Remoting Patterns - A Systematic Approach for Design Reuse of Distributed Object Middleware Solutions

Markus Völter
voelter
Ingenieurbüro für Softewaretechnologie
Germany
voelter@acm.org

Michael Kircher
Siemems AG
Corporate Technology
Software and System Architectures
Germany
michael@kircher-schwanninger.de

Uwe Zdun
New Media Lab
Department of Information Systems
Vienna University of Economics
Austria
zdun@acm.org

In many situations the developers or architects of a distributed system require a deep understanding of the middleware they use. We argue that patterns and pattern languages are a practical and useful means to convey this knowledge. Unfortunately, a comprehensive pattern language, explaining how to effectively use, extend, integrate, customize, or build distributed object middleware solutions was missing. Therefore, we propose a pattern language of remoting patterns as a systematic way to reuse software models, designs, and implementations in the area of distributed object middleware. This pattern language has rich dependencies to other patterns and pattern languages from related domains, such as networking, concurrency, resource management, server components, security, availability, scalability, fault tolerance, and aspect-orientation.

Introduction

When developing, maintaining, or evolving a complex distributed system often the only source of concrete design and implementation knowledge about the system are the experiences of the developers and the source code itself. For many systems, only a few 'gurus' have a thorough understanding of the system and the prevalent development practices used to build it. This, however, means that the efforts necessary to acquire or evolve this knowledge are very high for people who are not among the experts for the system. This problem becomes extremely pressing, for instance, when the system experts leave the company.

Even though a large number of approaches exist that document *how* a system is designed, not many approaches also explain *why* a system is designed in a certain way. As outlined by Schmidt and Buschmann [SB03], patterns and middleware are popular techniques for coping with these challenges. *Patterns* provide reusable design knowledge in form of proven solutions to recurring software problems in a particular context or domain. *Middleware* allows to reuse a piece of software that hides the details of low-level APIs, such as those of operating systems, network protocol stacks, and databases. Today, distributed object middleware belongs to the basic elements in the toolbox when developing distributed systems. Popular examples of distributed object middleware systems are CORBA, Web Services, DCOM, Java RMI, and .NET Remoting.

Schmidt and Buschmann argue that patterns and middleware complement each other to enhance the *systematic reuse* of successful software models, designs, and implementations that have already been developed and tested [SB03]. This consideration does not only apply for systems built on top of a middleware, but also for situations in which an understanding of the inner workings of the middleware is required. Examples of such situations are:

- For *using* a distributed object middleware in an effective manner, developers need to understand the concepts and inner workings of the middleware implementation well. Even though different middleware systems use different remoting abstractions, terminologies, implementation language concepts, and so forth, they share many concepts. Understanding these common concepts also helps to switch from one middleware system to another.
- In some situations a middleware systems needs to be *extended* with additional functionality. Consider, for instance, the middleware of choice does not support a security feature that is required for a distributed application. Then the developers need to implement this feature as an extension to the middleware system.
- Sometimes it is necessary to *integrate* different middleware systems. Just consider distributed applications that are developed independently in different departments of a company or in different companies. If these applications need to work together, the middleware systems used by the applications need to be integrated.
- More rarely developers need to *customize* a distributed object middleware, or even *build* it from scratch. For instance, in the DRE (distributed, real-time, and embedded) domain systems have tight constraints regarding memory consumption, performance, and real-time communication. If no suitable middleware product exists or all available products turn out to be inappropriate, the developers must customize one of the existing solutions, if possible, or otherwise develop their own solution.

In such situations it is essential to understand the inner workings of proven approaches and technologies in the field of distributed object middleware. In this article, we propose a pattern language as a systematic approach to provide knowledge about the middleware's inner workings, and how to use, extend, integrate, customize, or even build distributed object middleware.

Up until now, there has been no comprehensive pattern language that explains the inner workings of distributed object middleware. Existing patterns have explained specific aspects of middleware implementations - see for instance [SSRB00, Lea99, GT03] - or how to build higher-level systems on top of a middleware, as for instance the server component patterns in [VSW02]. The book *Remoting Patterns* [VKZ04], of which we describe the key patterns in this article, provides a glue between those other patterns and the middleware layer to leverage the systematic development of distributed system based on patterns.

This article is structured as follows: First we explain the specific kind of middleware we concentrate on - communication middleware in general and distributed object middleware in specific - to position this article. Next, we briefly introduce patterns and pattern languages. Then we explain the Remoting Patterns [VKZ04] - which are the main focus of this article. Finally, we conclude with the relationships of the Remoting Patterns to other patterns and pattern languages.

