage of using WSDL is to avoid having to write the stub and binding (or skeleton) classes, as well as the deployment descriptors, by hand. The WSDL generation tool is also integrated in the Axis engine: for a deployed service, Axis allows one to invoke a method to generate WSDL for this service on the fly (the WSDL output can for instance be viewed via a web browser).

As an example, we simply define the remote object's services in the used programming language (here: Java):

```
package wsdlDate;
public interface SimpleDate {
    public String getDate (String arg);
    public String getDate ();
}
```

We also implement these methods in a remote object class SimpleDateImpl:

```
public class SimpleDateImpl implements SimpleDate {
    ...
}
```

Now we can use the Axis tool Java2WSDL to create a WSDL file from the INTERFACE DESCRIPTION:

```
java org.apache.axis.wsdl.Java2WSDL -o wsdlDate/Date.wsdl
-l"http://localhost:8080/axis/services/wsdlDate"
-n urn:wsdlDate -p"wsdlDate" urn:wsdlDate wsdlDate.SimpleDate
```

The tool uses the remote object interface to realize a reflective variant of generating an INTERFACE DESCRIPTION. All missing information (such as service name, namespace, package, and address) is provided as command line parameters to the tool. The generated WSDL file defines the request and response messages, the service interface, the SOAP binding of the service, and the URL address of the service. The part of this WSDL file, describing the service, looks as follows:

```
...
<wsdl:message name="getDateResponse">
  <wsdl:part name="getDateReturn" type="xsd:string"/>
</wsdl:message>
...
<wsdl:portType name="SimpleDate">
  <wsdl:operation name="getDate" parameterOrder="in0">
    <wsdl:input name="getDateRequest"
      message="intf:getDateRequest"/>
    <wsdl:output name="getDateResponse"
      message="intf:getDateResponse"/>
  </wsdl:operation>
  <wsdl:operation name="getDate">
    ...
  </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="wsdlDateSoapBinding" type="intf:SimpleDate">
  <wsdlsoap:binding style="rpc"
    transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="getDate">
    <wsdl:message name="getDateResponse">
      <wsdl:part name="getDateReturn" type="xsd:string"/>
    </wsdl:message>
    ...
  </wsdl:operation>
  <wsdl:operation name="getDate">
    ...
  </wsdl:operation>
</wsdl:binding>

<wsdl:service name="SimpleDateService">
  <wsdl:port name="wsdlDate"
    binding="intf:wsdlDateSoapBinding">
  <wsdlsoap:address
    location="http://localhost:8080/axis/services/wsdlDate"/>
```

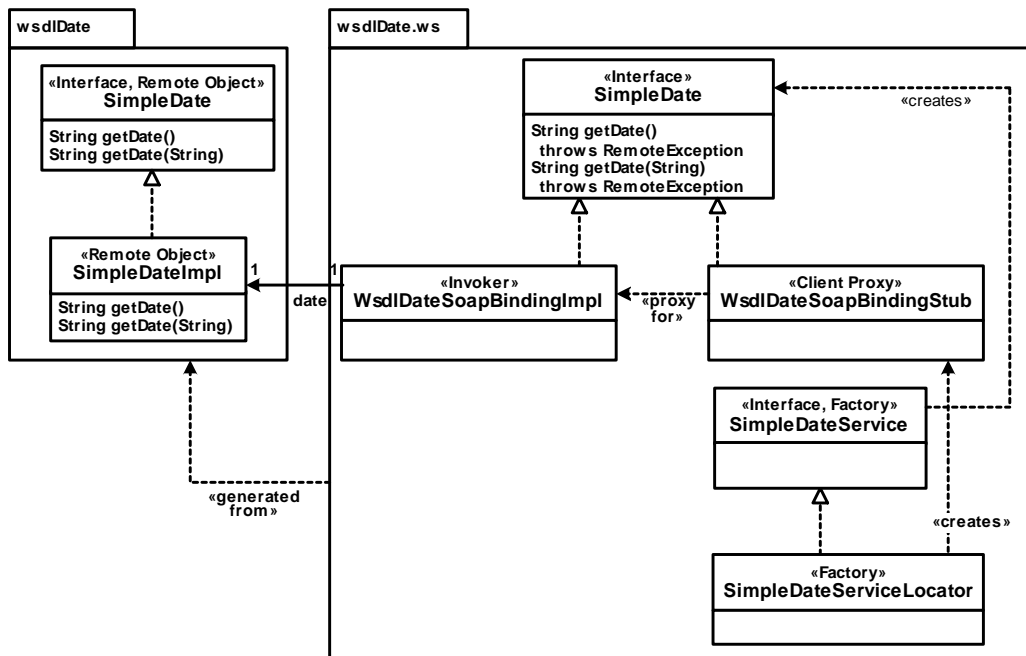
```
</wsdl:port>  
</wsdl:service>
```

The above WSDL code introduces a number of messages that are later used to define operations in port types. The port type “SimpleDate” is provided with two operations, both named “getDate”. The first of these operations receives a parameter, the other none. Each operation can receive inputs and outputs that are declared before as messages. The following WSDL binding maps the port type to a SOAP RPC invocation. The service is then used to map the port to the binding and configure the Web Service location.

An alternative for using generative tools on server side that create WSDL is to generate WSDL code directly from the deployed service at runtime. This has the advantage that the deployment information (such as service name, namespace, package, and address) is available and does not have to be specified using command line parameters. GLUE uses this variant; Axis provides both variants.

The following figure shows the classes that are generated on client and server side for the example. If WSDL is generated from the deployed service, only the client side classes need to be generated using the tool WSDL2Java (this can be specified using a command line option). In

other frameworks, such as GLUE, the CLIENT PROXY can also be generated on the fly (see the discussion of GLUE below).



The interface `SimpleDate` is the interface provided remotely. It is implemented by the server side binding and the generated CLIENT PROXY. The interface's operations throw a REMOTING ERROR (here: a `java.rmi.RemoteException`).

The generated package on server side provides classes to be used by the INVOKER. `WsdlDateSoapBindingImpl` implements the interface described in the WSDL file. It does not contain the Web Service implementation, but uses wrapper methods that simply forward invocations to the Web Service implementation. In Axis this file has to be edited by the server developer to forward invocations to the implementation class `SimpleDateImpl`. This class is hooked into the INVOKER (using the RPC provider - see next sections). Besides this class, two XML files are generated: `deploy.wsdd` to deploy the Web Service and `undeploy.wsdd` to undeploy the Web Service from the Axis engine.

On client side, `WsdlDateSoapBindingStub` contains an SOAP-based implementation of the CLIENT PROXY interface. `SimpleDate-`

Service contains an interface to access the service from the client. That means in particular it provides a *factory* for the SimpleDate CLIENT PROXY interface. SimpleDateServiceLocator contains an implementation of the CLIENT PROXY *factory* that creates objects of the type WsdlDateSoapBindingStub.

To create these files automatically, we have to invoke the WSDL2Java tool:

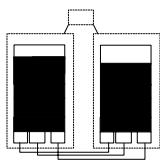
```
java org.apache.axis.wsdl.WSDL2Java -o . -d Session -s
-p wsdlDate.ws wsdlDate/Date.wsdl
```

On client side, a simpler client than the dynamic invocation client (from the previous section) can be implemented that abstracts from most of the distribution details. The following client code instantiates the *factory*, creates a CLIENT PROXY, and performs two remote invocations:

```
package wsdlDate;
public class DateClient {
    public static void main(String [] args) throws Exception {
        wsdlDate.ws.SimpleDateService service =
            new wsdlDate.ws.SimpleDateServiceLocator();
        wsdlDate.ws.SimpleDate dateProxy = service.getwsdlDate();
        System.out.println(dateProxy.getDate());
        System.out.println(dateProxy.getDate("dd-MM-yyyy"));
    }
}
```

The use of INTERFACE DESCRIPTION in Axis and other Web Service frameworks combines the benefits of using reflective code as a local interface repository with INTERFACE DESCRIPTIONS that can be sent to the client. WSDL is mainly used for providing interoperability and platform independence of Web Service implementations. With a WSDL description of a Web Service - say obtained with a lookup service such as UDDI or downloaded from the server - one can generate a CLIENT PROXY that also works with Web Service frameworks or programming languages other than Axis/Java. This way, for instance, a .NET Web Service client can access the Web Service implementation written in Axis/Java. Hence most Web Service frameworks dynamically generate WSDL code from a deployed Web Service and offer this WSDL code to be downloaded by remote clients.

Message Processing



Now that we have taken a look at the Web Service developer's view on REQUESTOR, CLIENT PROXY, and INVOKER, let's take a look at the internal message processing architecture of Axis.

There are many tasks to be performed within a Web Service REQUESTOR and INVOKER, both for the request and response of an invoked service:

- Within the REQUESTOR, the invocation has to be constructed, marshalled as an XML message, and handed to the CLIENT REQUEST HANDLER. When the response has arrived, it has to be obtained from the CLIENT REQUEST HANDLER, has to be de-marshalled, and handed back to the client.
- Within the server-side INVOKER, the invocation has to be received from the SERVER REQUEST HANDLER, a MARSHALLER has to be de-marshall the SOAP XML text, the service has to be looked up and invoked, the response has to be marshalled again as SOAP XML text, and it has to be handed back to the SERVER REQUEST HANDLER. Invocations can either use a REQUESTOR based invocation scheme (as in the first example above) or a CLIENT PROXY based invocation scheme (as in the second example above using WSDL).
- Within both, REQUESTOR and INVOKER, there are many add-on tasks, as for instance integration with the access control mechanisms of the Web Server, logging, session handling, and many more.

What we can observe here is that:

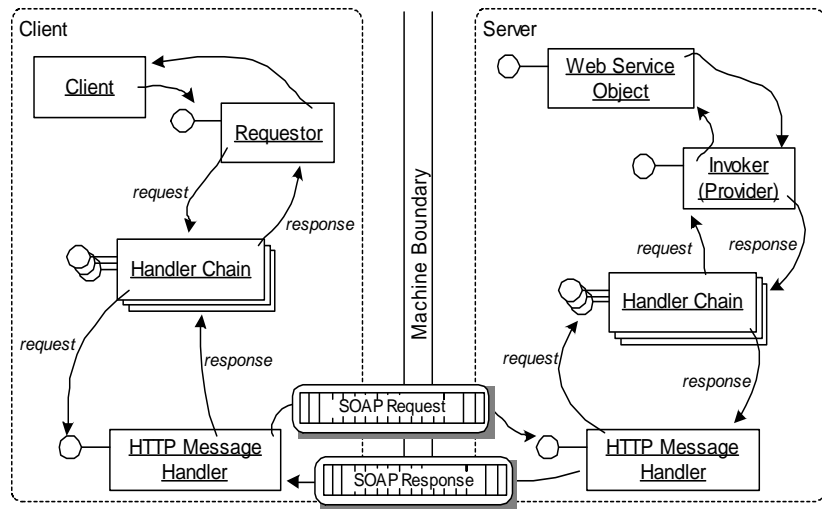
- there are many different, orthogonal tasks to be performed for a message,
- there is a symmetry of the tasks to be performed for request and response,
- similar problems occur on client side and server side, and
- the invocation scheme and add-ons have to be flexibly extensible.

These forces have led in the Axis architecture to a combination of the patterns REQUESTOR, INVOKER, INVOCATION CONTEXT, INVOCATION INTERCEPTOR, and CLIENT/SERVER REQUEST HANDLER. This message processing architecture is used both on client side and server side. The basic idea is to construct the message processing scheme and any add-on service as chained handlers. A handler basically is a *Command* [GHJV95], and these *Commands* are ordered in a *Chain of Responsibility*

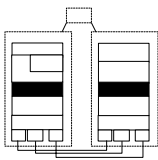
[GHJV95]. The handler chain is used to implement an INVOCATION INTERCEPTOR architecture (similar to the architecture described in [Vin02b]).

Each handler provides an operation `invoke` that implements the handler's task. This operation is invoked whenever a message passes the handler in the *Chain of Responsibility*. A message can pass a handler either as a request or response, both on client side and server side. That is, the REQUESTOR passes each request message through the client side handlers until the last handler in the chain is reached. This last handler (in Axis called "sender") hands the message to the CLIENT REQUEST HANDLER which sends the request across the network. On server side, the SERVER REQUEST HANDLER receives the request message and passes it through the server handler chain until the INVOKER (in Axis called "provider") is reached. The provider actually invokes the Web Service remote object. After the remote object has returned, the provider turns the request into either a response or a REMOTING ERROR. The response is passed in reverse order through the respective handler chains - first on server side and then on client side.

There are many different providers implemented in Axis, including a Java provider, CORBA provider, EJB provider, JMS Messaging provider, RMI provider, and others. These providers are responsible for looking up the target object and invoking the target operation, after they have checked that it is allowed to invoke the target operation, according to the deployment descriptor. Thus the provider abstraction supports heterogeneity regarding the implementation of Web Service remote objects.



In the above figure, we can see an example of how the CLIENT PROXY invokes the chain of handlers with a request. After the client side handler chain is traversed, the CLIENT REQUEST HANDLER passes the message across the network. The SERVER REQUEST HANDLER passes the received requests through the server side handler chain. The last handler is the provider that invokes the service and turns the request into a response. The response then traverses the server side and client side handler chains in reverse order.



Note that all handlers between REQUESTOR and CLIENT REQUEST HANDLER on client side and all handlers between INVOKER and SERVER REQUEST HANDLER on server side are INVOCATION INTERCEPTORS. These can perform tasks *before* an invocation, when a *request* passes the handler, and *after* an invocation, when a *response* passes the handler.

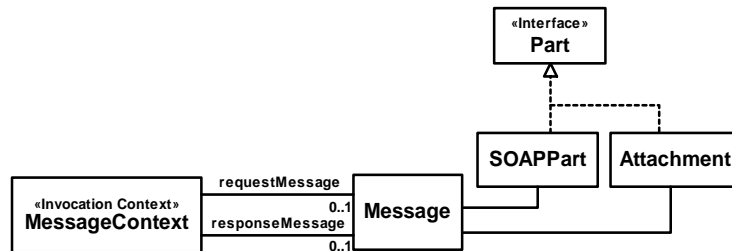
Basic tasks of REQUESTOR, CLIENT/SERVER REQUEST HANDLERS, and INVOKER are handled by these INVOCATION INTERCEPTORS. Examples are:

- interacting with the transport listener provided by the SERVER REQUEST HANDLER,
- invoking the MARSHALLER, or
- calling the URLMapper (a handler for mapping URLs to service names).

Orthogonal extensions of message processing are also handled by the same INVOCATION INTERCEPTOR architecture. Examples of such extensions are:

- LogHandler (for simple logging),
- SoapMonitorHandler (for connecting to a SOAP Monitor tool),
- DebugHandler (for dynamically setting the debug level based on the value of a soap header element), or
- HTTPAuth (for the HTTP-specific basic authentication).

To make INVOKERS and INVOCATION INTERCEPTORS work in handler chains the way described above, one more pattern is used which is of particular importance in this architecture. An INVOCATION CONTEXT (in Axis called the `MessageContext`) has to be created first, before the message is sent through the handler chain. This way different handlers can retrieve the data of the message and can possibly manipulate it. In Axis, the INVOCATION CONTEXT class `MessageContext` associates with the message object, which itself aggregates the message parts (such as SOAP parts and attachments). The corresponding class structure is illustrated in the following figure.



An INVOCATION CONTEXT is used on client and server side. Each message context object is associated with two message objects, one for the request message and one for the response message. The `MessageContext` class stores the invocation data of the message, such the target service identifier and the operation name, as well as the context information, such as a session object (if required by the service), the transport protocol name, and authentication information (user name, password, etc.).

The last handler in the client side handler chain (the sender) and the provider in the server side handler chain are also called pivot points, because they turn the message from a request into a response. A flag

havePassedPivot on the MessageContext indicates whether this pivot point is reached already. That is, each handler can find out with a method getPastPivot whether it currently processes a request or response.

Let us consider the predefined class LogHandler as an example of a handler class. It is implemented as follows:

```
public class LogHandler extends BasicHandler {
    ...
    public void invoke(MessageContext msgContext) throws AxisFault {
        ...
        if (msgContext.getPastPivot() == false) {
            start = System.currentTimeMillis();
        } else {
            logMessages(msgContext);
        }
        ...
    }
    private void logMessages(MessageContext msgContext)
        throws AxisFault {
        ...
        Message inMsg = msgContext.getRequestMessage();
        Message outMsg = msgContext.getResponseMessage();
        if (start != -1) {
            writer.println( "= " + Messages.getMessage("elapsed00",
                " " + (System.currentTimeMillis()
                    - start)));
        }
        writer.println( "= " + Messages.getMessage("inMsg00",
            (inMsg == null ? "null" : inMsg.getSOAPPartAsString())));
        writer.println( "= " + Messages.getMessage("outMsg00",
            (outMsg == null ? "null" : outMsg.getSOAPPartAsString())));
        ...
    }
    public void onFault(MessageContext msgContext) {
        try {
            logMessages(msgContext);
        } catch (AxisFault axisFault) {
            ...
        }
    }
    ...
}
```

The class is used as an INVOCATION INTERCEPTOR in the request flow and response flow. Automatically, its invoke operation is invoked whenever a message passes its position in a handler chain. In the request flow (the “before” part of the INVOCATION INTERCEPTOR) the

class remembers the start time. In the response flow (the “after” part of the INVOCATION INTERCEPTOR) the class logs the message. It uses the INVOCATION CONTEXT to obtain the request message and the response message. The previously stored start time is used to calculate the processing time of the service, if available.

Handler classes are used by specification in the deployment descriptor. For instance, we can configure the logging handler:

```
<handler name="logger"
  type="java:org.apache.axis.handlers.LogHandler"/>
```

Now we can further compose handlers into chains:

```
<chain name="myChain"/>
  <handler type="logger"/>
  <handler type="authentication"/>
</chain>
```

Either chains or individual handlers can be used in service descriptions. We can put the handler into the request flow, the response flow, or both, for instance:

```
<service name="DateService" provider="java:RPC">
  ...
  <requestFlow>
    <handler type="myChain"/>
  </requestFlow>
</service>
```

Note that the order of handlers is important. The order is conceptually handled in Axis by ordering the chains in three parts: transport chain, global chain, and service chain. Chain and flow definitions are a way to define CONFIGURATION GROUPS for Axis Web Services, implemented by handlers. These configurations can be reused for different services.

Axis uses a rather complex message processing architecture - a similar architecture can be found in the Adaptive Runtime Technology (ART) [Vin02b]. The main reason for using this combination of the extension and invocation patterns is to provide for flexibility and adaptability. Message processing in Axis is very generic and highly extensible. Another central advantage of the INVOCATION INTERCEPTOR based architecture is that almost the identical architecture can be (re)used on client side and server side - as explained above. The Web Service domain requires a high flexibility regarding the used invocation schemes, providers, protocols, marshalling, and diverse add-on services. But this flexibility is not for free. The disadvantages of this

highly flexible and reusable architecture are its complexity and its potential performance overhead. INVOCATION INTERCEPTOR based architectures need not necessarily be slower than other BROKER architectures (see [Vin02b]). But an INVOCATION INTERCEPTOR mechanism that is not properly designed and implemented can easily lead to a substantial performance degrade. Reasons are that interceptors require some additional indirections, dynamic lookups, and instantiations and destructions of interceptors. Some Web Service frameworks use a much simpler and less flexible message processing architecture (for instance, see the discussion of GLUE below).

JAX-RPC Handlers

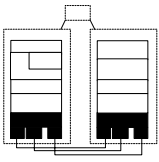
Sun's JAX-PRC API [Sun04b] supports a slightly different handler model than Axis, but it can also be used with Axis. Axis has a more flexible INVOCATION INTERCEPTOR architecture than JAX-RPC. However, over time it might get deprecated in favor of the JAX-RPC standard that provides portability across application servers. From the perspective of the pattern language both are pretty similar. Thus we do not discuss the differences in detail, only mention that JAX-RPC handlers have a slightly different interface:

```
public interface Handler {
    boolean handleRequest(MessageContext ctx);
    boolean handleResponse(MessageContext ctx);
    boolean handleFault(MessageContext ctx);
}
```

This interface can be used in an Axis chain by configuring the JAXRPCHandler. This is an *Adapter* for JAX-RPC compliant handlers that exposes an Axis handler interface. It can be configured as follows:

```
<requestFlow>
  <handler type="java:org.apache.axis.handlers.JAXRPCHandler">
    <parameter name="scope" value="session"/>
    <parameter name="className"
      value="test.MyHandler"/>
  </handler>
</requestFlow>
<responseFlow>
  <handler type="java:org.apache.axis.handlers.JAXRPCHandler">
    <parameter name="scope" value="session"/>
    <parameter name="className" value="test.MyHandler"/>
  </handler>
</responseFlow>
```

Protocol Integration



Next, let's take a closer look at the lowest layer of message processing in a Web Service framework: request handling. At this layer the heterogeneity of communication protocols of Web Service frameworks is provided. Again, most Web Service frameworks provide for some extensibility at this layer - even though slightly different REQUEST HANDLER/PROTOCOL PLUG-IN architectures are used.

In the default case that HTTP is used as a communication protocol, on the server side a Web Service framework has to behave like a Web Server. That means, the HTTP server has to be informed that specific HTTP requests (for instance, all requests on a specific port or URL extension) are handled by the Web Service framework. If such a port is accessed, the HTTP server does not handle the message itself, but forwards it to the SERVER REQUEST HANDLER of the Web Service framework. A Web Service framework can either be plugged into a server or embed a server.

The typical communication protocol used by Web Service REQUEST HANDLERS is HTTP. But SOAP also allows for other communication protocols. Such a variability of communication protocols can be implemented with PROTOCOL PLUG-INS for the protocols supported. On top of the PROTOCOL PLUG-INS, all protocols are handled in the same way as HTTP. That is, for instance, the same INVOKER can be used for all protocols. Axis supports PROTOCOL PLUG-INS for HTTP, Java Messaging Service (JMS), SMTP, and local Java invocations. PROTOCOL PLUG-INS are also responsible for implementing a MESSAGE QUEUE, if needed (as for instance implemented by JMS-based messaging).

PROTOCOL PLUG-INS need to be defined for both client side and server side. In Axis there are two main elements of the client-side PROTOCOL PLUG-INS:

- *Transport:* Axis contains a transport abstraction from which specific transport classes, such as HTTP transport, SMTP transport, JMS transport, etc. inherit. The main task of the transport is to create an INVOCATION CONTEXT that is specific for the chosen communication protocol. For instance, the HTTP message context may contain HTTP cookie options. The JMS message context contains JMS options like message "priority" and "time to live."

