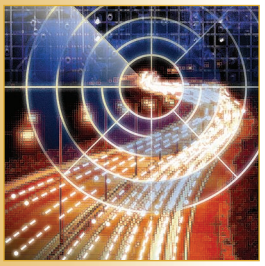

Editors: Doug Lea • *dl@cs.oswego.edu*
Steve Vinoski • *vinoski@ieee.org*

Remoting Patterns

Design Reuse of Distributed Object Middleware Solutions

Patterns and pattern languages offer a practical means for distributed system developers to both gain a deeper understanding of the middleware they use and to convey their knowledge about it. The proposed Remoting pattern language offers a systematic way to reuse software models, designs, and implementations to extend, integrate, customize, or build distributed object middleware solutions. This pattern language has rich dependencies with other patterns and pattern languages from related domains, including networking, concurrency, and resource management.

Uwe Zdun
Vienna University of Economics and Business Administration

Michael Kircher
Siemens AG Corporate Technology

Markus Völter
Ingenieurbüro für Softwaretechnologie

Developers' experiences and actual source code often provide the only concrete design and implementation knowledge for use in maintaining or evolving complex distributed systems. Often, only a few experts thoroughly understand a given system and the prevalent development practices used to build it. Nonexperts must thus invest significant effort to acquire this knowledge, especially after the established experts leave an organization.

Although numerous books and articles describe how to use middleware from a programmer's perspective, or how a specific aspect of a given middleware system was designed, few also explain the rationale behind the design. As outlined by Schmidt and Buschmann, patterns and middleware offer popular techniques for coping with these challenges.[1] Patterns provide reusable design knowledge in the form of proven solutions to recurring software problems in a particular context or domain. Middleware allows developers to reuse a piece of software that hides low-level API details, such as those of operating systems, network protocol stacks, and databases. Today, distributed object middleware — Corba, Web services, Java RMI, .NET Remoting, and so on — is a basic element in the distributed-systems development toolbox.

Schmidt and Buschmann argue that patterns and middleware complement each other to enhance the systematic reuse of successful software models, designs, and implementations.[1] This applies not only for systems built on top of middleware but also for situations that require an understanding of the middleware's inner workings. Although

 Published by the IEEE Computer Society 1089-7801/04/$20.00

middleware systems use different remoting abstractions, terminologies, implementation language concepts, and so forth, they share many concepts. Understanding these common concepts also helps developers of distributed systems switch from one middleware system to another. In some situations, developers need to extend a middleware system with additional functionality. Sometimes it's necessary to integrate different middleware systems to enable independently developed applications to work together. More rarely, developers need to customize distributed object middleware, or even build it from scratch, when no suitable product exists.

In this article, we describe a pattern language that provides a systematic mechanism for providing knowledge about the inner workings of distributed object middleware systems. Existing patterns have explained specific aspects of middleware implementations[2–4] or how to build higher-level systems on top of middleware,[5] but none have addressed the full range of how to use, extend, integrate, and even build distributed object middleware systems. In our book,[6] we present a detailed description of the Remoting pattern language, including full pattern descriptions and projections to middleware implementations such as .NET Remoting, Web services, and Corba.

Communication Middleware

Developers can build distributed systems directly on top of network protocols — communicating over TCP/IP sockets, for example[7] — but doing so forces them to handle all the low-level network programming details. Such systems are usually rather cumbersome and error-prone, and they're not easy to scale, maintain, or change.

Instead, developers typically use middleware as an additional software layer to hide underlying platform heterogeneity and provide transparency in distributed communications — that is, to make remote invocations as similar as possible to local invocations. Of course, full transparency isn't possible because remote invocations always introduce new kinds of errors, latency, and so forth.

Today's middleware systems use several different *remoting styles*, including those based on remote procedure calls (RPCs), messages, shared repositories, and data streams. In this article, we focus on middleware that uses object-oriented variants of the RPC style, but the patterns we present are also relevant for systems that implement other approaches. They are the basis for many other remoting styles, including mobile-code, peer-to-peer (P2P), remote-evaluation, grid computing, publish–subscribe, and transaction-processing systems. Developers often implement such high-level approaches using the basic remoting styles, or variants of them, but shield users from internal details such as RPC mechanisms and naming services used in ad hoc service location.