Communication Middleware

Distributed systems can be built directly on top of low-level network protocols, for instance using communication based on TCP/IP sockets [Ste98]. But this means that developers have to handle all details of low-level network programming. As a result, such systems are usually not easy to *scale* and are rather *cumbersome* and *error prone* to use by developers. Moreover, they are hard to *maintain* and *change*, and do not provide *transparency* of the distributed communication.

As a solution, typically a *Communication Middleware*, or simply *Middleware,* is used as an additional software layer. The main task of the middleware is to hide the heterogeneity of the underlying platforms and provide transparency of distributed communication for the developers. That is, a remote invocations should be made as similar as possible to local invocations. However, full transparency is not possible: remote invocations always introduce new kinds of errors, latency, and so forth.

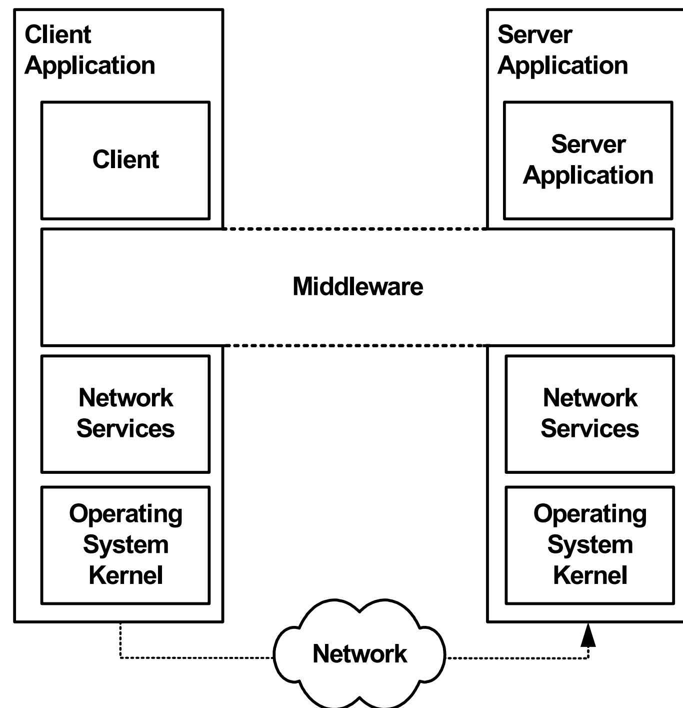

Figure 1. Communication Middleware

The middleware layer is typically implemented on top of the network services, offered by the operating system. The application layer hosting the client and server application is usually not allowed to bypass the middleware layer in order to access low-level network services directly. That means the middleware hides the heterogeneity of the underlying platforms from application developers.

There are a number of different remoting styles used in today's middleware systems, including systems based on *remote procedure calls* (RPC), *messages*, *shared repositories*, or *streams* of data. In this article we focus on middleware using object-oriented variants of the RPC style. However, all of the mentioned remoting styles can be used to implement the others; thus most of the patterns presented in this article are also relevant for systems implementing one of the other remoting styles.

It is important to understand that there are many other remoting styles that are based on the basic remoting styles. Examples of such higher-level remoting styles are code mobility, peer-to-peer (P2P) systems, remote evaluation, GRID computing, publish/subscribe systems, transaction processing, and many others. Note that these 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 neither have to deal with the internal RPC mechanisms, nor with the naming service used for ad hoc lookup of peer services.

Patterns and Pattern Languages

Over the past couple of years, patterns have become accepted as a mainstream software development technique. The most popular patterns are those for software design, for example the Gang-of-Four book [GHJV95] and POSA2 [SSRB00], and those on software architecture, for example POSA1 [BMR+96]. In addition, the patterns community has documented patterns for analysis

[Fow96], patterns for non-IT topics such as organizational or pedagogical patterns [PPP], and many more.

For this article we want to use the following definition of patterns by Jim Coplien, on the Hillside web-site [HS], which summarizes the longer definition by Christopher Alexander [AIS+77]:

> *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.*

A single pattern describes one solution to a particular, recurring problem. However, 'real big problems' usually cannot be described in one, single pattern. 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.
- F*amilies of patterns* are collections of patterns that solve the same general problem.
- C*ollections* or *systems of patterns* comprise several patterns from the same domain or problem area.
- P*attern languages* are the most powerful form of combining patterns . 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. Pattern languages do not only specify solutions to specific problems, but also describe a way to create a 'whole', the overall goal of the pattern language.

The patterns described in this article form a pattern language in the domain of remoting. That is, they explain how distributed object middleware systems work.