The transport is set up (by the CLIENT PROXY) before the handler chain is traversed. This is important because some handlers may depend on the information which transport is used. For instance, session handling can only set cookie options, if HTTP is used, otherwise the session information has to be transported in the SOAP header.

- *Sender*: There is a sender of each supported protocol, such as HTTP sender, JMS sender, SMTP sender, etc. A sender is a special handler that is used as the last handler in the client side handler chain. It actually sends a request message across the network using the sender's communication protocol (and may wait for the result in case of a blocking invocation). Thus, obviously, the sender must be specific for the transport protocol. The responsible sender class is chosen according to the transport name given in the INVOCATION CONTEXT of a request.

On server side, classes for receiving messages from a communication protocol are defined. These implement the SERVER REQUEST HANDLERS of the Web Service framework. The respective servers of the different communication protocols contain a *Reactor* [SSRB00] that reacts on network events and informs the SERVER REQUEST HANDLERS. On server side there are also PROTOCOL PLUG-INS for the different protocols supported, in particular:

- For HTTP a servlet is provided which can be used with any Java HTTP server that supports servlets (i.e. a servlet container like Tomcat).
- Axis also implements a simple stand-alone HTTP server (for testing purposes only).
- For JMS a *MessageListener* is implemented which handles JMS messages with a JMS worker class. This class uses Java's JMS implementation as the actual JMS server.
- For SMTP a simple *MailServer* is defined that connects to a POP3 mail server.

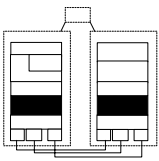
In all cases, the SERVER REQUEST HANDLER is connected to the used server, and it *observes* the network events generated by the server. The server lets the SERVER REQUEST HANDLER handle specific incoming requests. For instance, a certain port or range of URLs can be redirected to the SERVER REQUEST HANDLER. For each request that arrives, the

SERVER REQUEST HANDLER sets up the server side INVOCATION CONTEXT and invokes the server side handler chain (as described in the previous section).

Note that many typical SERVER REQUEST HANDLER tasks are handled by the respective server of the communication protocol, such as listening to the port, (de-)marshalling on the level of the communication protocol (that is, HTTP, SMTP, etc.), performance optimizations, etc. The Web Service framework's SERVER REQUEST HANDLER only performs the high-level tasks on the level of SOAP and above. Similarly, lower-level CLIENT REQUEST HANDLER tasks are performed by existing client APIs. This is quite typical for the Web Service domain because usually existing client and server implementations can be reused, because the used protocols are originally designed for other tasks than Web Services, and Web Services frameworks are implemented as an integration layer on top of these implementations.

Because of these reasons, the REQUEST HANDLERS and PROTOCOL PLUGINS of the Web Service framework are relatively generic and allow for simple exchange of the used communication protocol. A disadvantage of this architecture is that not each protocol implements every functionality of the other protocols. For instance, simple session handling is possible via HTTP cookies, but cookies are specific for HTTP and not supported by protocols such as JMS and SMTP.

Marshalling using SOAP XML Encoding



The MARSHALLER is a handler (i.e. an INVOCATION INTERCEPTOR) in the request and response flow that uses the SOAP implementation to encode messages in the SOAP XML format and decode messages from the SOAP XML format. During encoding it produces the SOAP message structure, and during decoding it extracts the information from it.

Each SOAP message has a so-called envelope. The SOAP envelope is a simple wrapper for the content. The envelope can contain an optional header that contains control information, for instance, for routing or authorization. Each SOAP message contains a SOAP body that contains the message payload. Attachments can hold other types of data, such as binary data, un-encoded text, and so on.

Consider the following simple SOAP invocation (from the SOAP interoperability suite). It specifies its used XML schemas in the envelope, and in the body it invokes an operation `echoInteger` with one integer argument that has the value “42”:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <ns1:echoInteger soapenv:encodingStyle=
      "http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:ns1="http://soapinterop.org/">
      <inputInteger xsi:type="xsd:int">42</inputInteger>
    </ns1:echoInteger>
  </soapenv:Body>
</soapenv:Envelope>
```

The SOAP server can respond to this invocation with a similar message, also referencing used XML schemas and namespaces in the envelope. In the body, a response to the operation invocation is embedded with a return type and value (again an integer with the value “42”):

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap=
  "http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:tns="http://soapinterop.org/"
  xmlns:types="http://soapinterop.org/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soap:Body soap:encodingStyle=
    "http://schemas.xmlsoap.org/soap/encoding">
    <types:echoIntegerResponse>
      <return xsi:type="xsd:int">42</return>
    </types:echoIntegerResponse>
  </soap:Body>
</soap:Envelope>
```

One can specify the encoding for the message in the SOAP body. There are two commonly used formats here:

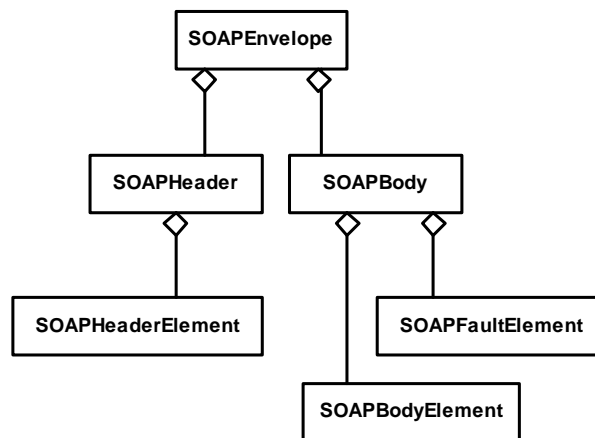
- *RPC encoding*: The SOAP specification defines the so-called RPC encoding (`http://schemas.xmlsoap.org/soap/encoding`). This encoding style has the goal to provide RPC invocations in the fashion of CORBA or Java RMI. Invocation parameters and the

result are encoded in the body. The RPC encoding defines primitive types, arrays, and structures of these types. Operations can have in/out and out parameters in addition to the return value.

- *Document literal*: An XML document with a format defined by a schema is transported with the SOAP messages. Compared to RPC encoding, document literal is more flexible, portable, and interoperable. Document literal requires code on client and server side to convert the data. .NET uses a variation of document literal with invocation parameters as children of root elements.

A SOAP message optionally contains a SOAP header. It contains add-on information, such as authentication information, session management information, or transaction management information. This information represents the INVOCATION CONTEXT of a message at the SOAP level.

The SOAP attachment specification [BTN00] deals with the problem that the character set of SOAP invocations is restricted in XML. The XML markup has to be well-formed. Encoding binary data has a significant overhead. These problems can be avoided by attachments that use MIME multipart message format. In Axis, SOAP attachments are an extensible part of the INVOCATION CONTEXT.



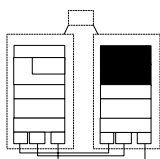
When a SOAP message is built up before serialization or after it is deserialized, the MARSHALLER creates objects conforming to the class

structure, shown in the figure above, out of the XML data. These SOAP parts are referenced by the INVOCATION CONTEXT of the message.

Before a message is serialized or after it is de-serialized, the SOAP parts are represented by respective classes, as shown in the above figure. In Axis special handlers and *Builders* are used to receive the events from the XML parser and build up these classes, when the corresponding XML elements are encountered. The building process of an element of the object tree is started, when the corresponding XML tag is encountered, and the building process of an element is finished, when the XML tag is closed.

An important part of SOAP marshalling is encoding of data types. In Axis, the MARSHALLER contains different classes that implement serializer and de-serializer interfaces. These have the task to encode programming language elements into XML and vice versa. For each supported SOAP encoding type (primitive data types, arrays, base64, Java beans, etc.), there is one serializer and one de-serializer class. Moreover, there is a type mapping table providing a mapping from a pair of programming language type and XML data type to a serializer and a de-serializer. Note that Axis associates with serializer and de-serializer *factories* to support multiple XML processing models, such as SAX and DOM. As each SOAP message can possibly specify its own encoding, there is one default type mapping table and mapping tables for other encodings (given as URIs in the SOAP message). The basic mapping between Java types and WSDL, XSD, and SOAP in Axis is determined by the JAX-RPC specification.

Lifecycle Management



Axis supports the following lifecycle management patterns using a *scope* option chosen in the deployment descriptor:

- **PER-REQUEST INSTANCE:** The HTTP protocol uses per default a simple request/response scheme of communication. That means HTTP itself is stateless. Due to the service character, the default lifecycle model of a Web Service is the PER-REQUEST INSTANCE, which is also stateless. That is, a Web Service remote object is created, is invoked once, returns a result, and then the remote

object is destroyed. In Axis, such a PER-REQUEST INSTANCE is called a service with “request” scope.

- **STATIC INSTANCE:** Axis also supports STATIC INSTANCES, when the so-called “application” scope is chosen. This scope will create a remote object as a shared object that handles all requests for a particular Web Service.
- **CLIENT-DEPENDENT INSTANCE:** Sessions are supported either by HTTP cookies or by - communication protocol independent - SOAP headers. If a service has “session” scope it will create a new object for each session-enabled client that accesses a service. With the session abstraction, CLIENT-DEPENDENT INSTANCES can be implemented for a service. Note that - as a second pattern - LEASES is required here to implement sessions. Each session object has a timeout (which can be set to a certain amount of milliseconds). After the timeout expires, the session is invalidated. A method touch can be invoked on the session object, which re-news the LEASE.

The standard Axis provider that exposes Java classes as services does not support POOLING for PER-REQUEST INSTANCES or CLIENT-DEPENDENT INSTANCES, though it would be no problem to add such a functionality here. Other providers are implemented as subclasses of this class. These subclasses have to override the *Factory Method* [GHJV95] `makeNewServiceObject` that creates the Java PER-REQUEST INSTANCES. Thus the POOLING functionalities of component frameworks such as EJB, for instance, can be used.

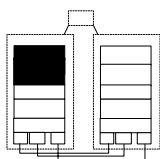
Note that Web servers and other communication protocol implementations usually pool the connections and/or worker threads that handle the HTTP requests.

The basic LIFECYCLE MANAGER of Axis is very simple. For services invoked with the standard Java provider, a `ServiceLifecycle` interface is implemented that just has the operations `init` and `destroy`. This interface is invoked by the Java provider when the service objects are created and destructed. The session implementation extends the lifecycle management to handle the current session state of an object and the LEASES as well.

Possibly other providers might shield frameworks that contain their own implementations of a LIFECYCLE MANAGER, POOLING, etc. For

instance, the `EJBProvider` shields the EJB component container that contains a `LIFECYCLE MANAGER` and a solution for `POOLING`. In other words, not all aspects of lifecycle management do necessarily have to be handled within the Web Service framework - instead these can be provided by the service implementations.

Client-Side Asynchrony



We have already provided two examples for client-side invocations, one building up the invocation at runtime with a generic `CLIENT PROXY`, and the second one using a `CLIENT PROXY` generated with WSDL. Both examples implement synchronous invocations. Before we discuss alternatives for asynchronous invocation, we briefly revisit the synchronous variants of Web Service invocation.

Asynchrony Supported by Axis and WSIF

The `Call` interface (as of Axis version 1.0) does only support synchronous request-response invocations and one-way invocations. The same options are also supported by other Web Service invocation frameworks for Java, such the Web Service Invocation Framework (WSIF) [Apa01] - a framework-independent invocation interface.

One-ways are specified in the WSDL specification and thus are supported by many Web Service frameworks. The one-way invocations follow the `FIRE AND FORGET` pattern. Both Axis and WSIF do not yet support asynchronous invocations on client side over synchronous protocols such as HTTP. That is, the other asynchrony patterns, `SYNC WITH SERVER`, `RESULT CALLBACK`, and `POLL OBJECT`, are not natively supported by Axis or WSIF.

But both, Axis and WSIF, have `PROTOCOL PLUG-INS` for asynchronous messaging protocols, such as Java Messaging Service (JMS) protocol. Thus support for asynchrony is provided by a `PROTOCOL PLUG-IN` (for JMS) rather than implementing the asynchrony patterns on top of HTTP. This capability can be used to implement `RESULT CALLBACKS` with JMS messaging as the underlying protocol.

The general client programming model for asynchrony via a messaging protocol such as JMS is that the client sends an asynchronous remote invocation and stores a correlation identifier (the term used for an

Asynchronous Completion Token in the messaging domain) together with the ID of a callback operation locally. Then it proceeds with its work. The Web Service receives this invocation and sends a response message back, after processing the message. The client acts also as a messaging server to be able to receive responses. In JMS a `JMSListener` within the client waits for messages. When a message arrives, the `JMSListener` uses the correlation identifier, sent back with the response message, to look up the associated callback operation, and invokes it.

Even though this scheme follows the `RESULT CALLBACK` pattern, there might be some problems involved. First, messaging protocols deal with a lot of additional aspects to support the QoS property “reliability of message transfer.” In cases where reliable message transport is not required, a messaging protocol is an overhead. Also, if HTTP should be used the messaging protocol might be problematic. In such cases we can implement the asynchrony patterns on top of synchronous invocations - this is explained in the next section.

Building the Asynchrony Patterns on Top of Axis

In this section, we explain how to implement the client-side asynchrony patterns in a generic and efficient way on top of Axis. To reach this goal we implement an asynchrony *Layer* on top of synchronous invocation *Layer* provided by Axis.

We describe the Simple Asynchronous Invocation Framework for Web Services (SAIWS) [ZVK03] here, an add-on for Apache Axis. The framework can be downloaded from <http://saiws.sourceforge.net>. An interesting aspect of this discussion is that this support for client side asynchrony was designed - using the pattern language presented in this book - on top of a given distributed object middleware, and not at the lower-level layers of the distributed object middleware as for instance in the CORBA technology projections.

In this framework, we provide two kinds of `REQUESTOR`s, one for synchronous invocations and one for asynchronous invocations. Both use the same invocation scheme. The synchronous `REQUESTOR` blocks the client until the response returns. A client can invoke a synchronous `REQUESTOR` as follows:

```
SyncRequestor sr = new SyncRequestor();
String result =
```

```
(String) sr.invoke(endpointURL, operationName, null, rt);
```

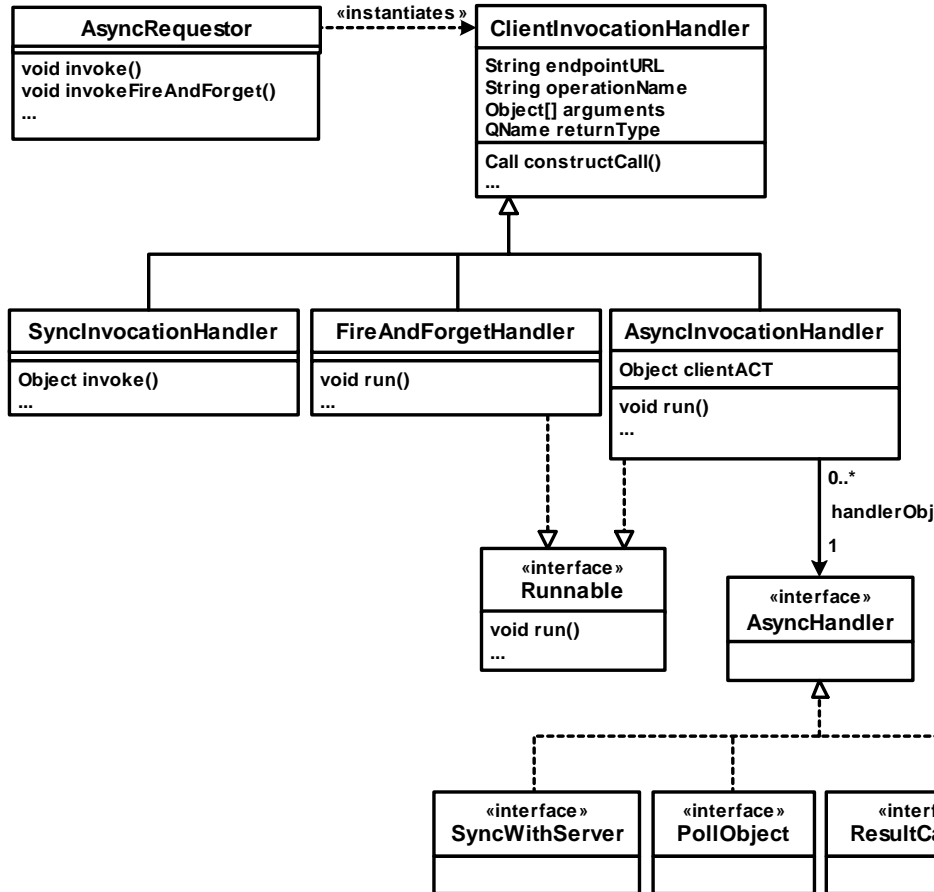
This REQUESTOR simply handles the invocation using the Axis Call interface presented before, and after it has returned, it returns to the client.

The asynchronous REQUESTOR is used in a similar way. It offers invocation methods that implement the client asynchrony patterns. The general structure of asynchronous invocation is quite similar to synchronous invocations. The only difference is that we pass an AsyncHandler and clientACT as arguments to the invoke operation and do not wait for a result:

```
AsyncHandler ah = ...;
Object clientACT = ...;
AsyncRequestor ar = new AsyncRequestor();
ar.invoke(ah, clientACT, endpointURL, operationName, null, rt);
// ... resume work
```

Note that the clientACT field is used here as a pure client side implementation of an *Asynchronous Completion Token* (ACT) [SSRB00]. The

ACT pattern is used to let clients identify different results of asynchronous invocations when the response has arrived.



The synchronous invocation handler mainly provides a method `invoke` that synchronously invokes a service and blocks. The asynchronous invocation handlers implement the `Runnable` interface. This interface indicates that the handler implements a variant of the *Command* pattern [GHJV95] that can be invoked in the handler's thread of control. Three `AsyncHandler` interfaces are provided for the different kinds of asynchronous invocations following the patterns SYNC WITH SERVER, POLL OBJECT, and RESULT CALLBACK.

The `AsyncInvocationHandler` decides on basis of the kind of handler object which asynchrony pattern should be used, `RESULT CALLBACK`, `POLL OBJECT`, or `SYNC WITH SERVER`.

The `FIRE AND FORGET` pattern is not implemented in the class `AsyncInvocationHandler` (or as a subclass of it) due to a specialty of Web Services: the WSDL standard that is used for `INTERFACE DESCRIPTION` of Web Services supports so-called one-way operations. These are thus implemented by most Web Service frameworks that support WSDL. Therefore, we do not implement `FIRE AND FORGET` with the `AsyncInvocationHandler` class, but use the one-way invocations to support `FIRE AND FORGET` operations. All other invocations, dispatched by the `AsyncInvocationHandler` class, are request-response invocations.

A `FIRE AND FORGET` invocation executes in its own thread of control. The `FIRE AND FORGET` invocation simply constructs the invocation, sends it, and then the thread terminates. A `FIRE AND FORGET` invocation is invoked by a special `invokeFireAndForget` method of the `AsyncClientProxy` class:

```
AsyncRequestor requestor = new AsyncRequestor();
requestor.invokeFireAndForget(endpointURL, operationName,
    null, rt);
```

To deal with the asynchrony patterns `RESULT CALLBACK`, `POLL OBJECT`, or `SYNC WITH SERVER` the client asynchrony handler types `ResultCallback`, `PollObject`, and `SyncWithServer` are provided. These are instantiated by the client and handed over to the `REQUESTOR`.

The asynchronous `REQUESTOR` handles the invocation with an `AsyncInvocationHandler`. Each invocation handler runs in its own thread of control and deals with one invocation. A thread pool is used to improve performance and reduce the resource consumption. The client asynchrony handlers are sinks that are responsible for holding or handling the result for clients.

For an asynchronous invocation, the client simply has to instantiate the required client asynchrony handler (that is, a class implementing a subtype of `AsyncHandler`) and provide it to the `REQUESTOR`'S operation `invoke`. This operation is defined as follows:

```
public void invoke(AsyncHandler handler, Object clientACT,
    String endpointURL, String operationName,
    Object[] arguments, QName returnType)
    throws InterruptedException {...}
```

The parameter handler determines the responsible handler object and type. It can be of any subtype of `AsyncHandler`. `clientACT` is a user-defined identifier for the invocation. The client can use the `clientACT` to correlate a specific result to an invocation. The four last arguments specify the service ID, operation name, and invocation data.

Consider a `POLL OBJECT` as an example use of an asynchronous invocation. The client might invoke a `POLL OBJECT` by first instantiating a corresponding handler and then providing this handler to the `invoke` operation. Subsequently, it polls the `POLL OBJECT` for the result and works on some other tasks until the result arrives:

```
AsyncRequestor requestor = new AsyncRequestor();
PollObject p = (PollObject) new SimplePollObject();
requestor.invoke(p, null, endpointURL, operationName, null, rt);
while (!p.resultArrived()) {
    // do some other task ...
}
System.out.println("Poll Object Result Arrived = " +
    p.getResult());
```

The class `SimplePollObject` implements the `PollObject` interface. Here the `clientACT` parameter is set to `null` because we can use the object reference in `p` to obtain the correct `POLL OBJECT`. More complex `AsyncHandlers` that handle multiple responses use the `clientACT` to identify the request belonging to a response.

The patterns `RESULT CALLBACK` and `SYNC WITH SERVER` are used in the same way, but the respective `AsyncHandler` has to be used. For instance, the following code instantiates a `RESULT CALLBACK` and then sends ten invocations, each handled by the same `RESULT CALLBACK` object. Each invocation gets an identifier as `clientACT` so that the `RESULT CALLBACK` is able to identify the results, when they arrive:

```
AsyncRequestor requestor = new AsyncRequestor();
ResultCallback r = (ResultCallback) new ResultCallbackQueue();
for (int i = 0; i < 10; i++) {
    String id = "invocation" + i;
    requestor.invoke(r, id, endpointURL, operationName,
        null, rt);
}
```

Web Services and QoS



Even though there are quite a few efforts and emerging standards around Quality of Service (QoS) of Web Services, currently QoS measures for Web Services can rather be seen as a set of best practices. We want to summarize these here for the QoS properties performance, accessibility, availability, reliability, and integrity.

Performance defines the number of requests for remote objects that are served in given period of time. Performance is influenced by client and server processing times and network latency. Web Service frameworks contain a few typical performance bottlenecks compared to other distributed object middleware, because XML processing requires various levels of XML parsing and XML validation. Unfortunately, not all Web Service frameworks apply best practices to maximize performance. These include the use of efficient XML parsers, avoiding XML validation in production mode, sending compressed XML over the network (e.g. by eliminating unnecessary white spaces, or using binary representations), and caching of Web Service results. The Web Service developer can improve the performance by using simple data types for SOAP interactions. There are quite a few tools to monitor the performance of Web Servers, mostly to debug web site or interactive web applications. Many of these tools can be used to implement a QOS OBSERVER for Web Services running in a server as well. An approach to implement a QOS OBSERVER specifically for Web Services is described in [Jen02].