Patterns and Pattern Languages

Over the past few years, patterns have become a mainstream software-development technique. The most popular patterns are for software design[8,2] and software architecture,[9] although the community has also documented patterns for analysis[10] and several non-IT topics.

In this article, we use Jim Coplien's definition of patterns (www.hillside.net/patterns/definition.html), which summarizes the longer version put forth by Christopher Alexander[11]:

"Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution ... As an element in the world, each pattern is a relationship between a certain context, a certain system of forces that occurs repeatedly in that context, and a certain software configuration that allows these forces to resolve themselves."

Consider the popular `strategy` design pattern as an example.[8] This pattern's *context* includes design situations in which more than one algorithm can (potentially) be applied for the same tasks. The *problem* it addresses is that different algorithms are appropriate at different times, so it should be possible to exchange algorithms — even at runtime — and to add other algorithms in the future. As a *solution* to this problem, developers can apply the `strategy` pattern, which defines a family of algorithms with one common interface, encapsulating each and making them interchangeable.

A pattern describes a solution to a particular, recurring problem. Given that large problems usually can't be described using single patterns, the community has created several ways to combine patterns to solve more complex problems or sets of related problems:

- *Compound patterns* are assembled from smaller patterns.

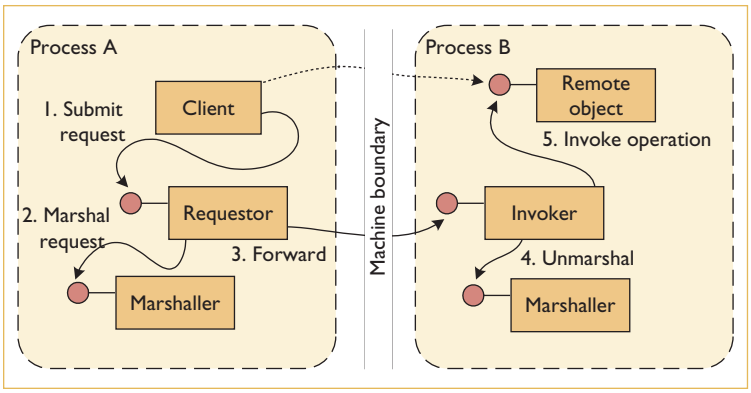

Figure 1. Broker architecture. The client uses a requestor to construct and send remote invocations. On the server side, an invoker receives the invocations and dispatches them to the target remote object.

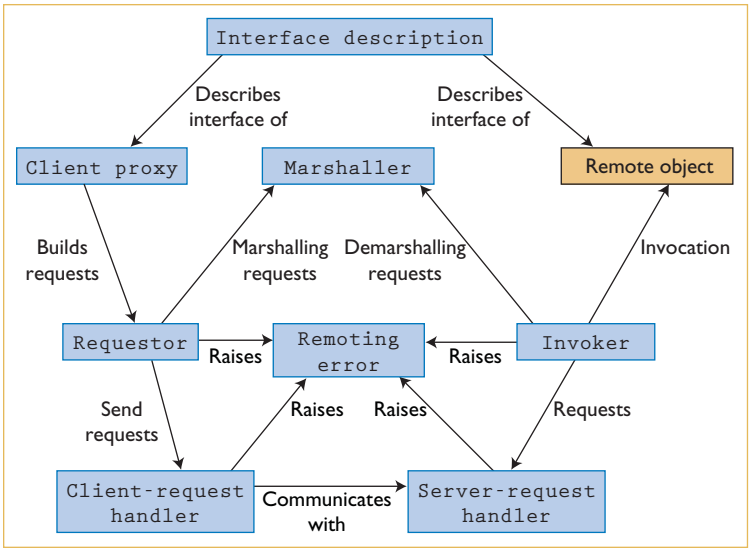

Figure 2. Dependencies in remoting patterns. The basic remoting patterns (blue boxes) provide a layered architecture for the primary tasks in a distributed object middleware: handling requests, dispatching invocations, and defining the application logic. The remote object (gold box) is not a pattern, but a participant the patterns interact with.