Remoting Patterns - Pattern Language Overview

In this section, we present the Remoting Patterns, as an comprehensive overview. Refer to [VKZ04] for more details on the pattern language.

Broker Architecture of a Middleware and Basic Remoting Patterns

The pattern BROKER[1] first described in POSA1 [BMR+96] is, from our perspective, a compound pattern that is typically implemented using a number of patterns from the Remoting Pattern language. The BROKER pattern addresses the problem that distributed software system developers face many challenges that do not arise in single-process software. One main challenge is the unreliable communication across 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 might loose their primary focus, to develop distributed applications that solve the application problems well.

The BROKER pattern separates the communication functionality of a distributed system from its application functionality by isolating all communication related concerns in a BROKER. A BROKER hides and mediates all communication between the objects or components of a system. Local BROKERS on client side and server side enable the exchange of requests and responses between the client and the remote object. 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

1. We highlight pattern names from the Remoting Patterns book [VKZ04] in SMALLCAPS font.

the transformation of requests and responses - from programming-language native data types into a byte array that can be sent over the wire. Figure 2 shows this typical BROKER architecture.

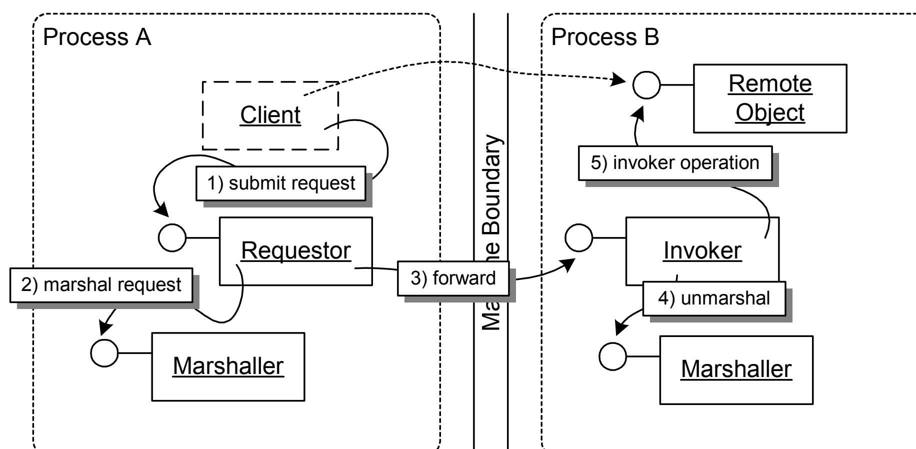

Figure 2. Broker Architecture

In addition to the core patterns consisting of REQUESTOR, INVOKER, and MARSHALLER, the BROKER typically relies on the following patterns:

- A CLIENT PROXY is a placeholder for the remote object in the client process, offering the same interface as the remote object. A CLIENT PROXY lets remote operation invocations look like local operation invocations from a client's perspective. Internally, the CLIENT PROXY transforms invocations of its operations into REQUESTOR invocations. The REQUESTOR is then responsible for constructing the request and for forwarding it to the target remote object.
- An INTERFACE DESCRIPTION is used to make the remote object's interface known to the clients. Thus the INTERFACE DESCRIPTION can be used to construct a CLIENT PROXY for a particular remote object type.
- The CLIENT REQUEST HANDLER and SERVER REQUEST HANDLER handle efficient sending, receiving, and dispatching of requests. These two patterns act at the layer beneath the REQUESTOR and the INVOKER. The request handlers forward and receive request and response messages from and to the REQUESTOR and the INVOKER, respectively.
- A remote invocation introduce new kinds of errors, compared to local invocations, as for instance, technical failures in the network communication infrastructure or problems within the server infrastructure. REMOTING ERRORS are used to signal these new errors types to the client side. The REQUESTOR and INVOKER are responsible for forwarding REMOTING ERRORS to the client, if they cannot handle the REMOTING ERROR on their own.

The patterns mentioned so far detail the BROKER pattern. Figure 3 shows the typical dependencies of the patterns, when they are used within a BROKER architecture.