Accessibility defines the capability of a remote object to serve the clients' requests. This QoS property can be enhanced either by stronger hardware, highly concurrent systems, or by highly scalable systems. Best practices to enhance accessibility are:

- POOLING of services can be implemented by the service provider. In Axis POOLING is not supported by the Web Service framework. Pooled objects should use *Thread-Specific Storage* [SSRB00] to reduce the risks of lock contention and racing conditions.
- *Thread Pooling* [KJ04] for the server's worker threads is provided by most servers of communication protocols used for Web Services. Specifically, modern Web Servers provide highly efficient *Pooling* architectures. Pooled workers usually can operate on their private data set and thus support *Thread-Specific Storage* as well.

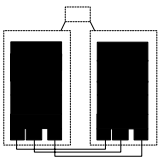
- Load balancing measures can improve scalability. Specifically, for Web servers, load balancing across multiple machines can be exploited. A root server only checks the URL for the responsible server. It then transparently redirects the requests (e.g. using DNS aliasing, “magic” routing, or HTTP Redirect) to the server that is actually responsible for handling the request. Redirection can be implemented as a LOCATION FORWARDER.

Availability defines whether a remote object is ready for servicing requests at a time, or not. This QoS property is not tackled by Web Service frameworks directly. Web servers usually allow one to manage and log the server, what allows for a weak control of availability (that is, one can check the times of unavailability and long response times). Availability can be improved by fault-tolerant systems or clustering.

Reliability defines the numbers of failing requests in a time, as well as the ordered delivery of messages. HTTP as the standard communication protocol of Web Services provides only best effort delivery. Many Web Service frameworks allow for the use of messaging protocols to guarantee message delivery and message order.

Integrity can refer to integrity of data and transaction integrity. Data integrity defines whether the transferred data is modified during transmission. Transactional integrity refers to the orchestration of a transaction that is longer than a single request. Both kinds of integrity are not handled in current Web Service frameworks. For transaction integrity and enduring business transaction there are a number of Web Service standards, such as BPEL4WS [ACD+03], WS-Coordination [CCC+03], and WS-Transaction [CCC+02].

Web Service Security

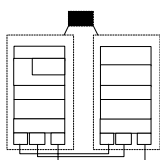


Among other things, security of remote invocations includes authenticating clients, encrypting of messages, and access control. In Web Service frameworks usually some means for authorization are provided via the Basic or Digest Access Control mechanism of the HTTP protocol (in Axis supported as a special handler). SSL encryption of the network traffic can be supported using HTTPS as a PROTOCOL PLUG-IN. Currently, different XML standards emerge for dealing with security at a broader basis. These are, however, not widely adopted yet

by most Web Service frameworks. Examples are the XML encryption specification [ERI+02], the XML key management specification [FHF+02], and XML Signature [BBF+02]. It is quite typical that the security is supported either at the communication protocol level by a PROTOCOL PLUG-IN or as an extension of message processing activities by an INVOCATION INTERCEPTOR. More complex security requirements can be coded in the Web Services themselves.

The simple security measures, summarized above, work for simple interactions; however, in larger service-oriented architectures there are usually additional security requirements. For instance, problems arise, if messages have to be routed across (untrusted) intermediaries, if more than one party needs control over security aspects, or if the messages need to be securely stored for later consumption. A number of newer security standards for Web Services are proposed to deal with these issues. WS-Security [ADH+02] is the premier standard for more complex authentication, confidentiality, and integrity of SOAP messages. It provides a framework for the exchange of X.509 certificates or SAML tokens. SAML [Oas03a] is an XML-based security protocol that provides the secure exchange of authentication and authorization assertions. WS-Trust [DDG+02] is another proposed standard for exchanging authentication and authorization assertions, but it is not as mature as SAML yet.

Security in a service-oriented architecture with a potentially larger number of participants requires a means to find out about the security policies supported by a partner without human intervention. XACML [Oas03a] allows developers to define access control policies. Web Service end-points can be described with the mandatory features of service invocation, optional features that they support, and a preference order amongst those features. WS-Policy [BCH+03] provides a framework for defining properties of Web Services as policy statements. Its security addendum WS-SecurityPolicy [DHH+02] can be used to describe the security properties of a Web Service.



Lookup of Web Services: UDDI

There are many possible ways to realize LOOKUP with Web Services; especially there are a number of proprietary implementations. UDDI [Oas02] is a standard for an automated directory service that allows

one to register and lookup services. In contrast to many simple naming and property services, UDDI started with the vision of providing a world-wide, replicated registry for business services, similar to the Domain Name Service (DNS) for Internet domain names. A central registry server is provided by a number of vendors, including Microsoft and IBM, called the Universal Business Registry (UBR).

All UDDI specifications use XML to define data structures. An UDDI registry includes four types of documents which make up the simple basic data model of UDDI:

- A *business entity* is an UDDI record for a service provider.
- A *business service* represents one or more deployed Web Services.
- A *technical model* (tModel) can be associated with business entities or services.
- A *binding template* binds the access point of a Web Service and its tModel.

UDDI allows a service provider to register information about itself and the services it provides, such as the service provider's name, contact information, and description. There can be arbitrary contact information, including URLs that refer to a Web Service. For a Web Service technical details, such as the supported protocol (e.g. HTTP or SMTP) are also registered. The most common Web Service protocols, such as HTTP and SMTP, are pre-registered in UDDI registries as so-called tModels. tModels are templates for technical protocol specifications in UDDI. A client can search for services that have an associated tModel complying with one of the client's protocols.

This simple model of UDDI alone is too simple to describe all information that are relevant for a business or a service. Instead of defining one highly complex model for all systems, UDDI supports predefined and user-defined taxonomies for the identification and categorization of the information. The taxonomies are themselves tModels - that is, they can be categorized themselves.

An important aspect is the integration with the INTERFACE DESCRIPTION format WSDL. The WSDL port type and binding are mapped to tModel elements (using categories predefined in UDDI). The role of the port element in WSDL is provided by the binding template: it associates an abstract communication scheme with a concrete communication

protocol. The WSDL service element obviously matches the business service in UDDI.

UDDI was originally designed according to the business vision from the early days of Web Services. The idea was to provide services publicly and let - perhaps until then unknown - business partners come together, for instance by finding an offered Web Service via UDDI. This vision has not yet gained much momentum, mainly because many businesses do not work this way. In most businesses, the first business transaction is preceded by a number of selection steps - instead of letting your business partner be found automatically e.g. via UDDI.

UDDI version 3 supports a number of improvements to make UDDI more practically usable. Technical improvements are an improved API for accessing a UDDI registry, a better performance, and ways to inherit from taxonomies. Conceptual improvements are the support for multiple, federated UDDI registries, instead of the central UBR, as well as ways to exchange and synchronize data in different UDDI instances.

Other Web Service Frameworks

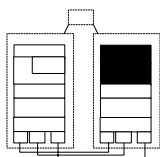
In this section we discuss some other Web Service frameworks. We present webmethods's (formerly: The Mind Electric) GLUE, Microsoft's .NET Web Services, and IONA's Artix as popular commercial Web Service implementation. We also discuss the framework Leela that extends Web Service frameworks with concepts known from P2P architectures, coordination technologies, and spontaneous networking.

GLUE

GLUE [Min03] is a commercial Web Service implementation in Java that is optimized for ease-of-use and performance. It can be run as a stand-alone application server or in other application servers.

Similarly to Axis, any Java class can be published as a Web Service. But in contrast to Axis, a programmatic deployment is primarily used instead of the XML deployment descriptor used by Axis. For instance, to publish a Web Service of the class HelloWorld:

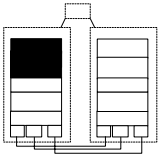
```
public class HelloWorld implements IHelloWorld {
    ...
}
```



one has to publish this class in a registry:

```
Registry.publish("helloworld", new HelloWorld());
```

The registry automatically generates a WSDL INTERFACE DESCRIPTION for the Web Service that is remotely accessible. By default all public methods (and interfaces) of a remote object are accessible. Instead of publishing all methods, one can specify only specific interfaces to be published. Even single methods can be published programmatically.



The registry is also provided on client side. Here, it is used to bind to a Web Service remote object. This is done by retrieving the WSDL INTERFACE DESCRIPTION of the Web Service. Then a CLIENT PROXY is generated from this INTERFACE DESCRIPTION. Finally, the CLIENT PROXY is casted to one of the Web Service's interfaces to allow for invocation of its methods. This can be done programmatically as well:

```
String url = "http://localhost:8015/glue/helloworld.wsdl";
IHelloWorld hw = (IHelloWorld)
    Registry.bind(url, IHelloWorld.class);
```

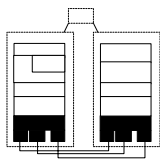
Instead of generating the CLIENT PROXY dynamically, a helper class can be generated by the tool "wsdl2java". This helper class is then stored on disk and compiled. This is not as flexible as the dynamic retrieval of the WSDL file above, but avoids the overhead of retrieving and processing the WSDL file at runtime. One can then bind to the helper, instead of using the registry:

```
IHelloWorld hw = HelloWorldHelper.bind();
```

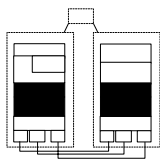
Generating the CLIENT PROXY is the default in GLUE to facilitate ease of use: the REQUESTOR is mostly hidden from the developer. However, sometimes we need to construct invocations dynamically, for instance, because we do not know an interface of a remote object before runtime. GLUE allows developers to access the REQUESTOR using the interface IProxy. But again, WSDL is used to bind to the Web Service - thus generally GLUE uses the automatically generated INTERFACE DESCRIPTION to ease the use of Web Services. When using the Axis REQUESTOR, in contrast, we have to deal with low-level details like service endpoints. Here is an example for using the GLUE REQUESTOR:

```
IProxy proxy = Registry.bind(
    "http://localhost:8015/glue/helloworld.wsdl" );
String result = (String) proxy.invoke("hello", null);
```

The method `invoke` receives the method name and an array of arguments. An interesting feature is that GLUE converts string-based arguments to the correct types in the WSDL INTERFACE DESCRIPTION automatically.



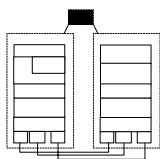
GLUE supports PROTOCOL PLUG-INS for HTTP, HTTP over SSL, and JMS. JMS has to be used to support client asynchrony. These PROTOCOL PLUG-INS are realized by instantiating one or more SERVER REQUEST HANDLER classes. The `Servers` class, the common superclass of these classes, contains static methods for manipulating a number of SERVER REQUEST HANDLER objects. This way multiple PROTOCOL PLUG-INS can be co-existing in one application.



The client side INVOCATION CONTEXT is provided by the `ProxyContext` class; the server side INVOCATION CONTEXT is provided by the `ServiceContext` class. These classes contain all basic invocation information, such as endpoint, HTTP information (like HTTP proxy, proxy password, etc.), XML namespace, XML encoding, etc., as well as authentication information and other add-on context information.

GLUE supports only so-called `SOAPInterceptors`, INVOCATION INTERCEPTORS on the level of SOAP message handling. A remote object can specifically throw a REMOTING ERROR by raising a SOAP exception in one of its methods. An alternative is to use INVOCATION INTERCEPTORS to set the REMOTING ERROR properties.

Glue supports three activation modes. `STATIC INSTANCES` are used per default. A Web Service can be configured as a `PER-REQUEST INSTANCE` or `CLIENT-DEPENDENT INSTANCE` (using a session abstraction).



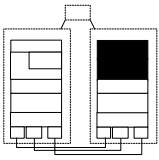
Providing automatically generated WSDL files is a very simple variant of LOOKUP. By only knowing the remote location of WSDL file, one can lookup a Web Service and its supported protocols. This mechanism can also be used as a simple variant of LOCATION FORWARDER: GLUE allows to override the endpoint in the context object of a Web Service. This can be used to route a Web Service invocation through a HTTP Proxy or Firewall.

UDDI is supported as a sophisticated LOOKUP variant.

In summary, in GLUE many design decisions and pattern variants are similar to Axis. However, GLUE provides more ease of use by providing a more simple invocation model. This is reached by automat-

ically generating an INTERFACE DESCRIPTION and remotely providing it as a very simple form of LOOKUP. Also, GLUE offers a high performance because it uses a simple invocation architecture without many indirections or other overhead. The REQUEST HANDLERS, PROTOCOL PLUG-INS, and the INVOKER are generally configured in a simple and performant way. Axis offers a more generic and more flexibly extensible architecture. GLUE uses per default a programmatic deployment model. The XML-based deployment model of Axis also is more complex and there is an overhead for processing the XML files.

Microsoft's .NET Web Services



The .NET Framework offers two mechanisms for distributed application development and integration: .NET Remoting and ASP.NET Web Services. We have already looked at .NET Remoting in much details. So let's first consider a simple Hello World Web Service example in ASP.NET first:

```
<%@ WebService Language="C#" Class="RemotingBook.HelloWorld" %>
namespace RemotingBook {
    using System;
    using System.Web.Services;
    public class HelloWorld : WebService {
        [WebMethod]
        public string hello() {
            return "Hello World!";
        }
    }
}
```

The above code is deployed by storing it in a file and copying it into a directory served by Microsoft's IIS Web Server. That is, the ASP.NET approach has also a runtime deployment model, similar as Axis. But, in contrast to Axis which uses an XML-based deployment descriptor, it stores the deployment information within the program code of the Web Service. Note the first line of the code contains a `WebService` directive, an ASP.NET statement declaring that the code that follows is a Web Service. The inline code `[WebMethod]` corresponds to declaring the allowed methods of a Web Service in Axis; that is, those operations which are dispatched by the INVOKER as a Web Service. Another typical characteristic of Web Services is that the actual service providers can vary: in ASP.NET different programming languages (supported by .NET) can be used to implement the service.

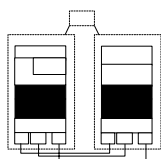
Even though the concepts are similar, we can see - as a severe difference to Axis where we have used an ordinary Java class as a remote object - that an ASP.NET Web Service has to be specifically prepared by making the service class inherit from the class `WebService` and by adding inline code such as the `[WebMethod]` statement.

ASP.NET Web Services rely on the `System.Xml.Serialization.XmlSerializer` class as a MARSHALLER for marshalling data to and from SOAP messages at runtime. They can generate WSDL descriptions containing XSD type definitions. As ASP.NET is relying on Web Service standards, ASP.NET Web Services can be used to interoperate with other (non-Microsoft) Web Service platforms. That also means that you can only marshal types that can be expressed in XSD - in other words .NET-specific types cannot be exposed as interoperable ASP.NET Web Services.



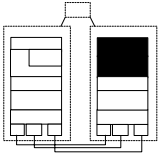
To a certain extent .NET Remoting can also be used to expose Web Services by choosing the SOAP formatter and the HTTP channel of .NET Remoting. But there are two problems. Firstly, the generated WSDL files always describe messages in terms of the SOAP-encoding rules instead of literal XSD. Secondly, the generated WSDL files (might) include extensions that are specific for .NET Remoting. This is legal in WSDL, as it supports extensions, but other Web Service frameworks (including ASP.NET) have to ignore these types. This is problematic, as some .NET data types (such as the .NET type `DataSet` for instance) have a special meaning. If you want interoperability with other Web Service frameworks, you need to restrict parameters to the built-in simple types and your own data types, but should not expose .NET data types.

IONA's Artix



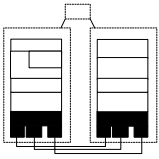
Artix is a Web Service framework implemented in C++ that focusses on high performance and middleware integration. The framework is built on the Adaptive Runtime Technology (ART) [Vin02b], an efficient and scalable INVOCATION INTERCEPTOR architecture that is also the foundation of IONA's Orbix CORBA implementation. Artix extends the basic concept of Web Services to include transparent conversions between different data encoding schemas and/or transport protocols. Besides the C++ implementation, a Java solution is offered. The Java solution is

not a native Java implementation, but a wrapper around the C++ libraries.

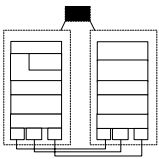


Developing simple Web Services in Artix is pretty similar to other Web Service frameworks. The developer of a Web Service needs to write a WSDL INTERFACE DESCRIPTION file. From this file, code for the client and server side is generated by a tool (a command line tool and a GUI tool are provided). Developers of Web Services need to add the method bodies of the Web Service implementation; developers of clients need to add invocation code, such as the input parameters of an invocation.

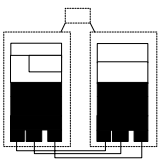
A REMOTING ERROR can be caused by the Artix runtime library, by an Artix service (such as the locator described below), or by user-defined code. The client code needs to catch the C++ exception and handle it. New REMOTING ERRORS can be specified in WSDL as WSDL faults. The WSDL generation tool generates an exception class for each fault. Optionally, WSDL code can be specified that causes a class containing user-defined details of the REMOTING ERROR to be generated.



A key difference of Artix to other Web Service frameworks is that it supports integration with other middleware without requiring the developer to write complex wrapper code or other integration code. Artix supports a number of PROTOCOL PLUG-INS, namely HTTP, BEA Tuxedo, IBM WebSphere MQ (formerly MQSeries), TIBCO Rendezvous, IIOP, IIOP Tunnel, and Java Messaging Service (JMS). In addition to that, Artix can automatically transform between the different payload formats, including SOAP, CORBA's GIOP, Tuxedo's FML, and many others. For these protocols and formats, Artix supports transparent transformations. These are performed using one-to-one converters to support high-performance message transformations.



The Artix locator provides LOOKUP for Artix services by providing a repository of endpoint references. The locator implements also a LOCATION FORWARDER, used for load balancing: if multiple ports are registered for a single service name, the locator load balances over the service's ports using a round-robin algorithm. Artix's locator is a remote object that exposes its interface using a WSDL INTERFACE DESCRIPTION.



Artix implements a number of other orthogonal services for Web Services, including security, sessions, transactions, and others. These are provided as ART plug-ins, INVOCATION INTERCEPTORS extending

the CLIENT REQUEST HANDLER and SERVER REQUEST HANDLER. Configuration of these services and other properties are transported using an INVOCATION CONTEXT, called the message attributes. The INVOCATION CONTEXT is extensible with key/value pairs.

For instance, the Artix session manager is realized by a number of ART plug-ins providing control over the number of concurrent clients. Services are registered as part of a CONFIGURATION GROUP, and sessions are handed out for the group. The session manager also controls how long each client can use the services in the group using the pattern LEASING. Clients, who are going to use a session for a longer period than the duration the session was granted, need to renew the session's lease in time.

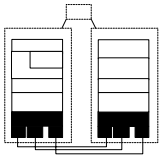
Leela: Loosely Coupled Web Service Federations

Leela [Zdu04b, Zdu04c] is an infrastructure for loosely coupled services that is built on top of a Web Service infrastructure. In addition to a Web Service infrastructure, as also provided by the Web Service frameworks discussed before, Leela implements higher-level concepts from the areas of P2P systems [CV02], coordination and cooperation technologies [GCCC85, CTV+98], and spontaneous networking [Jin03].

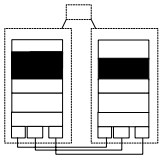
Leela integrates and extends these concepts by providing a *federated* remote object architecture. Each federation controls its peers. Peers cannot be accessed from outside of the federation without a permission of the federation. Within a federation, each peer offers Web Services (and possibly other kinds of services) and can connect spontaneously to other peers (and to the federation). Each remote object can potentially be part of more than one federation as a peer, and each peer decides which services it provides to which federation. Certain peers in a federation can be able to access extra services that are not offered to other peers in this federation via its other federations. To allow peers to connect to a federation, the federation itself must be accessible remotely. Thus the federation itself is a special peer. Peers can be dynamically added and removed to a federation.

Leela is implemented using the patterns presented in this book. Its goal is to provide a powerful remoting infrastructure which is extremely flexible and easy to use. Currently Leela is implemented in the object-oriented Tcl variant XOTcl [NZ00], but an equally powerful and easy-

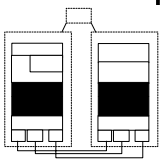
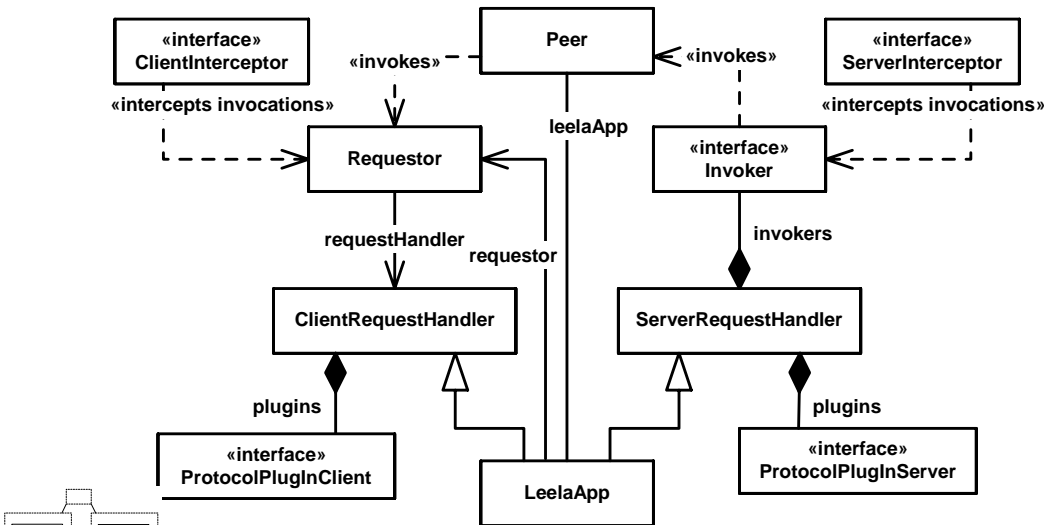
to-use implementation in Java is also under development. For the Java port, the pattern-based design of the Leela framework helps to reuse large parts of the design presented briefly in this section.



As its basic communication resource, each Leela application uses two classes implementing a CLIENT REQUEST HANDLER and a SERVER REQUEST HANDLER. Each Leela application acts as client and server application at the same time. The REQUEST HANDLERS contain PROTOCOL PLUG-INS for different protocols that actually transport the message across the network. Currently, Leela supports PROTOCOL PLUG-INS for different SOAP implementations. However, virtually any other communication protocol can be used as well because Leela's MARSHALLER uses a simple string-based format as a message payload and (re-)uses Tcl's automatic type converter to convert the string representation to native types, and vice versa.