- *Pattern families* are collections of patterns that solve the same general problem.
- *Pattern collections*, or systems, include several patterns from a given domain or problem area.
- *Pattern languages* are designed to guide users, in a step-by-step manner, toward a common overall goal. The patterns in a pattern language aren't necessarily useful in isolation, but they work together to holistically solve specific problems.

The patterns we describe in this article form a pattern language in the remoting domain. Together, the patterns explain how distributed object middleware systems work.

Remoting Patterns

The `broker`[9] architectural pattern describes the general architecture of distributed object middleware systems. However, a clear and detailed guide to understanding the components of a `broker` architecture was missing. The Remoting pattern language extends the `broker` architecture to address the full range of how to use, extend, integrate, and even build distributed object middleware systems. We present the pattern language in greater detail elsewhere.[6]

Broker Architecture and Basic Remoting Patterns

Distributed systems software developers face many challenges that don't arise in creating single-process software. Rather than attempting to master all these challenges, including unreliable communication across networks and the need to integrate heterogeneous components, developers of distributed systems need to focus on the domain-specific responsibilities of their distributed applications. The `broker` pattern provides a general architectural guideline that explains how to separate a distributed system's communication functionality from its application functionality by isolating all communication-related concerns. A `broker` hides and mediates all communications between the objects or components in a system. Local client-side and server-side `brokers` enable the exchange of requests and responses between clients and remote objects.

We view the `broker` as a compound pattern that is implemented using several patterns from the Remoting pattern language. Figure 1 shows the typical `broker` architecture in terms of the remoting patterns. A `broker` consists of a client-side `requestor` to construct and forward invocations, and a server-side `invoker` that calls the target remote objects' operations. A `marshaller` on each side of the communications path handles the transformation of requests and responses from programming-language-native data types into byte arrays that can be sent over the wire.

Figure 2 shows the typical pattern dependencies within a `broker` architecture. In addition to the core patterns, the `broker` typically relies on the following:

- A client proxy is a placeholder for the remote object in the client process. By presenting clients an interface that is the same as the remote object's, the proxy lets the client interact with the remote object as if it were a local object. Internally, the client proxy transforms the invocations it receives into requestor invocations, which the requestor pattern then constructs and forwards to the target remote object.
- An interface description is a specification for a remote object interface. Developers can use this description to construct a client proxy for a given remote-object type, for instance, or to generate stub (also called skeleton) for the invoker.
- The client-request handler and server-request handler send, receive, and dispatch requests. Specifically, these two patterns forward and receive request and response messages from the requestor and the invoker, which reside at the layer above the handlers.
- Remoting errors alert clients to the error types introduced by remote (as compared to local) invocations — technical failures in the network communication infrastructure or problems within the server infrastructure, for example. The requestor and invoker forward the remoting errors to the client if they can't handle them on their own.

The patterns described so far are the foundation of any broker architecture. Developers can also use several optional additions to it.

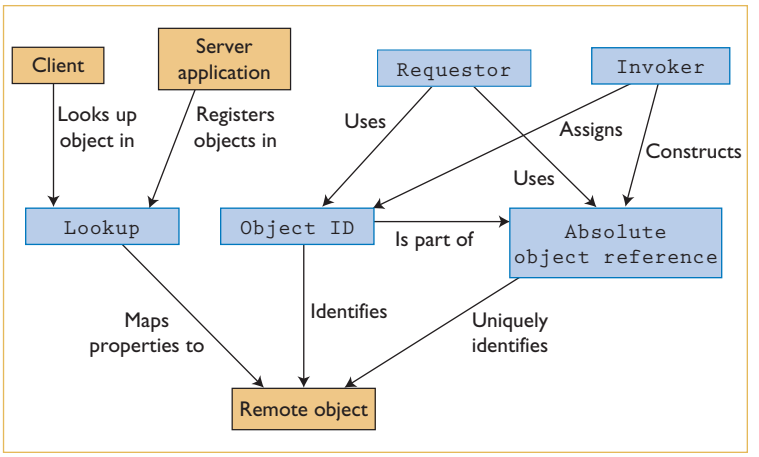

Figure 3. Dependencies in identification patterns. These patterns are used to uniquely identify and locate remote objects.