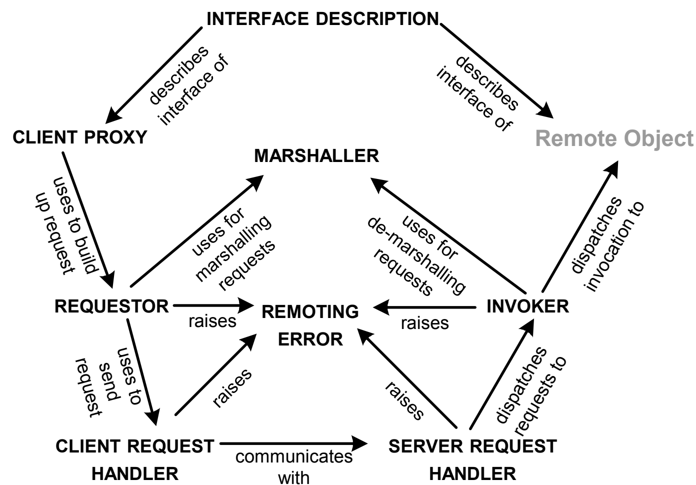

Figure 3. Basic Remoting Patterns - Dependencies

Identification Patterns

Clients of distributed applications need to find the correct remote object within the server application. Thus, means for identification, addressing, and lookup of remote objects are required.

The identification of remote objects in a server application is usually done by assigning logical OBJECT IDS for remote objects. These OBJECT IDS are embedded in remote invocations so that the INVOKER can find the correct remote object. However, OBJECT IDS only identify the remote object in the context of one particular server application. That is, in two different server applications two different objects with the same OBJECT ID might exist. For a remote invocation we additionally need to deliver the message to the correct server application. An ABSOLUTE OBJECT REFERENCE extends the concept of OBJECT IDS with location information, such as the hostname, the port, and the OBJECT ID of a remote object.

Often it is important that the location of remote objects does not need to be hard-wired into the system. For instance, when remote objects are moved to other hosts, the system integrity of the distributed application should not be compromised. The pattern LOOKUP allows server developers to register their remote objects at a central service e.g. by name or property. Clients can then discover the remote objects using this service. 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 dependencies of the identification patterns are visualized in Figure 4.

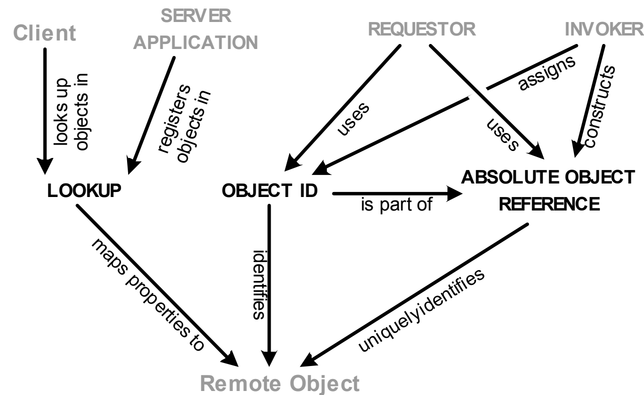

Figure 4. Identification Patterns - Dependencies

Lifecycle Management Patterns

Different remote objects require different lifecycles. Some remote objects need to exist from server application startup to termination. Other remote objects need to be available only for a limited period of time. In addition to difference in the lifecycles, a number of additional tasks might be coupled with the activation and deactivation of remote objects. An important aspect is that the activation and deactivation of remote objects have a strong influence on the overall resource consumption of the distributed application.

The following lifecycle management patterns describe some of the most common lifecycle management strategies used in today's distributed object middleware. The patterns are used to specify the details of activation and deactivation of remote objects.

There are three basic lifecycle management patterns:

- STATIC INSTANCES typically have a lifetime identical to the lifetime of their server application. They are used to represent fixed functionality in the system.
- PER-REQUEST INSTANCES are created for each new request and destroyed after the request. They are used for highly concurrent environments, where resource consumption of each running instance is an issue.
- CLIENT-DEPENDENT INSTANCES rely on the client to explicitly instantiate them. They are used to represent client state in the server.

The lifecycle management patterns internally make use of a set of specific resource management patterns [KJ04], which are:

- LEASING is used to deactivate a remote object after a pre-defined period of time automatically, if the client does not renew the lease before that period of time expires. For example, LEASING is used to properly release CLIENT-DEPENDENT INSTANCES when they are no longer needed.
- LAZY ACQUISITION is used for on-demand-activation of remote objects.
- POOLING manages remote object instances in a pool to optimize reuse of remote object instances. The pattern is especially useful when activation and deactivation incur a significant overhead. This is often the case for short living instances, such as PER-REQUEST INSTANCES.

Another important issue is how to handle situations in which the total number of remote objects exceeds the resources (especially the memory) of the server. One solution to this problem is described by the pattern PASSIVATION: temporarily unused instances are removed from memory

and stored in a persistent storage such as a database. Upon the next request, the instances are restored again.