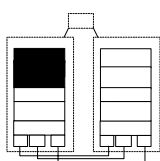


Remote invocations are abstracted by the patterns REQUESTOR and INVOKER. Leela also supports peer and federation proxies that act as CLIENT PROXIES, offering the interfaces of a remote peer or federation. The following figure illustrates the basic architecture:

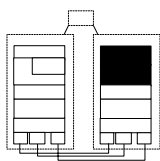


The Leela invocation chain on client side and server side is based on INVOCATION INTERCEPTORS. That is, the invocation on both sides can be

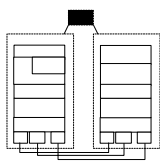
extended transparently with new behavior. The most prominent task of the INVOCATION INTERCEPTORS in Leela is control of remote federation access. On client side, an INVOCATION INTERCEPTOR intercepts the construction of the remote invocation and adds all federation information for a peer into the INVOCATION CONTEXT. On server side this information is read out again by another INVOCATION INTERCEPTOR. If the remote peer is not allowed to access the invoked peer, the INVOCATION INTERCEPTOR stops the invocation and sends a REMOTING ERROR to the client. Otherwise access is granted.



On client side, different styles of asynchronous invocations and result handling are supported, following the patterns FIRE AND FORGET, POLL OBJECT, and RESULT CALLBACK. Because in Leela each client is also a server, synchronous invocations – that let the client process block for the result – are not an option: if the Leela application blocks, it cannot service incoming requests anymore. Instead, Leela implements the asynchronous invocation patterns with a common callback model. The REQUEST HANDLERS work using an event loop that queues up incoming and outgoing requests in a MESSAGE QUEUE - both on client and server side. Invocations and handling of results are performed in separate threads. Events in these threads, like “result is ready for sending”, are placed into the event loop, serving as a simple synchronization mechanism - also used for handling events of multiple PROTOCOL PLUG-INS.



On server side the remote object is registered to be used with a particular INVOKER. There is one INVOKER per lifecycle management pattern: one for STATIC INSTANCES (using synchronized access), PER-REQUEST INSTANCES (using POOLING), and CLIENT-DEPENDENT INSTANCES. The CLIENT-DEPENDENT INSTANCES are created using a special *factory* operation of the federation peer and managed using LEASING. The pattern LIFECYCLE MANAGER is thus implemented by all these components of the Leela framework.



A semantic lookup service allows for finding peers using metadata, exposed by the peers according to some ontology. Thus it enables loosely coupled services and simple self-adaptations, for instance, to cope with interface or version changes. Peers can perform lookups in all lookup services of their federations. Leela uses the Resource Description Framework (RDF) [W3C04] to describe the peers. RDF supports semantic metadata about Web resources described in some ontology or schema. The federation provides metadata about all its

peers, such as a list of ABSOLUTE OBJECT REFERENCES, supported protocols, and OBJECT IDS (the service names). Each peer adds information for its exported methods, their interfaces, and their activation strategy. This information can be seen as a dynamic and remotely introspective INTERFACE DESCRIPTION together with location information and semantic metadata. Users can define their own, custom ontologies to provide semantic domain knowledge for peers.

Consequences of the Pattern Variants Used in Web Services

Now that we have discussed a few other Web Service frameworks as well, let us briefly revisit the consequences of applying our patterns for implementing Web Services. We have seen that there are some differences in the different Web Service implementations; yet these basic consequences are essentially the same.

First, let us summarize some characteristics of the pattern variants used in Web Service frameworks. The client side invocation architecture is quite flexible, and involves dynamic invocations using a REQUESTOR as well as runtime CLIENT PROXY generation from a WSDL INTERFACE DESCRIPTION. The INVOKER can also be (re-)configured at runtime using different means, such as XML configuration file, scripting, meta-information, or programmatic APIs. At least for the supported Web protocols, URLs are used as ABSOLUTE OBJECT REFERENCES, which is a simple but not ideal format because they do not stay stable over time. An advantage is that it is quite simple to implement a LOCATION FORWARDER for URLs (what even might be supported already, for instance, by HTTP Redirect). As Web Services are mostly an integration architecture, large parts of the lower-level patterns, such as REQUEST HANDLERS or PROTOCOL PLUG-INS, are not implemented by the Web Service framework itself, but by reusing existing implementations. The MARSHALLER might support some kinds of automatic type conversion, for instance, to and from strings (or other generic types).

Note that many of these properties are similar to inherent language properties of scripting languages. For instance, using WSDL for dynamic CLIENT PROXY generation is similar to introspection and runtime class generation in scripting languages. Automatic type

conversion is language supported in most scripting languages. Actually, scripting is used in many integration scenarios in other middleware as well. Steve Vinoski calls such scripting code the “dark matter of middleware” [Vin02a]. Web Services can be seen as standardized form to implement some integration tasks typically performed using the scripting approach in other middleware.

A central advantage of Web Services is that they provide a means for interoperability in a heterogeneous environment. They are also relatively easy to use and understand due to simple APIs. Because Web Services allow for integration of different other middleware and component platforms as PROTOCOL PLUG-INS or backends, Web Services can be used for integration of different platforms. Web Services offer standards for process flow orchestration, long business transactions, and other higher-level tasks - thus they can well be used in enterprise application integration (EAI) scenarios, where we are typically faced with a number of different platforms.

In the spirit of the original design ideas of XML [BPS98] and XML-RPC [Win99] as the predecessor of today’s standard Web Service message format SOAP, XML encoding was expected to enable simplicity and understandability as central advantages. However, today’s XML-based formats used in Web Service frameworks, such as XML Namespaces, XML Schema, SOAP, and WSDL, are quite complex and thus not very easy to read and understand (by humans).

XML as a (string-based) transport format is bloated compared to more condensed (binary) transport formats. This results in larger messages, as well as a more extensive use of network bandwidth. XML consists of strings for identifiers, attributes, and data elements. String parsing is more expensive in terms of processing power than parsing binary data.

In many cases, stateless communication as imposed by HTTP and SOAP causes some overheads because it may result in repeated transmission of the same data (for instance, for authentication or identifying the current session).

Unfortunately, the lookup protocol UDDI is - at the moment - rather viewed negatively. For example, there is no standardization of the tModels going on, making it difficult to find like services. Currently there is no real integration of WSDL and UDDI functionalities and no automated way for programs to use the information. Some frame-

works, for instance GLUE, Artix, and Leela, (also) support other non-standardized lookup services to get around these problems.

In summary, Web Services apply the patterns in a different way than conventional object-oriented middleware because they serve very different purposes. Even though Web Services can be used for RPC communication only, the liabilities in this area indicate that they should not be used for this purpose. Web Service are much more about loosely coupled exchange of messages. Thus their strength lies in the complex, service-based integration in heterogeneous environments, such as integrating other middleware, orchestrating processes, or enterprise application integration.

17 CORBA Technology Projection

CORBA is the old man amongst the distributed object middleware technologies used in today's IT world. CORBA stands for Common Object Request Broker Architecture. It defines a distributed object middleware by its interfaces, their semantics and protocols used for communication. In of the used communication paradigm it adheres strongly to the client/server model. We use C++ as programming language as most CORBA applications have been written using it in the past and today.

Brief History of CORBA

CORBA is represented by the OMG (Object Management Group), which is a consortium of over 600 companies, among them are almost all big players in the IT world.

The OMG issues specifications that are in turn implemented by various ORB vendors. An Object Request Broker (ORB) is the core of CORBA, what it is and how it works is described in this chapter. Around that core a series of so-called Common Object Services exist, such as the Naming Service, Event Service, Notification Service, and Logging Service.

The first version of CORBA was finalized in 1991. Since then many ORBs got implemented, commercial ORBs and open-source ORBs. Since approximately 1998 an increasing number of open-source implementations, such as TAO, OmniORB, and JacORB appeared. Today, open-source and commercial ORBs (such as thos of IONA and Borland) exist side-by-side.

CORBA allows to access objects in a distributed environment, very similar to how they are accessed locally. Still there is some explicit setup and configuration work necessary, burdening the programmer, but also allowing him to fully control the behavior.

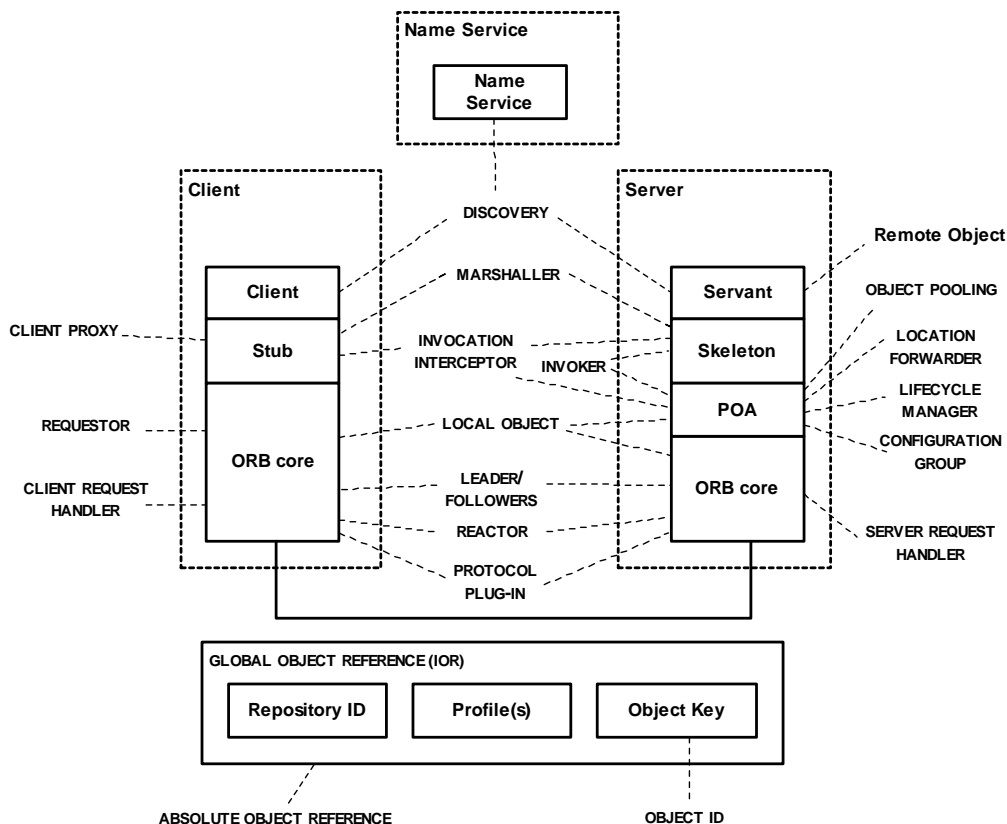
Component frameworks, such as EJB [Sun03a] or CCM [Sev03] hide this code by doing all the setup and configuration internally. CORBA is an important part of the J2EE standard, actually, every Java Development Kit (JDK) ships with CORBA support and a default implementation provided by Sun. This way, CORBA has become the working tier inside modern component platforms, and people get oblivious of CORBA, believing that it is outdated, therefore not needed any more in today's IT world. However, this view is not quite correct as CORBA is used as a distributed object middleware within these component and integration platforms.

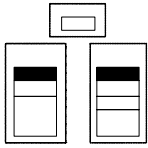
In the past few years, CORBA has gained increasing acceptance in distributed real-time and embedded systems development. By this CORBA followed the classic Technology Adoption Life Cycle [Rog95], where successful systems eventually start to appeal to conservative users and even techno-phobes, because they eventually become convinced that the technology is well proven, solid, and reliable. Traditionally, real-time and embedded systems developers are very conservative, so it's no surprise that it has taken this long for CORBA to appeal to them. The available processing power and low prices for memory helped, too.

The domain used to rely on proprietary middleware in the past, now it uses CORBA as standardized middleware. CORBA explicitly tries to adapt to this domain with its Real-Time CORBA and Minimum CORBA specifications.

Pattern Map

The following diagram summarizes the usage of the Remoting Patterns in this book aligned to the CORBA architecture. Patterns that cover only dynamics, such as those in the *Client Asynchrony* or the *Lifecycle Management* chapters, are therefore not visualized.





Initial Example

As many other distributed object middleware architectures, CORBA relies on interface specifications in separate files. The syntax used in those files is called IDL (Interface Description Language). The following code snippet shows the IDL file of a simple example.

```
interface Foo {
    void bar ();
};
```

An IDL compiler generates code from this, namely *stubs* and *skeletons* that are explained later in this chapter. The IDL compiler also generates an abstract class to be completed as the implementation of the remote object, the POA_Foo class in the example below. The following piece of code represents one of the simplest possible server implementations for the above IDL interface.

```
#include "FooS.h"
class Foo_Impl : public POA_Foo {
    void bar () { cout << "Hello world!" << endl; }
}
int main (int argc, char *argv[])
{
    CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);
    CORBA::Object_var poa_obj =
        orb->resolve_initial_references ("RootPOA");
    PortableServer::POA_var poa =
        PortableServer::POA::_narrow (poa_obj);
    Foo_Impl foo;
    poa->activate_object (&foo);
    PortableServer::POAManager_var poa_manager =
        poa->the_POAManager ();
    poa_manager->activate ();
    orb->run ();
    poa->destroy (1,0); // parameters are explained later
    orb->destroy ();
}
```

The main routine contains the setup and configuration information, necessary to:

- create and initialize the ORB (ORB_init),
- create an object adapter (resolve_initial_references),
- register the remote object implementation (activate_object),
- activate dispatching of requests to the remote object (activate), and

- run the event loop to actually dispatch requests (run).

A client that wants to invoke operations on the remote object looks as follows.

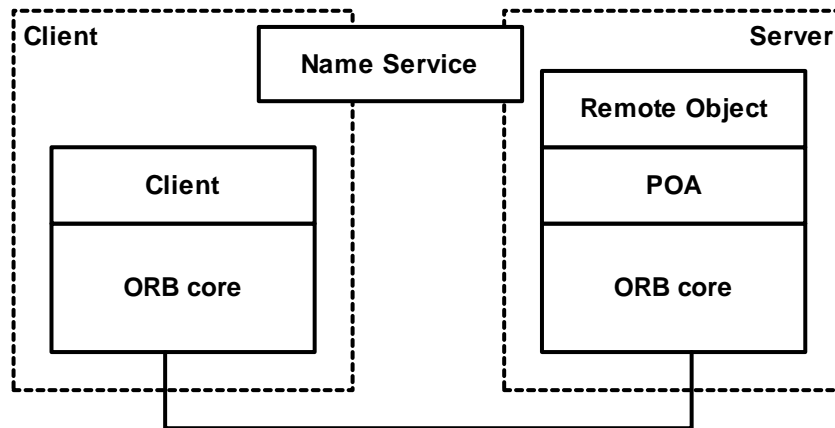
```
#include "FooC.h"
int main (int argc, char *argv[])
{
    // ...
    CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);
    // read the object reference to the remote object from the
    // arguments list in position 1
    CORBA::Object_var obj =
        orb->string_to_object (argv[1]);
    Foo_var foo = Foo::_narrow (obj);
    // do the actual invocation
    foo->bar ();
    orb->destroy ();
}
```

The client creates an ORB instance as the server does. It then creates a local proxy for the remote object by converting the stringified object reference, handed over as `argv[1]` in this case, to a CORBA object reference. To finally create a proxy of the correct type, it performs a `narrow` operation on the object reference. The proxy, represented by the variable `foo` is now ready for usage, operations can be invoked on it. The proxy follows the CLIENT PROXY pattern, but more about its responsibilities later.

CORBA Basics

The CORBA architecture consists of multiple components. The ones used in most applications are:

- Object Request Broker core (ORB core) – responsible for basic communication mechanisms.
- Portable Object Adapter (POA) – responsible for dispatching requests to registered remote objects.
- Naming Service – responsible for the central distribution of references to remote objects.



Looking at the constituent parts from a deployment perspective, the POA and ORB core are very often represented as libraries. Applications implementing remote objects are mostly executables that link with the POA and the ORB library. The reason for this separation is footprint optimization. Separate libraries have two advantages:

- Different implementations can be provisioned, customized to the use cases the ORB is employed in, and
- an application can include only those components that it really needs, keeping the overall footprint at a minimum.

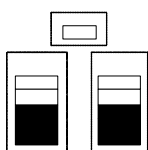
To give an example: Some ORB vendors provide a minimum POA implementation, specialized for embedded systems with fixed requirements, and a complete POA implementation, needed for enterprise systems. The savings in footprint comparing a full featured POA and a minimum POA can be quite significant. Further, pure client-side ORBs

can save footprint by not including the POA, as they do not serve remote objects.

One a general note: this technology projection of how CORBA is build on the patterns is independent of any ORB implementation, but how exactly the patterns are implemented is ORB vendor dependent. ORB vendors are free to implement the ORB as they see fit it, as long as they adhere to the standardized interfaces, semantics, and separation between responsibilities.

The ORB core itself is designed to be compact. It implements the essentials of connection and request handling: SERVER REQUEST HANDLER and CLIENT REQUEST HANDLER. Actual request dispatching functionality is kept in the POA: INVOKER. This *Microkernel* [BMR+96] architecture has been proved successful in many other architectures, it has been proved useful not only in ORB design, but also in operating systems. Microkernel architectures are built on a core that provides the most basic functionality, and layers around that extend this functionality. They are especially useful in situations, where future usage scenarios and extensions cannot be foreseen.

Looking at the current CORBA 3 specification, this design principle has been followed until today. Today, most ORB vendors are able to extend their ORBs with shells of functionality in add-on libraries implementing features, such as RT-CORBA or Portable Interceptors.



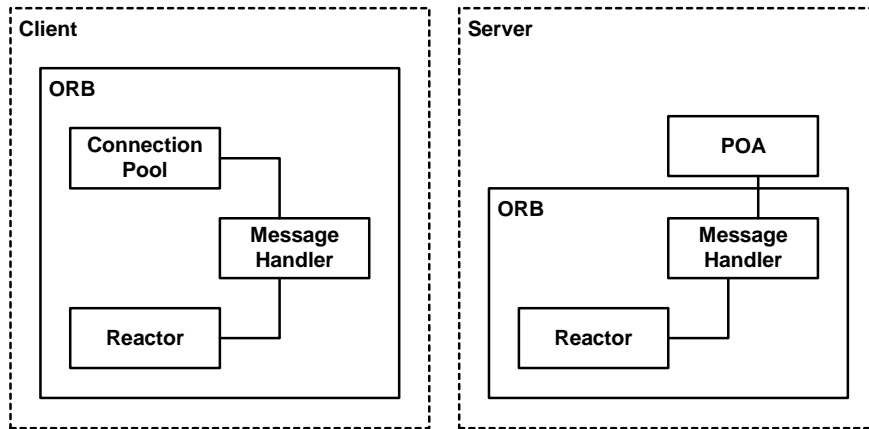
Basic Connection and Message Handling

For managing network connections and for basic sending and receiving of messages, the ORB implements the SERVER REQUEST HANDLER and CLIENT REQUEST HANDLER patterns. Both patterns implement basic request handling functionality. On an initial request from a client to a server, the client ORB will check the connection pool for existing connections to the server. If none is available, it will create one and add it to the connection pool [KJ04]. It then associates a message handler, implementing the CLIENT REQUEST HANDLER pattern, with that connection to send and receive bytes of marshaled requests and responses.

On the server-side, a new message handler, implementing the SERVER REQUEST HANDLER pattern, is created when a new connection is accepted, it gets associated with the connection, to receive and send

bytes of marshaled requests and responses. The POA, which implements part of the INVOKER pattern, finally dispatches the request.

The *Reactor* [SSRB00] helps to avoid blocking on a single connection establishment or send/receive operations. It allows to efficiently demultiplex and dispatch events, such as connection establishment, send, and receive, to corresponding message handlers. On the client-side the reactor dispatches responses, on the server-side it dispatches requests.



It should have become clear that client- and server-side ORBs are not so different. Both use much of the same infrastructure, for example the reactor. We will see in the remainder of this chapter that the reactor is not the only shared component.

References to ORB extensions, such as the POA¹ and the *Current* object, representing the current thread, are bootstrapped by an invocation to `ORB::resolve_initial_references`.

The `ORB::resolve_initial_references` operation looks up an internal table for the reference. In the sample code of the server above, we saw some code, saying:

```
CORBA::Object_var poa_obj =
    orb->resolve_initial_references ("RootPOA");
```

1. For a CORBA server, the POA is no extension, but something necessary. Still, from a purely architectural perspective, the POA extends the ORB core with customizable dispatching functionality. For real-time systems specialized POA implementations exist.

```
PortableServer::POA_var poa =
    PortableServer::POA::_narrow (poa_obj);
```

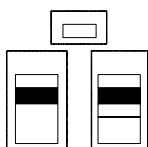
The code essentially provides the server application with a reference to the root POA. The root POA is, as the name, says, the root of the POA hierarchy, other POAs can be created and retrieved via the reference to the root POA. More about the POA hierarchy and the INVOKER later.

The ORB extensions themselves are represented as LOCAL OBJECTS and their creation is typically triggered by the ORB core creation. LOCAL OBJECTS are represented by instances of regular C++ classes that have the same reference passing semantics as remote objects, with the difference that they are not remotely accessible. They are registered ORB-internally, so that CORBA applications can query for references to them in order to configure and use them.

To avoid creating and initializing a POA in client applications that do not need a POA (it is responsible for dispatching requests to remote objects), the POA component can be created and initialized on the first lookup. So when a server application does the lookup of the root POA via `ORB::resolve_initial_references`, the POA gets created. The *Service Configurator* [SSRB00] pattern documents how components, such as the POA can be loaded on demand.

As the expression “Object Request Broker” already indicates, the ORB is responsible for brokering requests between objects, following the BROKER pattern. There has always been some confusion around the term ORB: some thought of it as one big conglomerate—one big communication bus—of many ORB instances running all over the world, others separated the ORB instances and looked at them as inter-operating, self-contained ORBs, sometimes even differentiating between client- and server-side ORBs. As reality is more like the later variant, we prefer to think of it this way.

Even on a local host, there is no single ORB process through which all requests and responses flow, much more there are many ORB instances, one, sometimes several, per operating system process that communicate with each other.