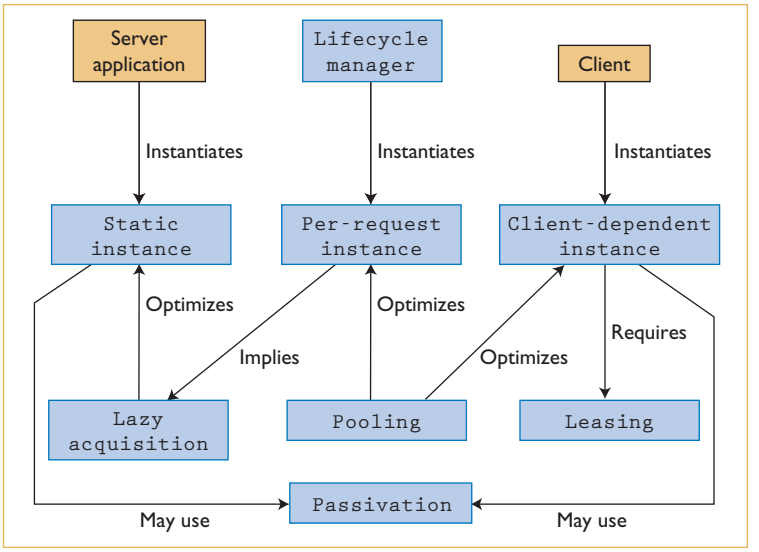

*Figure 4. Dependencies in lifecycle-management patterns. There are three common strategy patterns for lifecycle management (*static instance*,* per-request instance*, and* client-dependent instance*) and four patterns for resource management (*leasing*,* lazy acquisition*,* pooling*, and* passivation*).*

Identification Patterns

To find the correct remote objects within distributed server applications, clients need ways to identify, address, and locate them. Developers usually assign logical object IDs to the objects to identify them; the invoker can then locate a given remote object using the object ID that either the client or client proxy embeds in the remote invocation. Because object IDs are valid only in the context of a specific server application, however, objects in different server applications might have the same object ID. For a remote invocation, we must therefore have some way to deliver the message to the correct server application. An absolute object reference solves this problem by extending object IDs to include location information, such as host name and port, as well as the remote object's ID.

It is often important to avoid hardwiring remote objects' locations into a distributed application or system. Developers and administrators should be able to move objects to other hosts, for example, without compromising a distributed application's integrity. The lookup pattern simplifies the management and configuration of distributed systems by enabling clients to find remote objects, while avoiding tight coupling between them. The lookup pattern lets server

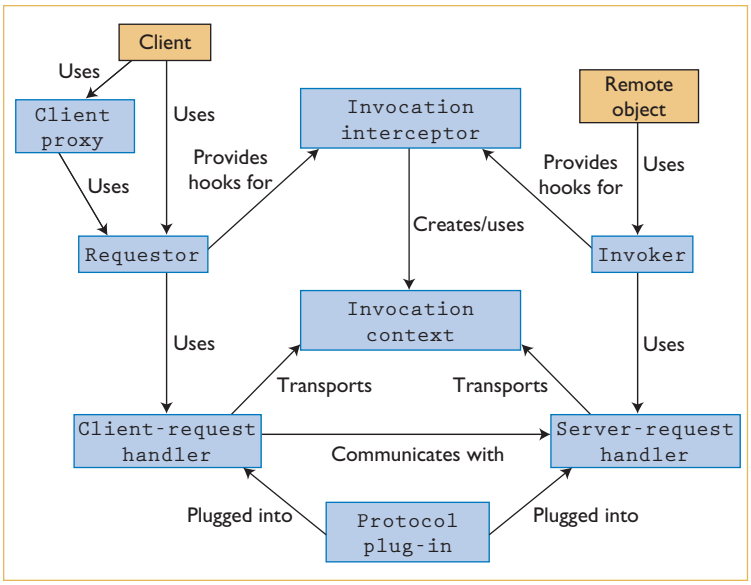

Figure 5. Dependencies among extension patterns. These patterns provide support for extensibility at the invocation and request-handling layer of a distributed object middleware system.