The lifecycle management patterns and their relationships are shown in Figure 4.

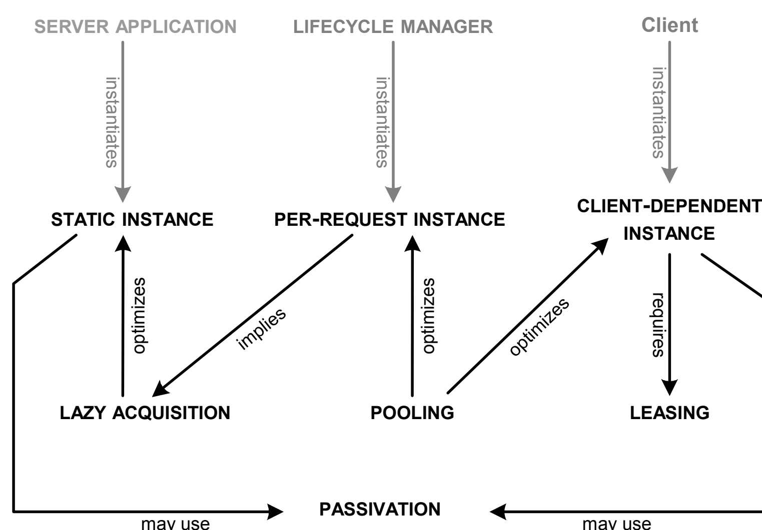

Figure 5. Lifecycle Management Patterns - Dependencies

Extension Patterns

Often, when developing distributed applications, developers need to deal with extension concerns in the context of remote invocations at various layers of the distributed object middleware. Such extension concerns are, for instance, support for security, support for transactions, or the exchange of communication protocols. To handle such extension concerns, remote invocations need to contain more information than just the operation name and its parameters: for instance, for transaction support a transaction ID needs to be transported between client and server. For that purpose INVOCATION CONTEXTS are used: they are added to the remote invocation on client side and read out on server side.

When the invocation process needs to be extended with behavior, INVOCATION INTERCEPTORS can be used. Just consider adding security credentials to a remote invocation. In this example, we require additional behavior for adding the credentials to the invocation on client side and checking them on server side before access to the remote object is granted. INVOCATION INTERCEPTORS can intercept remote invocations after they are invoked by the client on the client side and before they are invoked on the remote object on the server side. They are typically applied in an interceptor chain that is triggered by the REQUESTOR or CLIENT REQUEST HANDLER on the client side, and by SERVER REQUEST HANDLER or INVOKER on the server side. For passing information between clients and servers, INVOCATION INTERCEPTORS use the INVOCATION CONTEXT. INVOCATION INTERCEPTORS are also used to add information transparently to the INVOCATION CONTEXT (such as the security credentials in the example above).

In many distributed applications more than one communication protocol needs to be supported. For instance, an encrypted protocol might be needed in addition to an un-encrypted protocol e.g. to send sensitive data. Simple CLIENT and SERVER REQUEST HANDLER support only a fixed communication protocol. PROTOCOL PLUG-INS extend the CLIENT and SERVER REQUEST HANDLER with support for multiple, exchangeable communication protocols.

The relationships of the extension patterns are illustrated in Figure 6.

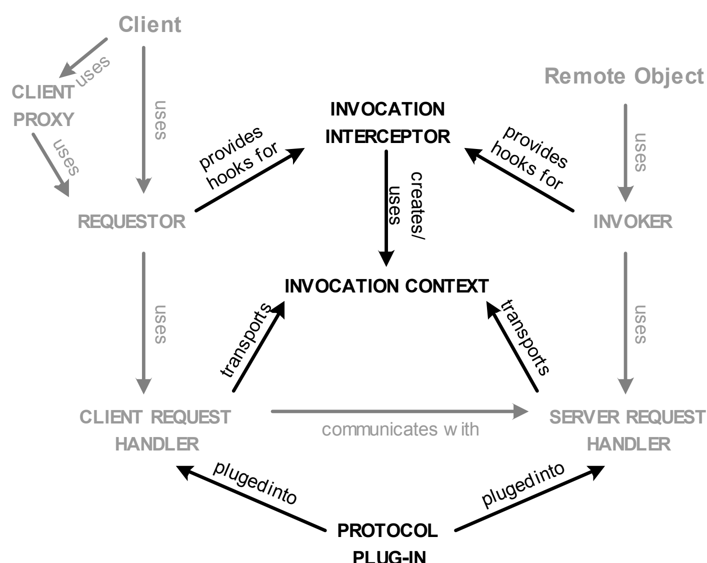

Figure 6. Extension Patterns - Dependencies

Extended Infrastructure Patterns

The following extended infrastructure patterns deal 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 patterns, explained above. It is typically implemented as a part of the INVOKER.