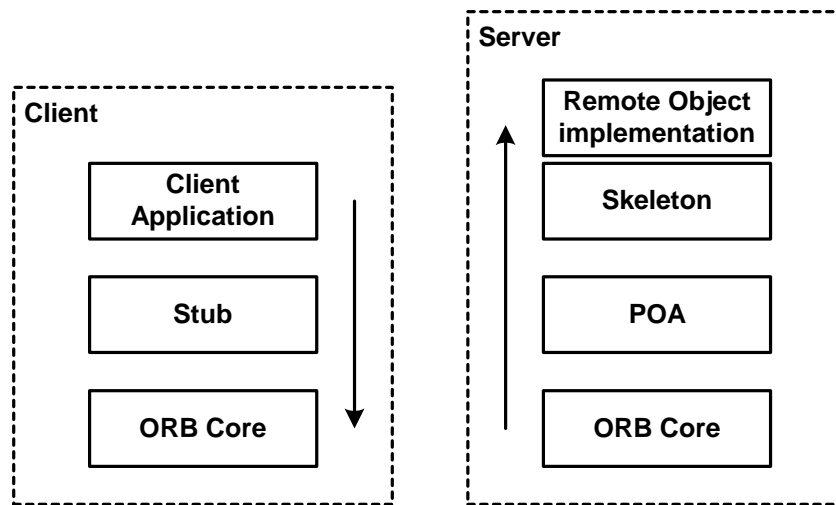


Client and Server Adaptation

Many distributed object middleware implementations represent remote objects to clients by the means of CLIENT PROXIES. In CORBA

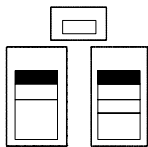
those proxies are called *stubs*. The stub provides the same interface as the remote object, while it also acts as REQUESTOR, so it is responsible for marshaling the request parameters, too. The MARSHALLER is either contained in the stub, or a separate component, used commonly by client stubs and the server-side pendant.

References to CLIENT PROXIES and LOCAL OBJECTS are reference counted, which means as soon as all reference holders give up their reference to them, they destroy themselves. The implementations follow the *Counted Handler* [Hen98] pattern.



The server-side pendant to the stub is the *skeleton*. The skeleton implements together with the POA the INVOKER pattern. The skeleton is therefore responsible for actually dispatching invocations to the target remote object. It has to demarshal in and inout parameters before the invocation of the remote object and marshall out and inout parameters and results after the invocation. As before, for marshalling and demarshaling the MARSHALLER pattern is applied.

Interface Description



Interfaces of remote objects are described in CORBA using a separate language, different than any programming language. The reason for this lies in the mission of CORBA to be platform and programming language independent. The language used is called *Interface Definition Language* (IDL) and is part of the INTERFACE DESCRIPTION pattern.

Descriptions aim at the interface and the data types used by a remote object. An IDL compiler, which is always part of an CORBA implementation, uses the descriptions to create the client-side stub and server-side skeleton.

The IDL also allows to define the exceptions an operation can throw; to be exact, the user exceptions that can be thrown. CORBA system exceptions can be thrown by every operation of an interface, even though they are not mentioned in the `INTERFACE DESCRIPTION`. The supported user exceptions have to be declared in IDL, like every other data type. The `REMOVING ERROR` pattern describes how they are used by remote objects and the respective server-side ORB to communicate execution errors back to the client.

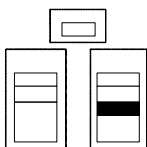
```
exception FooError { string reason; };

interface Foo
{
    void bar () raises (FooError);
};
```

The example declares a user exception `FooError` that can be thrown by the `bar` operation of the `Foo` interface. In C++ and Java, the language mapping allows to throw CORBA exceptions as native exceptions. The following code snippet shows how:

```
try {
    foo->bar ();
}
catch (FooError &ex) {
    cout << ex.reason << endl;
}
catch (CORBA::SystemException &ex) {
    cout << "Critical CORBA system exception" << endl;
}
}
```

Server-Side Dispatching



The Portable Object Adapter (POA) also implements the `CONFIGURATION GROUP` pattern. The POA is the glue between ORB and skeleton as displayed in the previous figure. Remote objects are registered by the server application with the POA. This allows the POA to apply configuration parameters to the remote objects. Invocations to remote objects are forwarded by the ORB core to the POA, which then dispatches them to the skeletons of remote objects. To find the correct POA, the

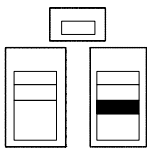
ORB core has to demarshal parts of the request — thus it needs the name of the POA. The POA then continues demarshaling parts of the request retrieving the OBJECT ID in order to find the correct remote object. The POA invokes the skeleton with a reference to the servant, the implementation of the remote object, plus the not yet demarshaled parts of the request, containing the in and inout parameters. The skeleton will demarshal the parameters and execute the invocation on the referenced servant. When the invocation returns, the skeleton marshals the out parameters, inout parameters, and result and hands them back to the POA as the response to be sent back to the client.

```
// Install a persistent POA in order to achieve a persistent IOR
// for our object.
CORBA::PolicyList policies;
policies.length (1);
policies[0] = root_poa->create_lifespan_policy(
    PortableServer::PERSISTENT);
PortableServer::POA_var persistent_poa =
    root_poa->create_POA("persistent",
        poa_manager,
        policies);
policies[0]->destroy ();
```

As the POA realizes a CONFIGURATION GROUP and has to support many configurations, a single POA is not enough. CORBA supports the nesting of POAs, meaning they can get arranged as a tree. At the root of the tree, the root POA is configured with some default policies. All child POAs, with their own set of configuration policies, are grouped below the root POA directly or nested as hierarchy. The deeper the POA hierarchy, the longer is the POA entry in the reference to the remote object, see the section on *Object References*.

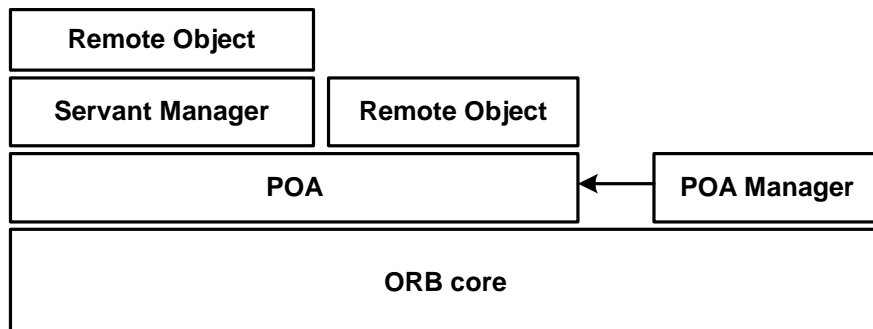
It should have become clear that the responsibilities between ORB core, POA, and skeleton are clearly separated into basic request handling, retrieval and configuration of the remote objects, and actual request dispatching. While ORB core and POA are generic, the skeleton is remote object specific.

Lifecycle Management



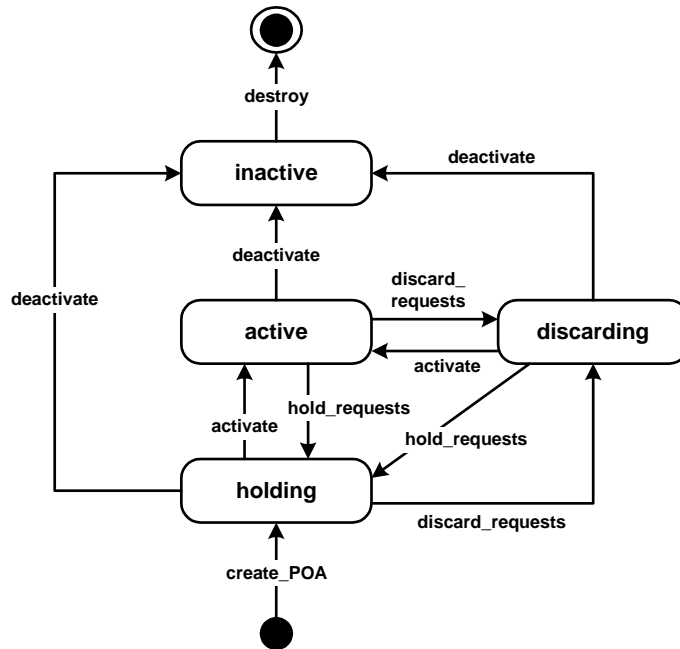
The LIFECYCLE MANAGER pattern is implemented by *POA managers* in combination with *servant managers*. A POA manager is responsible for activation and deactivation of the request flow in the POA to the remote objects, whereas a servant manager is responsible for the actual

activation and deactivation of the remote objects. A servant manager is invoked by the POA instead of the actual remote object. Note that even though POA managers and servant managers extend each others range of influence, they are separate concepts and can be applied solely, without each other.



A POA manager is invoked in the call chain before any servant manager and has no influence on the selection of the remote object. POA managers are optionally created on the call to `PortableServer::POA::create_POA` and are associated to one or

multiple POAs. The following state diagram shows the various states the POA manager can be in.



Servant managers are an ORB extension, they are not implemented by the ORB vendor, but only provisioned as interface. The application can implement custom activation and deactivation of remote objects by implementing the interface and registering the implementation with the POA, but more about them later.

Coming back to the parameters used on the `destroy` operation of the POA we have seen in the introductory Hello World example:

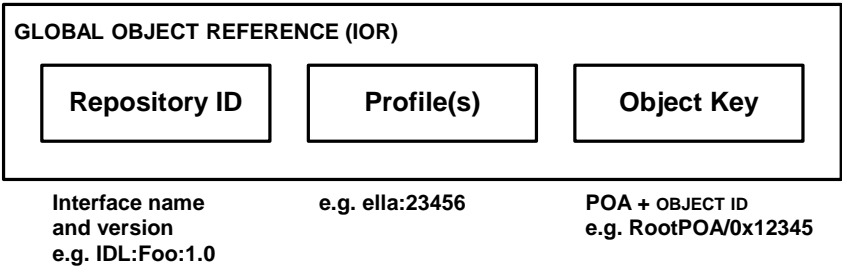
```
poa->destroy (1, 0);
```

The first parameter tells the POA to eventually notify the servant manager that the POA is going to be shut down. The second parameter tells the POA not to wait for pending requests to be completed before shut down. Not to wait for pending requests has the advantage that those requests cannot block the ORB, and thereby the application, from being destructed.

Object References

Remote objects have to be identified at different levels, which means the respective ID must be unique at each level. Inside the POA, remote objects are identified by an OBJECT ID. This OBJECT ID can be chosen by the server application or generated by the POA, when activating the remote object. CORBA identifies remote objects via object references. Object references always contain an OBJECT ID.

When object references are passed outside the local process, additional information is needed for unique identification. The generation of process-external unique object references follows the ABSOLUTE OBJECT REFERENCE pattern – in CORBA ABSOLUTE OBJECT REFERENCES are called *Interoperable Object Reference* (IOR). To make the IOR globally unique the ORB adds information about ORB endpoints and the POA, the remote object is registered with. Endpoint information is described in the form of so called *profiles*, containing host IP address and port number. The following figure illustrates the structure of the IOR.

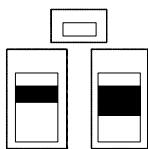


A specific servant can only be registered with one POA, and therefore one ORB. This restriction ensures that the servant is exposed only to intended concurrency – configured via the POA policies. As a POA can be configured to serialize requests to a remote object, the server application can ensure that the servant, alias the remote object, is protected from concurrent access. But if it would be registered with two ORBs, and thereby with two POAs, unintended concurrent access could occur and possibly lead to data corruption.

There is an interesting story about how CORBA got used in real-time systems in the early days: The problem was that servants existed that had to serve requests at different priorities. Neither the client-side nor the server-side of an ORB did distinguish between different request properties when sending or receiving requests, nor when dispatching them. The work-around was that multiple server-side ORBs were

instantiated, each running in a thread with a different priority; the servant was registered with each ORB—of course, the servant had to be thread-safe. IORs were generated for each remote object but identifying the same servant in *ORB A* and *ORB B*. This way clients were able to distinguish between priorities using either the IOR of the remote object served by *ORB A* or the one served by *ORB B*. In later sections you will see how this is done in a standardized way by Real-Time CORBA, today.

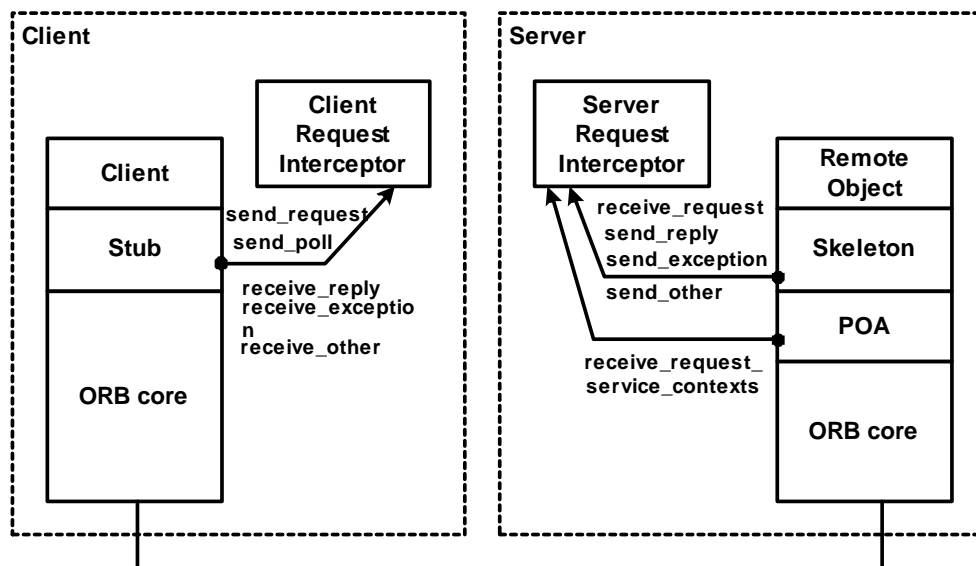
Invocation Extensions



It was already mentioned that CORBA was designed to be highly extensible. In situations, where transparent services, like security or transactions, need to be supported, low-level control during the actual dispatching of requests is needed. Adding just another wrapper around the microkernel is insufficient, as requests and responses would have to be demarshaled and marshaled again in order to make any changes or additions to invocations.

The INVOCATION INTERCEPTOR pattern allows to integrate such transparent services. In CORBA, the instantiation of this pattern are the Portable Interceptors.

Interceptors can be used at different levels of request processing. The following figure shows the interception points.



Additional services, as motivated above, often need some context information about the request or response, for example a transaction service needs to carry the transaction ID from the client to the server and back, to associate the request with the corresponding transaction. An INVOCATION CONTEXT is the common solution; in CORBA it is called *service context* and can contain any value including binary data. Service context information can be modified via the *portable interceptors*.

Portable interceptors in CORBA are separated into *client request interceptors* and *server request interceptors*. The following class declaration shows an empty implementation of a client request interceptor. The method implementations can access any request information via a pointer to the *client request information*.

```
class Foo_ClientRequestInterceptor
: public virtual PortableInterceptor::ClientRequestInterceptor
{
public:
    virtual char * name () throw (CORBA::SystemException)
    {
        return CORBA::string_dup ("Foo");
    }
    virtual void destroy () throw (CORBA::SystemException) {}
    virtual void send_request (
        PortableInterceptor::ClientRequestInfo_ptr
        throw (CORBA::SystemException,
            PortableInterceptor::ForwardRequest)
        {
            cout << "send_request invoked" << endl;
        }
    virtual void receive_reply (
        PortableInterceptor::ClientRequestInfo_ptr
        throw(CORBA::SystemException) {})
    // ...
};
```

Client and server request interceptors must get registered inside an *ORB initializer*. For this purpose the ORB initializer is separated into two methods, a *pre_init* and a *post_init* method, which are invoked before and after the initialization of the ORB respectively.

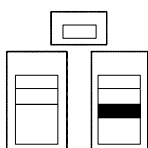
```
class Client_ORBInitializer :
    public virtual PortableInterceptor::ORBInitializer
{
public:
    void pre_init (PortableInterceptor::ORBInitInfo_ptr)
        throw (CORBA::SystemException)
    {
    }

    void post_init (PortableInterceptor::ORBInitInfo_ptr info)
        throw(CORBA::SystemException)
    {
        PortableInterceptor::ClientRequestInterceptor_var
            interceptor (new Foo_ClientRequestInterceptor);
        info->add_client_request_interceptor (interceptor);
    }
};
```


The ORB initializer is registered with the ORB library before a actual ORB instance is created by `CORBA::ORB_init`.

```
PortableInterceptor::ORBInitializer_var initializer (
    new Client_ORBInitializer);
PortableInterceptor::register_orb_initializer (initializer.in());

CORBA::ORB_var orb = CORBA::ORB_init (argc, argv, "");
```



Activation Strategies

The simplest and most straightforward usage of the POA is that for **STATIC INSTANCES**: Activation of remote objects at server application startup time.

CORBA servant managers have already been mentioned in the context of the POA above. They are used when the trivial activation of servants at startup time, as in the case of **STATIC INSTANCES**, is not sufficient. This typically happens in servers that have to deal with a huge number of remote object instances. Servant managers are a separate concept of interception besides the Portable Interceptors, they are specialized for servant activation and deactivation and are historically much older than Portable Interceptors.

LAZY ACQUISITION allows to defer the activation of remote objects to the time when the first request reaches the remote object. The activation is done by a *servant activator*, as one kind of CORBA servant manager. The servant activator is registered with the POA, which issued the object reference to the yet-not-activated remote object. The servant activator is invoked via the *incarnate* operation by the POA to activate the object. Further requests are dispatched to the now activated remote object directly.

In order to use a servant activator, the POA has to be configured in the following way:

```
CORBA::PolicyList policies;
policies.length (3);
policies[0] = poa->create_request_processing_policy(

PortableServer::USE_SERVANT_MANAGER);
policies[1] = poa->create_servant_retention_policy (
    PortableServer::RETAIN);
policies[2] = poa->create_id_assignment_policy (
    PortableServer::USER_ID);
```

```
PortableServer::POA_var servant_manager_poa =
    poa->create_POA("servant activator",
        poa_manager,
        policies);
policies[0]->destroy ();
policies[1]->destroy ();
policies[2]->destroy ();
```

For servant activators, the RETAIN policy has to be used, so that active object map of the POA keeps track of activated servants. The servant activator can now get registered with the newly created POA as follows:

```
MyActivator activator;
PortableServer::ServantManager_var servant_manager = &activator;
servant_manager_poa->set_servant_manager (servant_manager);
```

At this point, an object reference to the yet-not-activated remote object is created with a user-defined OBJECT ID, which allows us to identify requests for a specific servant.

```
PortableServer::ObjectId_var object_id =
    PortableServer::string_to_ObjectId ("Foo");

CORBA::Object_var object =
    servant_manager_poa->create_reference_with_id (object_id,
        "IDL:Foo:1.0");
```

Once activated, it is hard to monitor the further lifecycle of the servant by using servant activators, because activators are only involved on the first invocation on a remote object, not on any later invocation. *Servant locators* are more flexible. A servant locator is invoked on every invocation of the remote object. This has the advantage of better and more fine grained control, but obviously also influences the performance negatively, as the servant locator needs additional CPU-cycles on every invocation of the remote object.

The `PortableServer::ServantLocator` has two operations `preinvoke` and `postinvoke`. The operation `preinvoke` is invoked before every invocation on the remote object identified by the OBJECT ID, while `postinvoke` is invoked after every invocation.

The following servant locator implements the pattern PER-REQUEST INSTANCE. It trivially creates a new servant on every invocation. In this case the servant locator destroys the servant after the invocation has been executed, though it could also recycle servants.

```
class MyLocator : public virtual PortableServer::ServantLocator
```

```

{
public:
    virtual PortableServer::Servant preinvoke (
        const PortableServer::ObjectId & oid,
        PortableServer::POA_ptr poa,
        const char * operation,
        void * & cookie)
    {
        throw (CORBA::SystemException, PortableServer::ForwardRequest)
    }

    PortableServer::Servant servant = 0;
    servant = new Foo_Impl;
    return servant;
}

virtual void postinvoke (
    const PortableServer::ObjectId & oid,
    PortableServer::POA_ptr poa,
    const char * operation,
    void * cookie,
    PortableServer::Servant servant)
{
    throw (CORBA::SystemException)
{
    delete servant;
}
};

```

To be able to register the servant locator with the POA, the POA has to be configured in the following way.

```

CORBA::PolicyList policies;
policies.length (3);
policies[0] = poa->create_request_processing_policy(
    PortableServer::USE_SERVANT_MANAGER);
policies[1] = poa->create_servant_retention_policy (
    PortableServer::NON_RETAIN);
policies[2] = poa->create_id_assignment_policy (
    PortableServer::USER_ID);
PortableServer::POA_var servant_manager_poa =
    poa->create_POA("servant locator", poa_manager, policies);
policies[0]->destroy ();
policies[1]->destroy ();
policies[2]->destroy ();

```

Note, we use the NON_RETAIN policy, which tells the POA to not maintain an active object map with the mapping between OBJECT ID and servant; this differentiates the servant locator from the servant activator. The servant locator has to keep track of the OBJECT ID-to-servant mapping itself. The servant locator registration and object reference creation is similar as for the servant activator.

One way to apply servant managers is to perform POOLING, meaning the servant manager manages a pool of servants. A servant locator would look up a servant from the pool in `preinvoke` and hand it to the POA. On `postinvoke` it would return the servant into the pool, avoiding the repetitious creation and destruction of servants.

```
class ObjectPoolLocator :
    public virtual PortableServer::ServantLocator
{
public:
    virtual PortableServer::Servant preinvoke (
        const PortableServer::ObjectId & oid,
        PortableServer::POA_ptr poa,
        const char * operation,
        void * & cookie)
    {
        throw (CORBA::SystemException, PortableServer::ForwardRequest)
    }

    virtual PortableServer::Servant servant = 0;
    servant = list.get ();
    if (!servant)
        // throw an transient system exception if no servant
        // is available
        throw CORBA::TRANSIENT (CORBA::OMGVMCID,
                                CORBA::COMPLETED_NO);

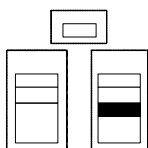
    return servant;
}

virtual void postinvoke (
    const PortableServer::ObjectId & oid,
    PortableServer::POA_ptr poa,
    const char * operation,
    void * cookie,
    PortableServer::Servant servant)
{
    throw (CORBA::SystemException)
{
    list.put (servant);
}

List<Servant> list;
};
```

The way CORBA is designed, it is not directly possible for servers to identify clients; remote objects but also the POA and servant locator have no chance to identify the client performing the invocation. Therefore, so called CLIENT-DEPENDENT INSTANCES have to be supported at the application layer. This means CLIENT-DEPENDENT INSTANCES are not supported transparently by CORBA. Of course, INVOCATION INTERCEPTORS implemented as part of the application layer could integrate this

transparently, while the rest of the application layer could stay oblivious of this.