developers register remote objects (by name or property, for example) at a central service, which clients can then use for object discovery. Clients thus need only the lookup service's `absolute object reference`, rather than the potentially vast number of `absolute object references` for objects with which they want to communicate. Figure 3 illustrates the dependencies found in identification patterns.

Lifecycle-Management Patterns

Figure 4 shows lifecycle-management patterns and their relationships. Different remote objects require different life cycles: some need to exist from server-application startup to termination, whereas others need to be available only for a limited time. In addition to differences in life cycles, several other tasks are sometimes coupled with the activation and deactivation of remote objects — activities that strongly influence a distributed application's overall resource consumption.

The most common strategies for managing remote objects' life cycles in today's distributed object middleware follow three basic patterns:

- `Static instances` typically have identical lifetimes to their server applications; they represent fixed functionality in a system.
- `Per-request instances` are created for each new request and destroyed afterward; they're used for highly concurrent environments in which each running instance's resource consumption is an issue.
- `Client-dependent instances` are explicitly instantiated by the client; they represent client state in the server.

Internally, lifecycle-management patterns use a set of specific resource-management patterns[12]:

- `Leasing` lets the patterns automatically deactivate remote objects after a predefined period, unless the client renews the lease before the period expires.
- `Lazy acquisition` lets the server application activate remote objects on demand.
- `Pooling` manages remote object instances in a pool to optimize reuse. This pattern is especially useful when activation and deactivation incur significant overhead, as is often the case for short-lived instances.

To handle situations in which the total number of remote objects exceeds the server's resources (especially the memory), developers sometimes turn to the `passivation` pattern.[5] The pattern describes how to remove temporarily unused instances from memory and store them in a persistent storage, such as a database, until restoring them upon the next request.

Extension Patterns

Developers often need to extend functionality — to support security, transactions, communication-protocol exchanges, and so on — at various layers of the distributed object middleware. In such cases, remote invocations must contain more information than just the operation's name and parameters — transaction support requires a transaction ID, for example. `Invocation contexts` extend remote invocations with an extensible data structure that is sent as part of the remote invocation from the client to server side.

To extend the invocation process with behavior, developers can use `invocation interceptors`. To include security credentials in a remote invocation, for example, requires additional behavior for adding the credentials on the client side and checking them on the server side before granting access to a remote object. Typically applied in a chain triggered by the `requestor` or `client-request handler` or by the `invoker` or `server-request handler`, `invocation interceptors` can intercept

remote invocations to pass information between clients and servers using the `invocation context`; they can also transparently add to the context information.

Many distributed applications require support for more than one communication protocol — for encrypted and unencrypted data, for example. Simple `client-` and `server-request handlers` support only single, fixed communication protocols, but `protocol plug-ins` extend them with support for multiple, exchangeable communication protocols.

Figure 5 illustrates the relationships among extension patterns.

Extended Infrastructure Patterns

Figure 6 shows the relationships among extended infrastructure patterns. These patterns provide support for configuring and accessing a distributed object middleware's infrastructure. The server-side `broker` architecture includes extended infrastructure patterns for several specific implementation aspects:

- The `lifecycle manager`, typically implemented as part of the `invoker`, activates and deactivates remote objects using the lifecycle-management patterns described earlier.
- `Configuration groups` allow developers to configure multiple remote objects that would be inefficient to configure individually — for instance, the lifecycle strategies, interceptors, or communication protocols can be configured for a group of remote objects.
- For distributed applications that require a system to meet specific quality of service (QoS) constraints, a `QoS observer` provides a way to monitor performance for various system components, such as the `invoker`, `client-request handler`, `server-request handler`, or even the remote objects.
- `Local objects` — the distributed middleware's infrastructure objects (`requestor`, `lifecycle manager`, `QoS observers`, and so on) that are inaccessible from remote sites — ease programming efforts by letting developers apply the same programming conventions for both remote objects and local instances.
- `Location forwarders` forward invocations between different server applications to implement load balancing, fault tolerance, and transparency in remote-object relocation.