In many situations it is inefficient to configure each remote object separately, e.g. with lifecycle strategies, interceptors, or communication protocols. CONFIGURATION GROUPS are used to configure groups of remote objects with these aspects.

In some distributed applications specific quality of service constraints of the system need to be ensured. To implement such measures, it is necessary to monitor the performance of various parts of the system, such as the INVOKER, the CLIENT and SERVER REQUEST HANDLER, or even the remote objects themselves. This is done using an QOS OBSERVER.

LOCAL OBJECTS are infrastructure objects of the distributed object middleware, such as the REQUESTOR, the LIFECYCLE MANAGER, or the QOS OBSERVERS, that follow the same programming conventions as remote objects, but are inaccessible from remote sites. They ease programming as the same programming conventions can be applied for remote objects and local instances.

LOCATION FORWARDERS can forward invocations between different server applications. They are used to implement load balancing, fault tolerance, and transparency of remote object relocation.

Figure 7 shows the extended infrastructure patterns and their relationships.

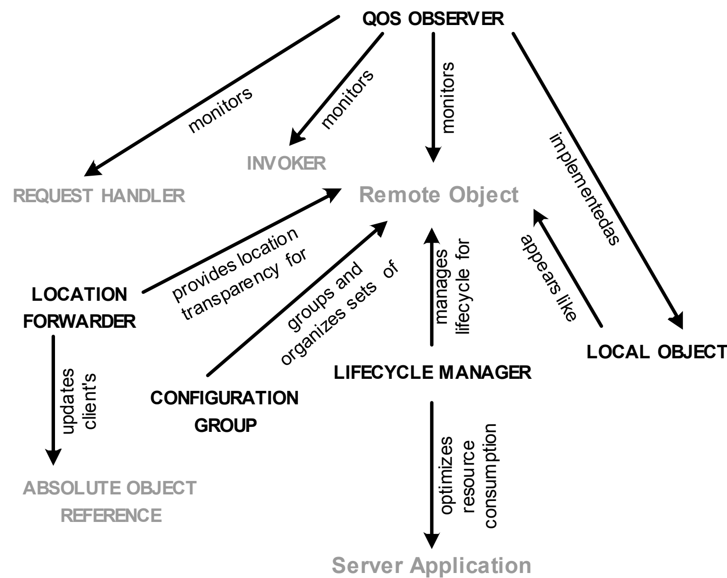

Figure 7. Extended Infrastructure Patterns - Dependencies

Invocation Asynchrony Patterns

Four alternatives for asynchronous invocations can be used to extend ordinary synchronous invocations:

- FIRE AND FORGET describes best-effort delivery semantics for asynchronous operations. It does not convey results nor acknowledgements.
- SYNC WITH SERVER sends an acknowledgement back to the client once the operation has arrived on the server-side, but does not convey results.
- POLL OBJECTS receive a result of an asynchronous invocation. They allow clients to query ("poll") the distributed object middleware for the result of an asynchronous invocation.
- RESULT CALLBACKS also receive a result, but do not wait for the client to poll for the result. Instead they actively notify the requesting client of a asynchronously arriving results.

Figure 8 illustrates the asynchronous invocation patterns and their dependencies.

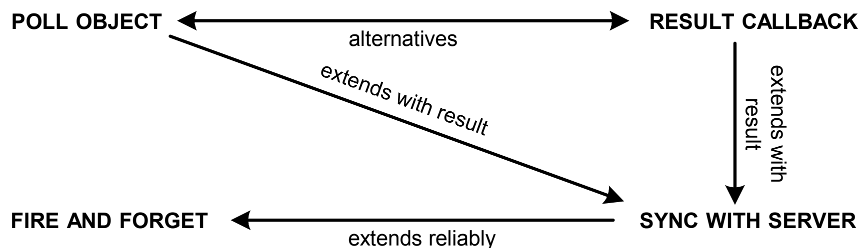

Figure 8. Invocation Asynchrony Patterns - Dependencies

Integration with other Pattern Languages

An important aspect of pattern languages is that they are domain-specific with a language-wide goal. As more and more pattern languages emerge and mature, pattern languages can be used as

a systematic way to integrate solutions from related but independent domains - by describing the links to other pattern languages and patterns, documented elsewhere. In the following we explain the most important links of the Remoting Patterns to closely related pattern languages:

- 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.
- POSA2 [SSRB00] contains many patterns that are used to implement distributed systems - especially at the communication protocol layer of a distributed object middleware. These patterns are mainly used as the constituents of the request handlers and PROTOCOL PLUG-INS. 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 some concurrency patterns with a special focus on Java.
- 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.
- 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 the management of any kind of resource, ranging from typical operating system resources such as threads or connections, up to remote objects or application services.
- 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.
- 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, CCM, or COM+ are typically used), the technical concerns are issues such as transaction management, resource access decision, fail-over, replication, and persistence. For remote access to the *Components* distributed object middleware is typically used.
- When a distributed object middleware is deployed on only one machine, availability and scalability are problematic. When this machine would go down, the whole system would fail. Under increased load conditions, the system might not be able to provide the same or similar levels of performance. To deal with such situations, Dyson and Longshaw introduce patterns for building highly available and scalable distributed systems, especially Internet systems [DL03].
- Fault tolerance techniques are applied to detect errors of a system and recover from errors or mask errors of a system. Saridakis presents basic fault tolerance techniques, including fault detection, recovery, and masking, as a system of patterns [Sar02]. These patterns have two relations to distributed object middleware. First, many fault tolerant systems use replication

on different hardware units, and these distributed units require remote communication. Second, some safety-critical distributed systems require fault tolerance of the implementation of remote objects.

- Security is another important orthogonal concern to be considered when building distributed systems. In [VSK04] a few best practices are discussed when combining the Remoting Patterns with security concerns.
- Aspect-oriented programming (AOP) [KLM+97] avoids tangled solutions for cross-cutting design concerns. AOP is an important future trend in the domain of object-oriented remoting. 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 [Zdu04] a projection of this pattern language to a number of popular aspect composition frameworks can be found. The patterns in this pattern language can also be used to implement aspect solutions for distributed object middleware.

Figure 9 summarizes the relationships of the Remoting Patterns to other patterns and pattern languages by showing the main problem - in form of a question - that leads to considering the other patterns or pattern languages.