Location Forwarding

Another interesting use case of servant managers is location forwarding. Location forwarding means that a server receiving a request for a remote object can forward the request to another remote object, possibly located on another server. The client ORB is informed about forwarding, and it is responsible for resending the original request to that (different) remote object. Resending is transparent to the client. Location forwarding can be triggered by applications but also ORB-internally.

To trigger a location forward, applications need to throw a `ForwardRequest` exception configured with the object reference of the remote object to forward requests to. Servant managers, servant locators as well as servant activators, can be used for this task by their `preinvoke` and `incarnate` operations, respectively. In that role, the servant managers implement the `LOCATION FORWARDER` pattern. For example, a servant locator could implement the `preinvoke` operation as follows:

```
virtual PortableServer::Servant preinvoke (
    const PortableServer::ObjectId & oid,
    PortableServer::POA_ptr poa,
    const char * operation,
    void * & cookie)
{
    throw (CORBA::SystemException, PortableServer::ForwardRequest)
    {
        // get the object reference of the remote object to forward to
        CORBA::Object_var my_forwarding_target = //...

        // forward the requests to forwarding_target
        throw PortableServer::ForwardRequest (
            CORBA::Object::_duplicate (forwarding_target.in ()));

        return 0;
    }
}
```

Servant managers are not the only way to forward requests at the application level, it can also be done by the client and server request interceptors mentioned above. If you look for `PortableInterceptor::ForwardRequest` exception in the throw definitions of the operation signatures, you can quickly identify which operations of the

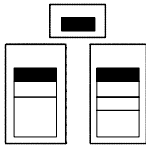
interceptors can be used to implement a LOCATION FORWARDER. Inside the ORB, the `ForwardRequest` exception is translated into a `LOCATION_FORWARD` status of the response to the client.

To implement the LOCATION FORWARDER pattern at the ORB level, the ORB has to decide when to return a location forward response. This is especially interesting in the case of scalability; the ORB would balance load between several remote objects by forwarding requests to remote objects on other servers.

One disadvantage of using the `LOCATION_FORWARD` status is that the transmission of the request parameters is lost when the request is forwarded. The client has to re-transmit the complete request to the new remote object. Client ORBs can avoid this by sending a so called *LocateRequest* instead of a regular request. The request will be answered with a *LocateReply* telling, either:

- the object does not exist,
- the object exists and is accessible, or
- the object is somewhere else and here is the object reference to it.

Initial Access to Remote Objects



Initial access to remote objects can be gained in multiple ways. When the desired remote object *X* is known to any other remote object *Y*, *Y* can pass *X*'s reference to the client. If no other remote object can be queried, the stringified ABSOLUTE OBJECT REFERENCE, the IOR, needs to be passed to the client by some means. When exchanged manually, it is mostly exchanged via a file, but it can also be exchanged via e-mail or cut & paste by application users.

In CORBA, the LOOKUP pattern is implemented by the CORBA Naming Service. The Naming Service is a special remote object. Its only purpose is the mediation of ABSOLUTE OBJECT REFERENCES. Server applications register object references of remote objects at the Naming Service with a unique name. These names can be hierarchically organized in so called *name contexts*. The object reference to the Naming Service itself is made available to clients via pre-configured IORs or a multicast-based protocol.

Before a client or server can use the Naming Service it needs to obtain a reference to its *initial name context*. The initial naming context can get

obtained from the ORB via the `resolve_initial_reference` operation on the `CORBA::ORB` interface. To get the initial naming context object reference, the operation must be used with the argument "NameService". The returned object reference must be narrowed to `CosNaming::NamingContext` before it can be used.

```
CORBA::Object_var object =
    orb->resolve_initial_reference ("NameService");
CosNaming::NamingContext_var naming_context =
    CosNaming::NamingContext::_narrow (object.in());
```

The ORB offers the startup option "-ORBInitRef" which can be used to specify the initial object reference for a service. This initial object reference is returned when the `resolve_initial_reference` operation is called on the ORB. For the Naming Service an entry like "-ORBInitRef NameService=IOR:0d3e42..." configures the ORB so that `resolve_initial_reference` will return the proper object reference to the Naming Service. Alternatively, some ORB vendors implement a multicast-based protocol that allows a call to `resolve_initial_reference` to find a Naming Service instance. The multicast-based lookup is often prone to errors and is therefore the last choice in designs of distributed applications, today.

The object references of remote objects can be bound to a naming context. As naming contexts can be nested, the naming context interface contains two operations to create bindings: one for regular objects (`bind` operation) and one for contexts (`bind_context` operation).

A regular remote object can be bound to a naming context in the following way:

```
CosNaming::Name name;
name.length (1);
name[0].id = CORBA::string_dup ("Foo");
naming_context->bind (name, foo);
```

If a binding already exists under that name an exception is thrown. To override an existing binding the `rebind` operation can be used instead of the `bind` operation.

A client has to use the `resolve` operation on a naming context of the Naming Service to lookup an object reference. Because the returned object reference is of the general type `CORBA::Object`, the client has to narrow the reference to its correct type.

```
CosNaming::Name name;
```

```
name.length (1);
name[0].id = CORBA::string_dup ("Foo");
CORBA::Object_var obj = naming_context->resolve (name);
foo->bar (...); // actual invocation
```

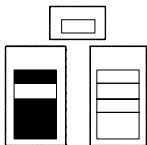
Messaging

For a long time CORBA supported only synchronous communication between client and server. By defining operations as *oneway* in the IDL this was somewhat relaxed, but firstly, oneways are not guaranteed to be asynchronous and secondly, they were defined as unreliable operations. So application programmers never quite could decide whether to use them or not. When they did, it sometimes lead to unexpected results, such as situations where the sending of requests blocked—TCP flow control kicked in— or users expected oneways to be received at the server in the same order the client sent them, but ORB implementations threw messages away or reordered them—which is actually allowed by the specification.

The Event and Notification CORBA Common Object Services, also provide asynchronous behavior, but only at a high expense. The event and notification services are represented as remote objects that are accessed synchronously via two-way operations and act as a pretty heavyweight *Mediator* [GHJV95]. They are mainly applicable to scenarios where producers and consumers need to be decoupled. When filtering is important, the notification service offers a valuable solution out of the box.

With the Messaging specification as part of CORBA 2.4, things changed. *Asynchronous Method Invocations* (AMI) got introduced in two variants: The *callback* and the *polling* model.

Client-Side Asynchrony



In practice the callback model, implementing the RESULT CALLBACK pattern, revealed to be more practical, than the polling model, implementing the POLL OBJECT pattern. Therefore, mostly only the callback model is implemented by ORB vendors. The callback model relies on so called *reply handlers*, implemented by the client and registered with the client ORB. The reply handlers receive responses to asynchronously invoked requests. It is important to note that AMI is a pure

client-side functionality, the server-side ORB does not notice the difference between asynchronous and synchronous invocations.

To explain how AMI works, we define a new example:

```
exception MyException { string reason; };
interface Foo
{
    long bar (in long in_1,
              inout long inout_1,
              out long out_1)
        raises (MyException);
};
```

To send invocations asynchronously, a new operation is introduced for every original operation. This additional operation always has the prefix `sendc_` followed by the name of the original operation. The `sendc_` operations and the reply handler interface do not have to be specified in IDL, instead the IDL compiler can be instructed to generate them.

For this, the IDL compiler typically creates intermediate code internally, adding a `sendc_` operation to every interface and adding a corresponding reply handler interfaces. So for the example above the following additional IDL definitions are generated internally in the IDL compiler.

```
interface Foo
{
    // ..
    void sendc_bar (out long out_1, in long inout_1);
};

valuetype AMI_HelloWorldExceptionHandler
: Messaging::ExceptionHandler
{
    void raise_foo ()
        raises (MyException);
};

interface AMI_HelloWorldHandler : Messaging::ReplyHandler
{
    void foo (in long result,
              in long inout_1,
              in long out_1);
    void foo_excep (in AMI_HelloWorldExceptionHandler excep_holder);
};
```

All the application programmer sees from this are the extended stubs of the original interfaces and the skeletons for the reply handlers.

The client has to implement the reply handler interface and register the implementation with its POA. Note that the reply handler servant forces the client be a server as well - a server for the replies of the `Foo` servant. For some applications this can increase the memory footprint and execution overhead (as there is an additional thread running `ORB::run`) significantly.

```
class Foo_Handler_Impl : public POA_AMI_FooHandler
{
public:
    void bar (CORBA::Long result,
              CORBA::Long inout_l,
              CORBA::Long out_l)
    {
        throw (CORBA::SystemException)
        { cout << "bar result arrived" << endl; }

        void bar_excep (AMI_FooExceptionHolder * excep_holder)
        {
            throw (CORBA::SystemException)
        }
    }
};
```

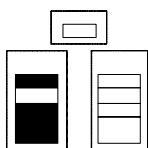
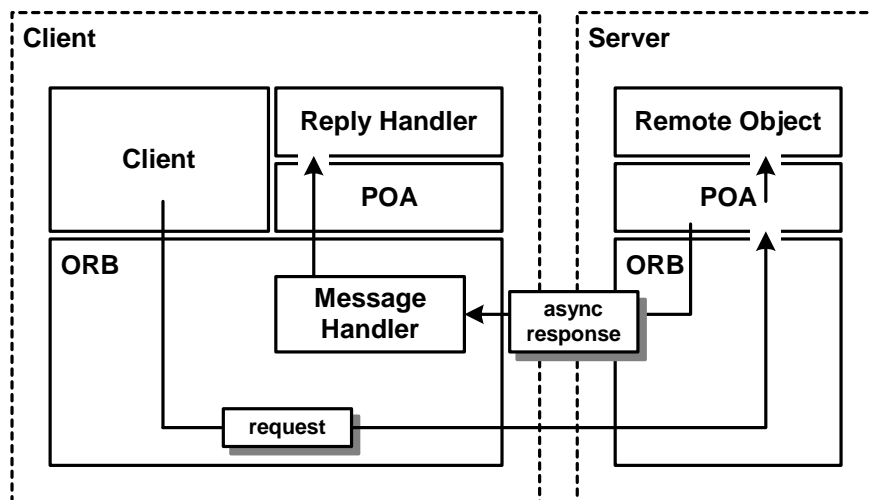
To invoke an operation asynchronously on the server, the client uses the `sendc_operation`. It has only to supply the `in` and `inout` arguments, no variables need to be provided for `out` and `inout` arguments or results. As first argument, the object reference to the reply handler has to be handed over.

```
foo->sendc_bar (reply_handler, in_number, inout_number);
```

When the server sends the reply back to the client-side ORB, the ORB will dispatch the reply as a request to the reply handler, invoking the `bar` operation on it with `inout`, `out` arguments, and the return value. If the reply contains an exception, it will invoke the `foo_excep` operation.

The following figure shows how requests from the client are sent to the server-side ORB via the message handler. It memorizes the request ID used and associates the handed-over reply handler with it. As soon as

the asynchronous reply arrives at the message handler, it dispatches it to the responsible reply handler via its POA.



Oneways

Before CORBA version 2.4, oneways were defined to be unreliable. This has changed with the appearance of the Messaging specification, CORBA now defines so called reliable oneways. To set the reliability of a oneway operation, the client has to set a policy.

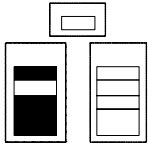
The policies of which the client can choose are:

- **SYNC_NONE** – The client is guaranteed not to block, the ORB returns control before it passes the request message to the transport protocol. This policy implements FIRE AND FORGET.
- **SYNC_WITH_TRANSPORT** – The ORB returns control to the client only after the transport protocol has accepted the request message. This policy results in a slightly more reliable invocation style than FIRE AND FORGET.
- **SYNC_WITH_SERVER** – The ORB returns control to the client only after the server-side ORB has accepted the request message. For this, the server-side ORB sends a reply before invoking the remote object. This policy implements SYNC WITH SERVER.

- **SYNC_WITH_TARGET** – The ORB returns control to the client only after the remote object has been invoked. This policy is equivalent to a synchronous invocation with no inout, out arguments, and return value.

Looking at the code, the policy can be set in various ways, at the ORB level, the `PolicyCurrent` (thread) level, or the proxy level. In the following code snippet, the policy is set at the `PolicyCurrent` – the thread – level.

```
CORBA::Object_var object =
    orb->resolve_initial_references ("PolicyCurrent");
CORBA::PolicyCurrent_var policy_current =
    CORBA::PolicyCurrent::_narrow (object.in ());
CORBA::Any scope_as_any;
scope_as_any <= Messaging::SYNC_WITH_SERVER;
CORBA::PolicyList policies(1); policies.length (1);
policies[0] =
    orb->create_policy (Messaging::SYNC_SCOPE_POLICY_TYPE,
                      scope_as_any);
policy_current->set_policy_overrides (policies,
                                     CORBA::ADD_OVERRIDE);
```



Timeouts

Besides asynchronous method invocations and oneways, the Messaging specification also standardized how timeouts are set. The same policy framework as before is used to set the round-trip timeout of operations – they can be applied to synchronous as well as asynchronous operations. When an operation runs over the set timeout, an `REMOVING ERROR` of the type `CORBA::TIMEOUT` exception is raised by the client-side ORB while the operation may or may not have been executed by the server. The following code shows how a timeout is set for the current thread.

```
// Put timeout in an Any.
TimeBase::TimeT timeOut = requestTimeOut;
CORBA::Any timeOutAny;
timeOutAny <= 50000; // 5 ms = 50000*100ns

// Create policy object.
CORBA::PolicyList policies (1); policies.length (1);
// Create the policy.
policies[0] = orb->create_policy
    (Messaging::RELATIVE_RT_TIMEOUT_POLICY_TYPE,
     timeOutAny);
```

```
// Add the policy to the ORBs policies.
policy_current->set_policy_overrides (orbPolicies,
                                     CORBA::ADD_OVERRIDE);
```

Real-Time CORBA

In the distributed, embedded, and real-time (DRE) domain, often the problem occurs that the QoS of invocations on remote objects must be controlled. As most of the connection management and dispatching behavior is transparent to CORBA applications, stringent predictability and performance requirements could not be met. CORBA lacked explicit control of QoS properties.

With the Real-Time CORBA specification this has changed. The main areas RT-CORBA addresses are:

- Portable priorities—ensuring that priorities, which are different on each operating system, are mapped properly between client and server.
- End-to-end priority preservation—ensuring that priorities are observed end-to-end between client and server inside the involved ORBs.
- Explicit connection management—allowing the server application to determine the time connections are established and defining priority ranges for individual connections.
- Thread pooling—standardizing the support and configuration of thread pools by ORBs.

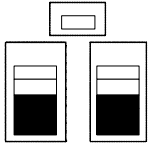
Introducing new features to the existing and widely used CORBA API is a problem. How can the ORB be extended with RT-CORBA features without changing the `CORBA::ORB` API? The solution is to use the *Extension Interface* pattern [SSRB00] which allows to transition between interfaces of the same component. This allows components to extend with additional interfaces, while keeping existing ones; lookup of extended interfaces is done via a special operation.

The aforementioned `resolve_initial_references` on the `CORBA::ORB` interface is used to obtain a reference to the `RTCORBA::RTORB` extension, implementing the extension interface. Thus, non-real-time ORBs and applications are not affected by RT-CORBA extensions.

```

CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);
CORBA::Object_var obj = orb->resolve_initial_references ("RTORB");
RTCORBA::RTORB_var rtorb = RTCORBA::RTORB::_narrow (obj);

```

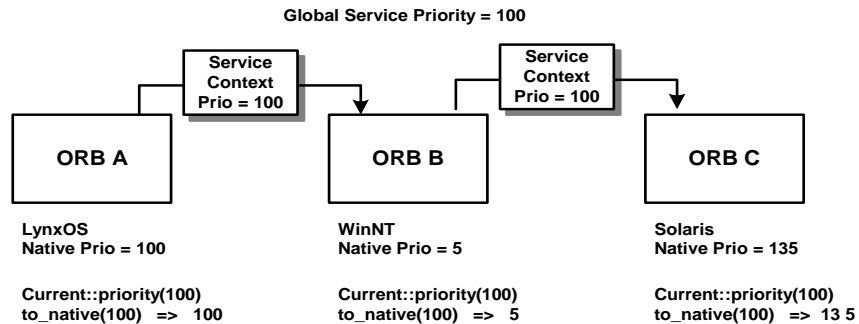


End-To-End Priority Preservation

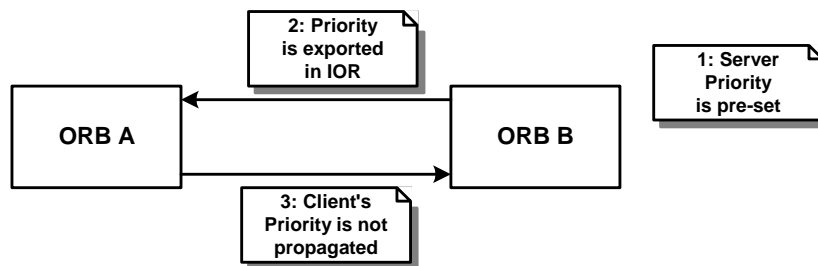
In order to enable end-to-end preservation of QoS, priorities need to be mapped between various operating systems, as every operating system defines its own range of priorities. For example, when invocations of a client running on VxWorks in a thread with a certain priority are invoked on a server running LynxOS, the priority of the dispatching and executing thread should be aligned with that client priority.

The way priorities are handled end-to-end follows two different models:

- Client-propagated model – the client thread's priority gets sent to the server using the INVOCATION CONTEXT, as illustrated in the figure below.



- Server-declared model – the priority gets set in the POA and sent to clients as part of the ABSOLUTE OBJECT REFERENCE. The following figure illustrates this.



In the following code example we register a CORBA object with the POA for the client-propagated model. Operations are invoked with the current client's priority. This means that different operations on the same remote object can be invoked at the different priorities eventually.

```
CORBA::PolicyList policies (1); policies.length (1);
policies[0] = rtorb->create_priority_model_policy
    (RTCORBA::CLIENT_PROPAGATED,
     DEFAULT_PRIORITY /* For non-RT ORBs */);

PortableServer::POA_var my_poa =
root_poa->create_POA ("My_POA",
    PortableServer::POAManager::_nil (),
    policies);

// Activate a servant
my_poa->activate_object (my_servant);
```

Note that the `CLIENT_PROPAGATED` policy is set on the server and exported to the client as part of the ABSOLUTE OBJECT REFERENCE.

Knowing how to configure for end-to-end priorities, we also need to know how to change CORBA priorities at the client. How can RT-CORBA client applications change the priority of operations? The solution is to use the `RTCurrent LOCAL OBJECT` to change the priority of the current thread explicitly, which is similarly accessed as the above mentioned `Current LOCAL OBJECT`. An `RTCurrent` can also be used to query the priority. The values are expressed in the CORBA priority range and the behavior of `RTCurrent` is thread-specific.

```
CORBA::Object_var obj =
    orb->resolve_initial_references ("RTCurrent");
RTCORBA::RTCurrent_var rt_current =
    RTCORBA::RTCurrent::_narrow (obj);
rt_current->the_priority (10/*VERY_HIGH_PRIORITY*/);

// Invoke the request at <VERY_HIGH_PRIORITY> priority
foo->bar ();
```

The implementation of client-propagated policies uses the underlying prioritized connection management system (described below) of the client-side ORB, but also transfers the client's thread priority via the `INVOCATION CONTEXT` to the server-side ORB, so that it can set the correct priority of the server thread.

When operations must be invoked on the remote object always at the same priority, it is recommended to use the server-declared priority model. Priorities are set at the POA level and published to clients via

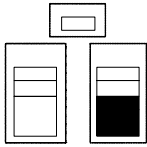
the ABSOLUTE OBJECT REFERENCE. The client-side ORB will already ensure that marshaling, sending, and the selection of the connection happen at the correct priority.

```
CORBA::PolicyList policies (1); policies.length (1);
policies[0] = rtorb->create_priority_model_policy
(RTCORBA::SERVER_DECLARED, LOW_PRIORITY);
PortableServer::POA_var base_station_poa =
    root_poa->create_POA ("My_POA",
        PortableServer::POAManager::_nil (),
        policies);
my_poa->activate_object (my_servant);
```

By default, server-declared objects inherit the priority of their RT-POA. However, it is possible to override this priority on a per-object basis.

The priority of the client thread, invoking an operation of a remote object with a server-declared policy associated, is set to the correct priority in the CLIENT PROXY. Thus marshaling and sending happen at the expected priority.

Thread Pools



Embedded systems are often highly concurrent, for instance, many sensors and actuators are driven by individual threads. As they often depend on each other in a client-server relationship, eventually many clients invoke requests on the same remote object. To cope with such situations, where multiple clients execute the same server applications POOLING is used. Thread pools as implementation of POOLING existed already a long time in CORBA, but the API to use them was not standardized. With RT-CORBA this has changed.

It is now possible to configure thread pools using a standardized API. Instances of thread pools are registered with the POA and are therefore available to all remote objects registered with it. Thread pools are not only groupings of threads, but they also have priorities associated with them. By a concept of so called *lanes*, threads can be grouped by priority inside a thread pool. This way it can be ensured that enough threads of a priority are available. The API of the RT-ORB to create thread pools is shown below.

```
interface RTCORBA::RTORB {
    typedef unsigned long ThreadpoolId;
    ThreadpoolId create_threadpool (
        in unsigned long stacksize,
        in unsigned long static_threads,
```



```

        in unsigned long dynamic_threads,
        in Priority default_priority,
        in boolean allow_request_buffering,
        in unsigned long max_buffered_requests,
        in unsigned long max_request_buffer_size);

void destroy_threadpool (in ThreadpoolId threadpool)
    raises (InvalidThreadpool);
};

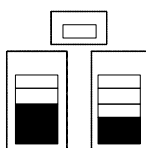
```

The specification also provisioned the API to support request buffering, a topic that existed as long as CORBA and never really got closed. It seems like a good idea to have buffers because you would not lose requests and you would always be available to clients, ideally no requests would be discarded. But looking at the problem of the temporal unavailability of enough server resources, it is hopeless to introduce buffers, as they only delay the problem, they can never really solve it. If we have scarce resources, for example not enough threads or not enough CPU cycles, we only delay the denial of service, we can never avoid it. Plus, request buffering introduces significant overhead and jitter by allocating memory on the heap, that would otherwise be allocated from the stack. So in summary, buffering is very seldom implemented by in ORBs.

The management of threads of the same thread pool is typically implemented using the *Leader/Followers* [SSRB00] pattern, which means the threads manage themselves on who gets to dispatch the next message coming in, which is either a request or response to a previously sent request.

When a thread pool has lanes, every thread implementing `SERVER REQUEST HANDLER` functionality has its own *Reactor* [SSRB00] associated to avoid the risk to dispatch incoming requests and responses at the wrong priority, which could happen if the priority of the dispatching thread is set after it is read from the socket.

Explicit Connection Management



Proper connection management is crucial in order to make invocations more predictable. RT-CORBA allows for explicit connection management in three ways, represented as distinct policies:

- Pre-allocating connections – avoids the initial delay on the first invocation.

- Priority-banded connections — allocates connections to specific priority ranges.
- Private connections — reserves private connections between a client and a remote object.

Note that the policies can be applied in combination, for instance, you can pre-allocate all priority-banded connections to a specific object.

In the following we will go into detail for each of the three policies. Pre-allocated network connections are the solution to the problem that *Lazy Acquisition* [KJ04] of connections, as it is typically done by ORB implementations, can result in unacceptable jitter, which is detrimental to time-critical applications. Pre-allocated connections follow the *Eager Acquisition* [KJ04] pattern. The time at which the connection is acquired eagerly can be determined by the call to `Object::_validate_connection` (defined in the CORBA Messaging specification). It will cause the client-side ORB internally to establish a new connection and tell the server, which priority the connection should have.

```
Foo_var foo= ...; // Obtain reference to the remote object
CORBA::PolicyList_var policies;
CORBA::Boolean successful =
    foo->_validate_connection (policies);
```

Priority-banded connections are useful in situations where requests of various priorities are invoked on the same server application. As a high-priority request might be delayed by a previously sent low-priority request, priority inversions can occur. Using different connections for different priority ranges, setup via the RT CORBA `PriorityBandedConnectionPolicy`, invocations of different priorities are decoupled.

Priority-banded connections allow sharing of connections between a client-side and a server-side ORB for a range of priorities and multiple remote objects; the priority bands are defined at the ORB level. The following code segment shows how two priority bands are set for requests to a remote object. As you can see, when policies are set on a CLIENT PROXY, a new CLIENT PROXY is returned in turn.

```
RTCORBA::PriorityBands bands;
bands.length (2);
bands[0].low = 10000;
bands[0].high = 10005;
```

```

bands[1].low = 26000;
bands[1].high = 30000;
CORBA::PolicyList policies; policies.length (1);
policies[0] =
    rt_orb->create_priority_banded_connection_policy (bands);
Foo_var new_foo =
    client_propagated_obj->set_policy_overrides
        (policies, CORBA::SET_OVERRIDE);

```

When requests to a remote object are so important that they should never get queued behind any request to other remote objects in the same server application, private connects are a way to guarantee non-multiplexed connections between a client and a remote object. Private connections are set via the RT-CORBA PrivateConnectionPolicy policy. The code to set the policy is quite simple. This time we set the policy at the thread level.

```

CORBA::PolicyList policy_list; policy_list.length (1);
policy_list[0] = rt_orb->create_private_connection_policy ();
policy_current->set_policy_overrides (policy_list,
                                     CORBA::SET_OVERRIDE);

```

Protocol Plug-ins

Besides the explicit management of connections many applications in the DRE domain need to configure protocol properties. The protocol properties influence the sizes of the send and receive buffer as well as specific protocol attributes. The RT-CORBA specification incorporated this demand in the following way.

```

RTCORBA::ProtocolProperties_var tcp_properties =
    rtorb->create_tcp_protocol_properties (
        64 * 1024, /* send buffer */
        64 * 1024, /* recv buffer */
        false, /* keep alive */
        true, /* dont_route */
        true /* no_delay */);

```

Next, we configure the list of protocols to use.

```

//First, we create the protocol properties
RTCORBA::ProtocolList plist; plist.length (1);
plist[0].protocol_type = IOP::TAG_INTERNET_IOP; // IIOP
plist[0].trans_protocol_props = tcp_properties;

RTCORBA::ClientProtocolPolicy_ptr policy =
    rtorb->create_client_protocol_policy (plist);

```

Unfortunately, the configuration of protocol properties is only defined for TCP. This is mostly due to the lack of standardized PROTOCOL

PLUGINS that allow the exchange of the protocol layer underneath the GIOP (General-Inter-ORB-Protocol), which defines the messages between a client and server and how those are marshaled. Besides the most widely used TCP PROTOCOL PLUGIN (IIOP — Internet- Inter-ORB-Protocol), other plugins supporting UDP or ATM as protocol are conceivable. The Extensible Transport Framework specification, adopted in January 2004, addresses this. As soon as standardized PROTOCOL PLUGINS are available, the standardization of protocol properties for those plugins becomes possible.

Neither CORBA, nor RT-CORBA make use of any reflection techniques to track QoS properties. This is left to applications on top of CORBA. Frameworks, such as Quality Objects [Bbn02], implement QOS OBSERVER functionality; in Quality Objects the QOS OBSERVER is called QoS Tracker.

18 Appendix

Appendix A: Building Blocks for Component Infrastructures

In this appendix, we provide a brief overview of the component infrastructure patterns, presented in [VSW02]. These patterns are referenced in several places of our book. Component infrastructures are often implemented on top of a distributed object middleware. We have already provided a brief discussion of component infrastructures in the Chapter *Related Concepts, Technologies, and Patterns*; here we go into a bit more detail.

A *Component* encapsulates a well-defined piece of the overall application functionality. An application is assembled from collaborating components accessing each other through a well-defined *Component Interface*. Because the functionality of components and the way to access them is well-defined and self-contained, existing components can be reused in several applications. The interface is technically separated from the *Component Implementation* which can be exchanged without affecting clients.

The strict separation of interface and implementation allows the container to insert *Component Proxies* into the call chain between the *Client Proxy* and the *Component Implementation*. On behalf of the *Container*, these proxies handle technical concerns. The *Lifecycle Callback* interface of a *Component* is used by the *Container* to control the lifecycle of a component instance. This includes instantiating components, configuring instances, activating and passivating them over time, checking their state (running, standby, overload, error, ...) or restarting one in case of severe problems. Because all *Components* are required to have the same *Lifecycle Interface*, the *Container* can handle different types of components uniformly.

Usually, an *Application Server* provides different kinds of *Containers*. The different kinds are distinguished by the way they handle a compo-

nent's lifecycle, how they can be accessed, and how concurrency is handled. Examples include:

- *Service Components* are stateless entities and can be accessed concurrently. The container might provide *Instance Pooling* to make their usage more efficient.
- *Session Components* represent private, stateful, and client-dependent components. Typically, concurrency synchronization is not supported. *Passivation* might be provided by the *Container* to free resources used by temporarily unused instances.
- *Entity Components* represent persistent data entities. Persistence management as well as serialization of concurrent access is provided by the *Container*.

Annotations are used by the *Component* developer to declaratively specify technical concerns (i.e. which of a container's services are needed by a component and in which way). A *Component Context* is an interface passed to the *Component Implementation* that allows it to control some aspects of the container such as reporting an error and request shutdown or accessing *Managed Resources*.

A *Component* is not allowed to manage its own resources. It has to request access to *Managed Resources* from the container, allowing the *Container* to efficiently manage resources for the whole application (i.e. several *Components*). These resources also include access to other *Component Interfaces*.

All the resources a component instance needs to use at runtime must be declared in the *Annotations* to allow the *Container* to determine if a component can correctly run in a given context; that is, whether all *Required Resources* are actually available.

Invocation are transported from the client to the *Application Server* (and to the *Containers* within it) using a *Component Bus*. When operations are invoked on instances, the invocation might carry an additional *Invocation Context* that contains, in addition to operation name and parameters, data structures which the *Container* can use to handle the technical concerns (such as a transaction ID).

Last but not least, a *Component* is not just dropped into a *Container*, it has to be explicitly installed in it, allowing the container to decide if it

| can host the *Component* in a given environment (based on the *Annotations* and *Required Resources*).

Appendix B: Extending AOP Frameworks for Remoting

An important future trend for distributed object middleware is aspect-oriented programming (AOP). We have already provided a brief discussion of AOP in the remoting context in the Chapter *Related Concepts, Technologies, and Patterns*; in this appendix we explain the relationship in more detail.

In [Zdu03] a pattern language is described that explains how aspect composition frameworks are realized internally. In [Zdu04a] a projection of this pattern language to a number of popular aspect composition frameworks can be found, including AspectJ [KHH+01], Hyper/J [Tar03], JAC [PSDF01], JBoss AOP [Bur03], XOTcl [NZ00], Axis [Apa03], Demeter/DJ [OL01], and others. In this appendix we explain how these patterns can be combined with distributed object middleware.

Let us briefly introduce a few terms that have become quite well accepted in the AOP community (and that we use in the subsequent examples). These terms originate from the AspectJ [KHH+01] terminology. They describe the constituents of an *aspect* in a number of AOP approaches (note that there are other kinds of aspects as well):

- *Joinpoints* are specific, well-defined events in the control flow of the executed program.
- An *advice* is a behavior that is triggered by a certain event and that can be inserted into the control flow, when a specific joinpoint is reached. Advices allow one to transparently apply some behavior to a given control flow.
- *Pointcuts* are the glue between joinpoints and advices. A pointcut is a declaration of a number of joinpoints. These declarations are used by developers to tell the aspect composition framework at which joinpoints it has to apply the advices.
- *Introductions* change the structure of an object system. Typical introductions add methods or fields to an existing class or change the interfaces of an existing class. Introductions are not supported by all aspect composition frameworks.

An important application area for AOP are distributed object middleware systems, especially when used for distributed component

infrastructures. Compared to general purpose aspect composition frameworks, where the aspect can potentially be applied to all invocations (and other things, such as field accesses, class structures, etc.) within a system, aspects for distributed object middleware have a more limited scope: the aspects are applied only to the remote invocations and the remote objects. Thus we have the following choices, when we want to adopt an AOP solution within a distributed object middleware:

- We can use a general purpose aspect composition framework, such as AspectJ or Hyper/J, for adapting the respective invocations to the classes of the distributed object middleware and the remote objects.
- We can use an aspect composition framework specifically designed for remoting purposes. Existing examples are JAC or JBoss AOP. Both are also examples of server component infrastructures.
- We can extend a distributed object middleware to support AOP. Actually, this is not as much works as it sounds. Many distributed object middleware already support some adaptation techniques that are almost aspect-oriented. For instance, the systems in our technology projections, .NET Remoting, CORBA, and Web Service frameworks, as well as many other distributed object middleware, offer some support for INVOCATION INTERCEPTORS. As explained below, we can use INVOCATION INTERCEPTORS as a basic infrastructure to support AOP.

Below we explain examples for these three choices using the patterns from [Zdu03, Zdu04a]. First, let us briefly explain the most important patterns from this pattern language.

A Pattern Language for Implementing Aspect Composition Frameworks

All popular aspect composition frameworks implement the pattern *Indirection Layer*. Obviously, an aspect composition framework requires some way to trace the joinpoints specified by the pointcuts. An *Indirection Layer* traces all relevant invocation and structure information of a (sub-)system at runtime. It is a *Layer* [BMR+96] between the application logic and the instructions of the (sub-)system that should be traced. The general term “instructions” can refer to a whole

programming language, but it can also refer to a more specific instruction set - such as those instructions required to invoke remote objects. The *Indirection Layer* wraps all accesses to the relevant sub-system and should not be bypassed. It provides “hooks” to trace and manipulate the relevant information. Note that a distributed object middleware is already one variant of this pattern - it indirections all remote invocations to remote objects and cannot be bypassed by remote clients.

Two popular ways to add an *Indirection Layer* to a system are:

- The program representation (for instance, the source code or byte-code) can be manipulated (instrumented) using the patterns *Hook Injector*. That is, a semantically equivalent program variant is produced which sends all (relevant) invocations through the *Indirection Layer*. Here, the invocations can be manipulated at runtime. Except for programming languages with a dynamic object systems, this variant requires static instrumentation of the aspectized classes. The pattern *Parse Tree Interpreter* can be used to manipulate the source code of the classes to be aspectized. A *Parse Tree Interpreter* is applied at compile time. Alternatively, a *Byte Code Manipulator* can be used to manipulate a byte code representation of the program. The latter alternative has the advantage that it can be applied to third party code (where the source code is not available) and that it can be applied at load time. Both patterns, *Parse Tree Interpreter* and *Byte Code Manipulator*, are relatively easy to apply in compiled languages.
- Alternatively, the invocations can be intercepted at runtime and then manipulated, using a *Message Redirector*. A *Message Redirector* is a *Facade* [GHJV95] to the *Indirection Layer*. Clients do not access the *Indirection Layer* objects directly, but send symbolic (e.g. string-based) invocations to the *Message Redirector*. The *Message Redirector* dispatches these invocations to the respective method and object. This variant has the benefit that it allows for dynamic aspectization at runtime. In the context of distributed object middleware a further advantage is that the *Message Redirector* infrastructure is already present. The marshalled invocations can be used as symbolic invocation information, and the INVOKER (or sometimes the SERVER REQUEST HANDLER) can be used as a *Message Redirector*.

In most aspect composition frameworks the *Message Redirector* or *Hook Injector* is used to insert invocations to *Message Interceptors* (a general purpose variant of the pattern `INVOCATION INTERCEPTOR`). Because many distributed object middleware systems support `INVOCATION INTERCEPTORS`, we can reuse this architecture to realize AOP on top of it.

For an aspect and for the *Indirection Layer* of an aspect composition framework it is important to find out about the context of an invocation. This context is passed in an *Invocation Context*. Note that the *Invocation Context*, required for an aspect framework, is different to a remote `INVOCATION CONTEXT`. For aspects, we require information about the caller and callee scope, and the chain of aspects. A remote `INVOCATION CONTEXT` rather carries extra information, such as authentication information or session IDs. However, the remote `INVOCATION CONTEXT` can be extended to carry information for the AOP *Invocation Context*.

For the specification of pointcuts and applying them to joinpoints at runtime, we require information about the structures and relationships of the system. This information can be provided by the pattern *Introspection Options*. In the context of remoting that means, the distributed object middleware (or its programming language) should especially provide *Introspection Options* for the remote objects and their classes. Here, especially reflective information about the interface is required - something that is usually present on the `INVOKER` of a distributed object middleware.

If aspects should not be configured in the main programming language only, the patterns *Metadata Tags* and *Command Language* can be used to configure aspects either with metadata or using a programmatic configuration language. These patterns are especially used for pointcut definition and aspect configuration.

In the remainder of this Appendix we explain examples how these patterns are used to introduce AOP into a distributed object middleware.

Using AspectJ to Introduce Remoting Aspects

AspectJ is a popular, general purpose aspect language. In AspectJ, the aspects are described in an extension of the Java language, consisting of a set of additional instructions. As additional statements, AspectJ

introduces the aspect statement, as well as pointcuts (such as `call`, `target`, `args`, etc.) and advices (`before`, `after`, `around`). Advices are *Message Interceptors* that are executed, when the pointcut condition applies to a particular joinpoint. AspectJ uses a dynamic joinpoint model, but weaves the aspects statically using a *Parse Tree Interpreter* or a *Byte Code Manipulator*.

When the goal is to manipulate user-defined remote objects for which the source code is available, we can apply AspectJ aspects in a very similar way as for other local objects. However, this way it is hard to define generic aspects that can be applied for the complete distributed object middleware or all remote objects. If the code of the distributed object middleware is accessible, we can define aspects for the elements of the distributed object middleware. For instance, the following simplified aspect intercepts all invocations by a `INVOKER` class, and forwards them to an authentication class before they reach the `INVOKER`:

```
public aspect AuthenticationAspect {
    public pointcut requiresAuthenticationInfo():
        call(* Invoker.invoke(..));
    before() : requiresAuthenticationInfo(...) {
        Authentication.authenticate(...);
    }
}
```

AspectJ has a few potential problems in the area of remoting. Specifically AspectJ uses static weaving. In some usage scenarios, server applications cannot be shut down for instrumentation of classes. Therefore, some other solutions, described below, prefer load time or runtime aspect weaving. In AspectJ, runtime elements of aspects can be expressed using the dynamic jointpoint model (that is, using the `thisJoinPoint Invocation Context`). However, the classes must be statically aspectized at compile time. Note that this is especially a problem for long-running server applications. For many client applications static instrumentation is not a problem.

Java Aspect Components: JAC

JAC [PSDF01] is a framework for distributed aspect components in Java. In JAC, aspects can be deployed and un-deployed at runtime. To prepare the Java classes to be used with aspects, BCEL [Dah03] is used as *Byte Code Manipulator* to support aspectization at load time. BCEL

offers an API to access and manipulate the Java byte-code. This *Byte Code Manipulator* is used by the *Hook Injector* of JAC. The inserted hooks have the responsibility to indirect invocations into the JAC *Indirection Layer* that implements the JAC AOP features. There are three main features to support dynamic aspects in JAC: aspect components, dynamic wrappers, and domain-specific languages.

Aspect components are classes for which pointcuts can be defined to add before, after, or around behavior for base methods. In contrast to AspectJ, methods are specified with strings and looked up using *Introspection Options*. The pointcuts of the aspect components are used to invoke *Message Interceptors*, defined by dynamic wrappers. Dynamic wrappers can be seen as generic advice. Wrappers are ordered in wrapping chains. The methods of the wrapper have the aspect's *Invocation Context* as a parameter. It contains information about the wrapped object, method, arguments, and wrapping chain of the invocation. The aspects can define a *Command Language*, called a domain-specific language in JAC, to configure the pointcuts of the aspects. In the context of remoting, this feature can especially be used to configure the INVOCATION CONTEXT and the INVOKER, as for instance access rights and authentication data.

JBoss Aspect-Oriented Programming

The JBoss Java application server contains a stand-alone aspect composition framework [Bur03]. It is similar to JAC, but there are some interesting differences in the design decisions.

The *Hook Injector* of JBoss AOP also allows for load time instrumentation of “advisable” classes. Internally, Javassist [Chi03] is used as a *Byte Code Manipulator*. In contrast to BCEL, it provides a source-level abstraction of the byte-code. An advice is implemented as a *Message Interceptor*. All *Message Interceptors* must implement the following interface:

```
public interface Interceptor {
    public String getName();
    public InvocationResponse invoke(Invocation invocation)
        throws Throwable;
}
```

The name returned by `getName` is a symbolic interceptor name. `invoke` is a callback method to be called whenever the advice is to be

executed. The parameter of the type *Invocation* and the return type *InvocationResponse* implement the *Invocation Context*. All pointcut definitions are given using XML-based *Metadata Tags*, for instance:

```
<interceptor-pointcut class="mypackage.MyClass">
  <interceptors>
    <interceptor class="TracingInterceptor" />
  </interceptors>
</interceptor-pointcut>
```

This way the class loader knows which classes it has to instrument at load time, and the aspects can be configured easily. JBoss AOP also offers a programmatic API to compose the interceptors for instrumented classes at runtime.

Extending Axis Handler Chains for AOP

Many distributed object middleware systems implement some form of *INVOCATION INTERCEPTORS*. For instance, in all three technology projections, .NET Remoting, CORBA, and Web Services, some support was available. This infrastructure can be used to build a simple aspect framework from scratch. Let us consider the Axis handler chains that we have discussed already in the Web Services technology projection. These can be used as *Message Interceptors* for the aspect framework.

Besides the *Message Interceptors*, Axis provides an *Invocation Context* (in Axis called the *MessageContext*), usable for AOP purposes. Also, aspect configuration is possible in a way similar to JBoss AOP, because handler chains can be flexibly configured using *Metadata Tags* in the XML deployment descriptors. In contrast to the solutions explained before, Axis uses a *Message Redirector* based architecture. The remote object classes are not instrumented at runtime, but instead the *Message Redirector* (here: in the *SERVER REQUEST HANDLER*) indirects the invocation into the handler chain and finally to the *INVOKER*.

Note that this infrastructure alone is not an AOP framework. What is missing is a way to specify pointcuts and apply these. This can be done quite easily by hand, because all necessary information is provided to the *Message Interceptors* in the *Invocation Context*. Of course, with this information only one type of joinpoints can be specified: remote invocations.

A simple implementation variant is to make invoke a *Template Method* [GHJV95] defined for an abstract class `AspectHandler`. All aspect handlers inherit from this class and implement the method `applyAspect`. This method is only called, if there is a pointcut for the current aspect and message context defined.

```
public abstract class AspectHandler extends BasicHandler {
    public boolean checkPointcuts(MessageContext msgContext) {
        // check whether pointcuts apply and return true/false
        ...
    }
    public void invoke(MessageContext msgContext) throws AxisFault {
        if (checkPointcuts(msgContext) == true) {
            applyAspect(msgContext);
        }
    }
    abstract public void applyAspect(MessageContext msgContext);
}
```

Note that the pointcuts are defined at two levels (similar to the instrumentation in JAC and JBoss): the aspectized classes are defined by the Axis handler chains - as CONFIGURATION GROUPS. The implementation of the method `checkPointcuts` determines which joinpoints are intercepted. Using the Java Reflection API we can further implement *Introspection Options* for the remote objects - to support non-invasive pointcuts. Also, we need some way to express the pointcuts - if we do not want to implement pointcuts programmatically, we can define XML-based *Metadata Tags*, for instance.