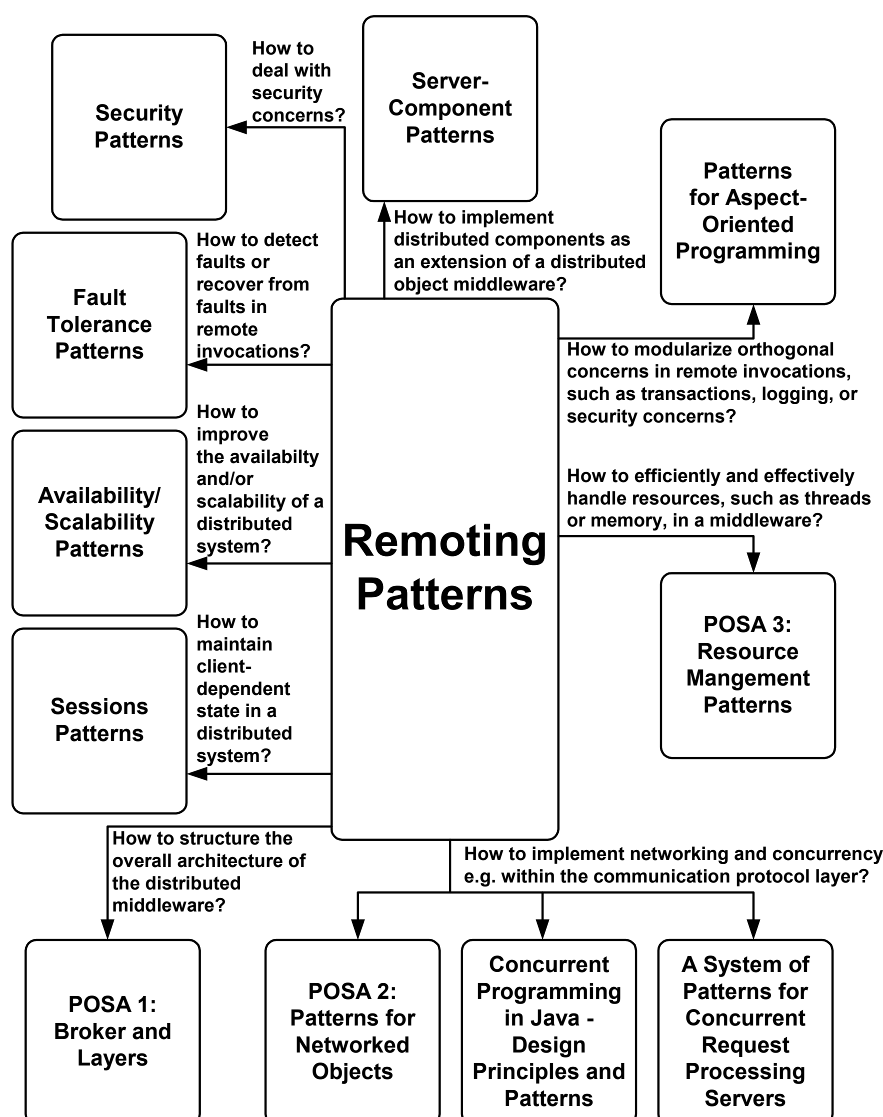

Figure 9. Remoting Patterns and other Patterns/Pattern Languages: Overview

In summary, most of the closely related domains of Remoting Patterns are well captured by other patterns and pattern languages. Thus the Remoting Patterns act as a glue between these patterns and pattern languages, when applied to distributed object middleware or distributed application development.

There are many patterns for extensions of the core concepts of distributed object middleware, 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 not available yet). Also, pattern languages for systems built on top of distributed object middleware are missing in many domains. For instance, there are only a few patterns for P2P systems or GRID computing available yet. The mentioned AOP patterns are only explaining how AOP can be realized, but not how to build distributed AOP applications. This is not astonishing, however: Fields like P2P, GRID, or AOP are still emerging, so it is no wonder that mature patterns are missing for these fields because patterns describe established knowledge. We expect patterns for these fields to appear, when the fields have become more mature.

Conclusion

In this article we have presented an overview of a comprehensive pattern language for distributed object middleware, as well as its connections to other pattern material. The goal of this pattern language is to systematically use, extend, integrate, customize, or even build distributed object middleware - in other words: to document the existing design knowledge in this domain. Also, this pattern language serves as a glue for other pattern material relevant for distributed object middleware systems and distributed applications. This article only gives a brief overview of the Remoting Patterns language. Refer to [VKZ04] for a much more detailed view, including full pattern descriptions and technology projections of the patterns to a number of middleware implementations: .NET Remoting, Web Services, and CORBA/Real-time CORBA.

References

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.

BMR+96 F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. Pattern-Oriented Software Architecture, Volume 1: - A System of Patterns. John Wiley and Sons, 1996.

DL03 P. Dyson and A. Longshaw. Patterns for High-Availability Internet Systems. In Proceedings of EuroPlop 2003, Irsee, Germany, June 2003.

Fow96 Martin Fowler, Analysis Patterns: Reusable Object Models, Addison-Wesley, 1997.

GHJV95 E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Addison-Wesley 1994.

GT03 B. Gröne, P. Tabeling. A System of Patterns for Concurrent Request Processing Servers. Proceedings of VikingPLoP, 2003.

HS James O. Coplien, A Pattern Definition, http://hillside.net/patterns/definition.html

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, Finnland, June 1997. LCNS 1241, Springer-Verlag.

KJ04 M. Kircher, P. Jain, Pattern-Oriented Software Architecture, Volume 3: - Patterns for Resource Mangement, Wiley & Sons 2004

Lea99 D. Lea. Concurrent Programming in Java - Design Principles and Patterns. Addison-Wesley, Reading, Mass., 1996.

PPP The Pedagogical Patterns Project, http://www.pedagogicalpatterns.org

Sar02 T. Saridakis. A System of Patterns for Fault Tolerance. In Proceedings of EuroPlop 2002, Irsee, Germany, July 2002.

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, Volume 2: Patterns for Concurrent and Distributed Objects. John Wiley and Sons, 2000.

SB03 D. C. Schmidt and F. Buschmann. Patterns, Frameworks, and Middleware: Their Synergistic Relationships, Proceedings of the IEEE/ACM International Conference on Software Engineering, Portland, Oregon, May 3-10, 2003.

Ste98 R. Stevens. UNIX Network Programming. Prentice Hall. 1998.

VSW02 M. Voelter, A. Schmid, and E. Wolff. Server Component Patterns. John Wiley and Sons, 2002.

VKZ04 M. Voelter, M. Kircher, and U. Zdun. Remoting Patterns - Foundations of Enterprise, Internet, and Realtime Distributed Object Middleware. John Wiley and Sons, 2004.

Zdu03 U. Zdun. Patterns of tracing software structures and dependencies. In Proceedings of EuroPlop 2003, Irsee, Germany, June 2003.

Zdu04 U. Zdun. Pattern language for the design of aspect languages and aspect composition frameworks. IEE Proceedings Software, 151(2): 67-83, April 2004.