19 References

- ACD+03 T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. *Business Process Execution Language for Web Services, Version 1.0*. 2003. <http://www.ibm.com/developerworks/webservices/library/ws-bpel>
- ADH+02 B. Atkinson, G. Della-Libera, S. Hada, M. Hondo, P. Hallam-Baker, J. Klein, B. LaMacchia, P. Leach, J. Manferdelli, H. Maruyama, A. Nadalin, N. Nagaratnam, H. Prafullchandra, and J. Shewchuk, D. Simon. *Web Services Security (WS-Security)*. 2002. <http://www-106.ibm.com/developerworks/webservices/library/ws-secure/>
- AIS+77 C. Alexander, S. Ishikawa, M. Silverstein, M. Jakobson, I. Fiksdahl-King, and S. Angel. *A Pattern Language – Towns, Buildings, Construction*. Oxford Univ. Press, 1977.
- ALR01 A. Avizienis, J.-C. Laprie, and B. Randell. *Fundamental Concepts of Dependability*. Research Report N01145, LAAS-CNRS, April 2001.
- Ans04 ANSI Accredited Standards Committee (ASC) X12. *EDI*. 2004. <http://www.x12.org/>
- Apa01 Apache Software Foundation. *Web Services Invocation Framework (WSIF)*. 2001. <http://ws.apache.org/wsif/>
- Apa02 Apache Software Foundation. *Web services invocation framework (WSIF)*. 2002. <http://ws.apache.org/wsif/>
- Apa03 Apache Software Foundation. *Web Services - Axis*. 2003. <http://xml.apache.org/axis/>
- Arp93 ARPA Knowledge Sharing Initiative. *Specification of the KQML agent-communication language*. ARPA

- Knowledge Sharing Initiative, External Interfaces Working Group, July 1993.
- Aut04 The Autosar Consortium. *Autosar - Automotive Open System Architecture*. 2004. <http://www.autosar.org>
- Bar02 T. Barnaby. *Distributed .NET Programming in C#*. APress, 2002.
- BBF+02 M. Bartel, J. Boyer, B. Fox, B. LaMacchia, and E. Simon. *XML-Signature Syntax and Processing*. 2002. <http://www.w3.org/TR/xmlsig-core/>
- BBL02 M. Baker, R. Buyya, and D. Laforenza. Grids and Grid technologies for wide-area distributed computing. In: *Software: Practice and Experience*, 32(15), Wiley, 2002.
- Bbn02 BBN Technologies. *Quality Objects (QuO)*. 2002. <http://quo.bbn.com/>
- BCH+03 D. Box, F. Curbera, M. Hondo, C. Kale, D. Langworthy, A. Nadalin, N. Nagaratnam, M. Nottingham, C. von Riegen, and J. Shewchuk. *Web Services Policy Framework (WSPolicy)*. 2003. <http://www.ibm.com/developerworks/library/ws-policy>
- BCK98 L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.
- Bea03 BEA. *BEA Tuxedo 8.1*. 2003. <http://www.bea.com/products/tuxedo/index.shtml>
- BEK+00 D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. *Simple object access protocol (SOAP) 1.1*. 2000. <http://www.w3.org/TR/SOAP/>
- BHC+03 D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard. *Web Services Architecture*. W3C Working Draft 8, August 2003. <http://www.w3.org/TR/2003/WD-ws-arch-20030808/>
- BM98 M. Breugst and T. Magedanz. Mobile agents - enabling technology for active intelligent network implementation. *IEEE Network Magazine*, 12(3):53-60, August 1998.

- BMR+96 F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *Pattern-Oriented Software Architecture - A System of Patterns*. J. Wiley and Sons Ltd., 1996.
- BN84 A. Birrell and B. Nelson. Implementing Remote Procedure Calls. *ACM Transaction on Computer Systems*, 2(1), February 1984.
- Bor04 Borland Inc., *Janeva - Platform interoperability for the Enterprise*, <http://www.borland.com/janeva/>
- Bpm02 Business Process Management Initiative. *Business Process Modeling Language (BPML)*. November 2002. <http://www.bpmi.org>
- BPS98 T. Bray, J. Paoli, and C.M. Sperberg-McQueen. *Extensible markup language (XML) 1.0*. 1998. <http://www.w3.org/TR/1998/REC-xml-19980210>
- BTN00 J. J. Barton, S. Thatte, and H. F. Nielsen. *SOAP Messages with Attachments*. W3C Note 11, December 2000. <http://www.w3.org/TR/SOAP-attachments>
- Bur03 B. Burke. *JBoss aspect oriented programming*. 2003. <http://www.jboss.org/developers/projects/jboss/aop.jsp>
- CCC+02 F. Cabrera, G. Copeland, B. Cox, T. Freund, J. Klein, T. Storey, and S. Thatte. *Web Services Transaction (WS-Transaction)*. August 2002. <http://www.ibm.com/developerworks/library/ws-transpec/>
- CCC+03 L. Cabrera, G. Copeland, W. Cox, M. Feingold, T. Freund, J. Johnson, C. Kaler, J. Klein, D. Langworthy, A. Nadalin, D. Orchard, I. Robinson, J. Shewchuk, and T. Storey. *Web Services Coordination (WS-Coordination)*. September 2003. <http://www-106.ibm.com/developerworks/library/ws-coor/>
- CCM+01 E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. *Web services description language (WSDL) 1.1*. 2001. <http://www.w3.org/TR/wsdl>

- CDK94 G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems - Concepts and Design*. Addison-Wesley, Reading, MA, 1994.
- Chi03 S. Chiba. *Javassist*. 2003. <http://www.csg.is.titech.ac.jp/~chiba/javassist/>
- Cia04 CAN in Automation (CiA). *Controller Area Network*. 2004. <http://www.can-cia.de>
- Cop04 James O. Coplien. *A Pattern Definition*. 2004. <http://hillside.net/patterns/definition.html>
- CTV+98 P. Ciancarini, R. Tolksdorf, F. Vitali, D. Rossi, and A. Knoche. Coordinating Multiagent Applications on the WWW: a Reference Architecture. *IEEE Transactions on Software Engineering*, 24(5): 362-375, 1998.
- CV02 E. Chtcherbina and M. Völter. Peer to Peer Systems - EuroPLoP 2002 Focus Group Results. In: *Proceedings of EuroPlop 2002*, Irsee, Germany, July 2002. <http://www.voelter.de/data/pub/P2PSystems.pdf>
- Dah03 M. Dahm. *The bytecode engineering library (BCEL)*. 2003. <http://jakarta.apache.org/bcel/>
- DDG+02 G. Della-Libera, B. Dixon, P. Garg, P. Hallam-Baker, M. Hondo, C. Kaler, H. Maruyama, A. Nadalin, N. Nagaratnam, A. Nash, R. Philpott, H. Prafullchandra, J. Shewchuk, D. Simon, E. Waingold, and R. Zolfonoon. *Web Services Trust Language (WS-Trust)*. 2002. <http://www.ibm.com/developerworks/library/ws-trust/index.html>
- DHH+02 G. Della-Libera, P. Hallam-Baker, M. Hondo, T. Janczuk, C. Kaler, H. Maruyama, N. Nagaratnam, A. Nash, R. Philpott, H. Prafullchandra, J. Shewchuk, E. Waingold, and R. Zolfonoon. *Web Services Security Policy (WS-SecurityPolicy)*. 2002. <http://www.ibm.com/developerworks/library/ws-secpol/index.html>
- DL03 P. Dyson and A. Longshaw. Patterns for High-Availability Internet Systems. In: *Proceedings of EuroPlop 2003*, Irsee, Germany, June 2003.

- DL04 P. Dyson and A. Longshaw. *Architecting Enterprise Solutions: Patterns for High-Capability Internet-based Systems*, John Wiley & Sons, 2004.
- Dub01 O. Dubuisson. *ASN.1 - Communication between heterogeneous systems*. Morgan Kaufmann, 2001.
- Emm00 W. Emmerich. *Engineering Distributed Objects*. Wiley & Sons, 2000.
- ERI+02 D. Eastlake, J. Reagle, T. Imamura, B. Dillaway, and E. Simon. *XML Encryption Syntax and Processing*. W3C Recommendation. 10 December 2002. <http://www.w3.org/TR/xmlenc-core/>
- FF00 R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In: *OOPSLA Workshop on Advanced Separation of Concerns*, Minneapolis, USA, October 2000.
- FGM+97 R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. *Hypertext Transfer Protocol -- HTTP/1.1*. RFC 2068. January 1997
- FHA99 E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces - Principles, Patterns, and Practice*. Addison-Wesley, 1999.
- FHF+02 W. Ford, P. Hallam-Baker, B. Fox, B. Dillaway, B. LaMacchia, J. Epstein, and J. Lapp. *XML Key Management Specification (XKMS)*. W3C Note 30 March 2001. <http://www.w3.org/TR/xkms/>
- Fip02 FIPA. *Agent Communication Language Specifications*. 2002. <http://www.fipa.org/repository/aclspecs.html>
- Fow96 M. Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, 1996.
- Fow03 M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.
- FPV99 A. Fuggetta, G. P. Picco, and G. Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5): 342-361, May 1998.

- FKT01 I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputer Applications*, 15(3), 2001.
- FV00 A. Fricke and M. Voelter. *SEMINARS – A Pedagogical Pattern Language on how to Teach Seminars Efficiently*. 2000. <http://www.voelter.de/publications/seminars.html>
- GCCC85 D. Gelernter, N. Carriero, S. Chandran, and S. Chang. Parallel programming in Linda. In: *Proceedings of the 1985 International Conference on Parallel Processing*, pages 255-263, 1985.
- GCK+02 R. S. Gray, G. Cybenko, D. Kotz, R. A. Peterson, and D. Rus. D'Agents: Applications and Performance of a Mobile-Agent System. *Software - Practice and Experience*, 32(6):543-573, May, 2002.
- Gel99 D. H. Gelernter. *Machine Beauty: Elegance and the Heart of Technology*. Basic Books, 1999.
- GHJV95 E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.
- Gnu01 Gnutella Homepage, 2001. <http://www.gnutella.com>
- GNZ01 M. Goedicke, G. Neumann, and U. Zdun. Message Redirector. In: *Proceedings of EuroPlop 2001*, Irsee, Germany, July 2001.
- Gra78 J. N. Gray. Notes on Database Operating Systems. *Operating Systems: An Advanced Course. Lecture Notes in Computer Science*, 60: 393-481, Springer-Verlag, 1978.
- Gri97 R. Grimes. *Professional DCOM Programming*. Wrox Press Inc., 1997.
- Gri03 Grid Computing Info Centre. 2003. <http://www.gridcomputing.com>
- Gro01 W. Grosso. *Java RMI*. O'Reilly & Associates, 2001.

- GT03 B. Gröne and P. Tabeling. A System of Patterns for Concurrent Request Processing Servers. In: *Proceedings of VikingPLOP 2003*, Bergen, Norway, 2003.
- Hen98 K. Henney. Counted Body Techniques. In: *Overload 25*, April 1998. http://boost.org/more/count_bdy.htm
- Hen01 K. Henney. C++ Patterns: Reference Accounting. In: *Proceedings of EuroPLOP 2001*, Irsee, Germany, July 2001.
- Her03 W. Herzner. Message Queues – Three Patterns for Asynchronous Information Exchange. In: *Proceedings of EuroPlop 2003*, Irsee, Germany, June 2003.
- HV99 M. Henning and S. Vinoski. *Advanced CORBA Programming with C++*. Addison-Wesley, 1999.
- HW03 G. Hohpe and B. Woolf. *Enterprise Integration Patterns*. Addison-Wesley, October 2003.
- Ibm00 IBM. *TSpaces*. 2000. <http://www.almaden.ibm.com/cs/TSpaces/>
- Ibm03 IBM. *CICS (Customer Information Control System) Family*. 2003. <http://www.ibm.com/software/http/cics/>
- Ibm04 IBM. *WebSphere MQ Family*. 2004. <http://www-306.ibm.com/software/integration/mqfamily/>
- Jen02 R. Jennings. Monitor Web Service Performance. *XML & Web Services Magazine*, 10, 2002. http://www.fawcette.com/xmlmag/2002_10/online/webservices_rjennings_10_30_02/
- Jin03 The Jini Community. *Jini Community Homepage*. 2003. <http://www.jini.org/>
- KHH+01 G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and G. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59-65, Oct 2001.
- KJ04 M. Kircher and P. Jain. *Pattern-Oriented Software Architecture - Resource Mangement*. Wiley & Sons, 2004.
- KLM+97 G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented

- programming. In: *Proceedings of ECOOP97*, Finland, LCNS 1241, Springer-Verlag, June 1997.
- Kop97 H. Kopetz. *Real-Time Systems. Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Boston, Dordrecht, London, 1997.
- Lap85 J.-C. Laprie. Dependability: Basic Concepts and Terminology. In: *Proceedings of the 15th International Symposium on Fault-Tolerant Computing (FTCS-15)*, Michigan, USA, June 1985.
- Lea99 D. Lea. *Concurrent Programming in Java - Design Principles and Patterns*. Addison-Wesley, Reading, Mass., 1990.
- Lem97 M. Lemmon. *Berkeley Sockets*. Lecture Notes, 1997. <http://www.nd.edu/~lemmon/courses/UNIX/l6/node3.html>
- LO98 D. B. Lange and M. Oshima. *Programming and deploying Java mobile agents with Aglets*. Addison Wesley, 1998.
- Low03 J. Lowy. Microsoft. Decouple Components by Injecting Custom Services into Your Object's Interception Chain. *MSDN Magazine*, March 2003. <http://msdn.microsoft.com/msdnmag/issues/03/03/ContextsinNET/default.aspx>
- Lyu95 M.R. Lyu (ed.). *Software Fault Tolerance*. John Wiley & Sons Ltd., 1995.
- Mae87 P. Maes. *Computational Reflection*. Technical Report TR-87-2, VUB AILAB, 1987.
- Maf96 S. Maffeis. The Object Group Design Pattern. In: *Proceedings of the 2nd Conference on Object-Oriented Technologies and Systems*, Toronto, Canada, June 1996.
- Mic02 Microsoft. *Information on Microsoft COM+ technologies*. 2002. <http://www.microsoft.com/com/tech/COMPlus.asp>
- Mic03 Microsoft. *Microsoft Transaction Server (MTS)*. 2003. <http://www.microsoft.com/com/tech/MTS.asp>

- Mic04a Microsoft. *MSMQ Microsoft Message Queue Server*. 2004. <http://www.microsoft.com/msmq/default.htm>
- Mic04b Microsoft. *Indigo*. 2004. <http://msdn.microsoft.com/Longhorn/understanding/pillars/Indigo/default.aspx>
- Mic04c Microsoft. *Data Transfer Object*. 2004. <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpatterns/html/DesDTO.asp>
- Min03 The Mind Electric. *GLUE*. 2003. <http://www.themindelectric.com/glue/>
- Moc87 P. V. Mockapetris. *RFC 1035: Domain names - implementation and specification*. Nov. 1987.
- Nob97 James Noble. Basic Relationship Patterns. In: *Proceedings of the European Conference on Pattern Languages of Program Design (EuroPLOP)*. Irsee, Germany, 1997. <http://www.riehle.org/europlop-1997/p11final.pdf>
- NZ00 G. Neumann and U. Zdun. XOTcl, an object-oriented scripting language. In: *Proceedings of Tcl2k: The 7th USENIX Tcl/Tk Conference*, pages 163–174, Austin, Texas, USA, February 2000. <http://www.xotcl.org>
- Oas02 Organization for the Advancement of Structured Information Standards (OASIS). *UDDI Version 3.0*. Oct 2002. <http://www.uddi.org/specification.html>
- Oas03a Organization for the Advancement of Structured Information Standards (OASIS). *Security Assertion Markup Language (SAML). Version 1.1*. September 2003. <http://www.oasis-open.org/committees/security/>
- Oas03b Organization for the Advancement of Structured Information Standards (OASIS). *eXtensible Access Control Markup Language Security Assertion Markup (XACML). Version 1.0*, February 2003. <http://www.oasis-open.org/committees/xacml>
- Obj97 ObjectSpace. *Voyager core package technical overview*. Version 1.0. ObjectSpace, Inc., December 1997.

- OG03 J. Oberleitner and T. Gschwind. Transparent Integration of CORBA and the .NET Framework. In: *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, Springer LNCS, Volume 2888, 2003.
- OL01 D. Orleans and K. Lieberherr. DJ: Dynamic adaptive programming in Java. In: *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns*, pages 73-80, Kyoto, Japan, Sep 2001.
- OLW98 J. Ousterhout, J. Levy, and B. Welch. The Safe-Tcl Security Model. In: G. Vigna, editor, *Mobile Agents and Security*, LNCS vol. 1419. Springer, 1998.
- Omg04a Object Management Group. *Common Object Request Broker Architecture (CORBA/IIOP)*. 2004. <http://www.omg.org/technology>
- Omg04b Object Management Group. *Object Transaction Service*. 2004. <http://www.omg.org/technology>
- Ope91 Open Software Foundation, *DCE Application Development Guide*, Revision 1.0, Cambridge, MA, 1991.
- Ope97 The Open Group. *Universal Unique Identifier*. 1997. <http://www.opengroup.org/onlinepubs/9629399/apdxa.htm>
- Ope01 OpenNap. *The Open Source Napster Server*. 2001. <http://opennap.sourceforge.net/>
- Ope03 The Open Group. *Transaction Processing*. 2003. <http://www.opengroup.org/products/publications/catalog/tp.htm>
- Ped04 The Pedagogical Patterns Project. 2004. <http://www.pedagogicalpatterns.org>
- PS97 H. Peine and T. Stolpmann. The architecture of the Ara platform for mobile agents. In: *Proceedings of the First International Workshop on Mobile Agents*, volume 1219 of Lecture Notes in Computer Science, pages 50-61, Springer-Verlag, Berlin, Germany, April 1997.
- PSDF01 R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: a flexible framework for AOP in Java. In: *Reflection*

- 2001: *Meta-level Architectures and Separation of Cross cutting Concerns*. pages 1-24, Kyoto, Japan, Sep 2001.
- Qui03 M. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, 2003.
- Ram02 I. Rammer. *Advanced .NET Remoting*. APress, 2002.
- Rie98 Dirk Riehle. Bureaucracy. In: *Pattern Languages of Program Design 3*, Editor: Robert Martin, Dirk Riehle, and Frank Buschmann, Addison-Wesley, 1998.
- Rog95 E. M. Rogers. *Diffusion of Innovations*. 4th Ed, Free Press, 1995.
- Rox03 Roxio, Inc. The Napster Homepage. 2003. <http://www.napster.com>
- RSB+98 D. Riehle, W. Siberski, D. Bäumer, D. Megert, and H. Züllighoven. Serializer. In: *Pattern Languages of Program Design 3*. Editor: R. Martin, D. Riehle, F. Buschmann. Reading, Massachusetts: Addison-Wesley, 1998.
- Sar02 T. Saridakis. A System of Patterns for Fault Tolerance. In: *Proceedings of EuroPlop 2002*. Irsee, Germany, July 2002.
- Sar03 T. Saridakis. Design Patterns for Fault Containment. In : *Proceedings of EuroPlop 2003*. Irsee, Germany, June 2003.
- Set03 Seti@Home. *The Search for Extraterrestrial Intelligence*. 2003. <http://setiathome.ssl.berkeley.edu/>
- Sev03 D. Sevilla. *The CORBA & CORBA Component Model (CCM) Page*. <http://ditec.um.es/~dsevilla/ccm/>
- Sor02 K. E. Sorensen. Sessions. In: *Proceedings of EuroPlop 2002*. Irsee, Germany, July 2002.
- SSRB00 D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture - Patterns for Concurrent and Distributed Objects*. Wiley and Sons Ltd., 2000.

- Ste98 R. Stevens. *UNIX Network Programming*. Prentice Hall. 1998.
- Sta00 M. Stal. *Activator Pattern*. <http://www.stal.de/articles.html>, 2000.
- Sta03 M. Stal. *Distributed .NET Tutorial*. http://www.stal.de/Downloads/OOP2003/oop_distrnet.pdf
- Sun88 Sun Microsystems. *RPC: Remote Procedure Call Protocol Specification*. Tech. Rep. RFC-1057, Sun Microsystems, Inc., June 1988.
- Sun03a Sun Microsystems. *Enterprise Java Beans Technology*. 2003. <http://java.sun.com/products/ejb/>
- Sun03b Sun Microsystems. *Project JXTA*. 2003. <http://www.jxta.org>
- Sun04a Sun Microsystems. *Core J2EE Patterns - Transfer Object*. 2004. <http://java.sun.com/blueprints/corej2eepatterns/Patterns/TransferObject.html>
- Sun04b Sun Microsystems. *Java API for XML-Based RPC (JAX-RPC)*. 2004. <http://java.sun.com/xml/jaxrpc/>
- Sun04c Sun Microsystems. *Java Message Service (JMS)*. 2004. <http://java.sun.com/products/jms/>
- Swi02 SWIFT (ed.): *Annual Report 2002*. La Hulpe, Belgium, 2002. http://www.swift.com/temp/4129/43081/ar_2k2_full_color
- Tar03 P. Tarr. *Hyper/J*. 2003. <http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm>
- Tib04 Tibco. *Messaging Solutions*. 2004. http://www.tibco.com/software/enterprise_backbone/messaging.jsp
- Tra00 Transarc. *Encina*. 2000. <http://www.transarc.com>
- TS02 A. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2002.
- Upn03 UPNP Forum. *Universal Plug and Play*. 2003. <http://upnp.org/>

- Vin02a S. Vinoski. Toward Integration Column: Middleware Dark Matter. *IEEE Internet Computing*. Sep-Oct 2002.
- Vin02b S. Vinoski. Toward Integration Column: Chain of Responsibility. *IEEE Internet Computing*. Nov-Dec 2002.
- Vin03 S. Vinoski. Toward Integration Column: Integration With Web Services *IEEE Internet Computing*. Nov-Dec 2003.
- Voe03 M. Voelter. *Small Components - A generative component infrastructure for embedded systems*. 2003. <http://www.voelter.de/data/pub/SmallComponents.pdf>
- VSW02 M. Voelter, A. Schmid, and E. Wolff. *Server Component Patterns*. John Wiley and Sons, 2002.
- Win99 D. Winer. *XML-RPC Specification*. 1999. <http://www.xmlrpc.com/spec>
- W3C04 W3C. *Resource Description Framework (RDF)*. 2004. <http://www.w3.org/RDF/>
- WWWK94 J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. *A note on distributed computing*. Technical Report TR94 -29, Sun Microsystems Laboratories, Inc., November 1994.
- Zdu03 U. Zdun. Patterns of tracing software structures and dependencies. In: *Proceedings of EuroPlop 2003*. Irsee, Germany, June 2003.
- Zdu04a U. Zdun. Pattern Language for the Design of Aspect Languages and Aspect Composition Frameworks. *IEE Proceedings Software*. 151(2): 67- 83, 2004.
- Zdu04b U. Zdun. Loosely Coupled Web Services in Remote Object Federations. In: *Proceedings of 4th International Conference on Web Engineering (ICWE 2004)*. Munich, Germany, 2004.
- Zdu04c U. Zdun. *Leela*. <http://leela.sourceforge.net>, 2004.
- Zer03 ZeroC Software. *The Internet Communications Engine*. 2003. <http://www.zeroc.com/ice.html>
- ZVK03 U. Zdun, M. Völter, and M. Kircher. Design and Implementation of an Asynchronous Invocation Framework for Web Services. In: *Proceedings of*

*International Conference on Web Services Europe
(ICWS-Europe 2003)*, Erfurt, Germany, pages 64-78, 2003.