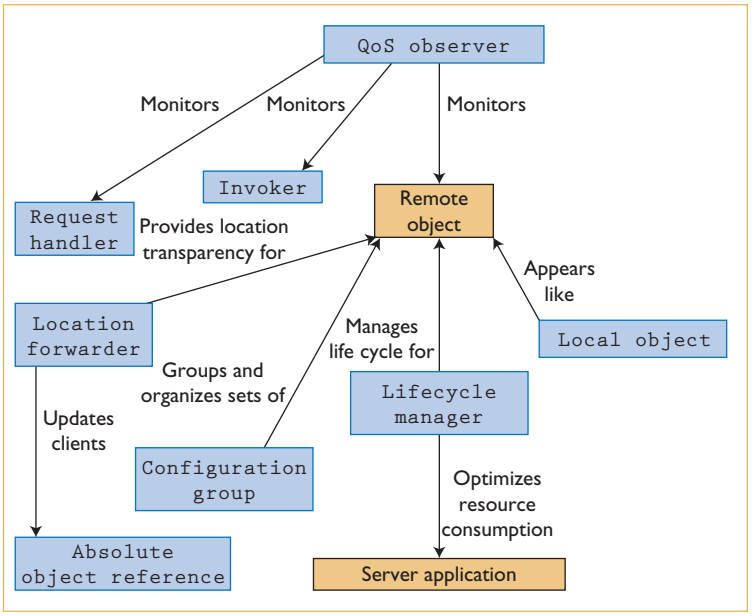

Figure 6. Dependencies among extended infrastructure patterns. These supplement the basic patterns with support for specific implementation aspects.

All extended infrastructure patterns are optional in a `broker` architecture, although larger middleware systems support most of them.

Asynchronous Invocation Patterns

In contrast to synchronous or blocking invocations, asynchronous invocations allow a client to resume its work while a remote invocation is running. The most common variants of client-side asynchrony are described by the asynchronous invocation patterns. Figure 7 illustrates the asynchronous invocation patterns and their dependencies. Developers can use four asynchronous invocation patterns, rather than ordinary synchronous invocations, if invocation asynchrony is required:

- The `fire-and-forget` pattern describes best-effort delivery semantics for asynchronous operations but doesn't convey results or acknowledgments.
- The `sync-with-server` pattern describes invocation semantics for sending an acknowledgment back to the client once the operation arrives on the server side, but the pattern doesn't convey results.
- The `poll-object` pattern describes invocation semantics that allow clients to query ("poll") the distributed object middleware for the results of asynchronous invocations.

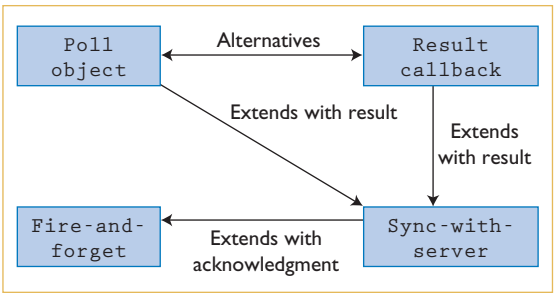

Figure 7. Dependencies among asynchronous invocation patterns. These patterns provide four alternatives for when synchronous invocations are inappropriate because an invocation should not block.

- The `result-callback` pattern also describes invocation semantics that allow the client to receive results; in contrast to `poll object`, however, it actively notifies the requesting client of asynchronously arriving results rather than waiting for the client to poll for them.

All four patterns can be used when synchronous invocations are insufficient because an invocation should not block. Developers can use `fire-and-forget` or `sync-with-server` when no result should be sent back to the client. `Poll object` is appropriate when a result is required and the client has a sequential programming model, whereas `result callbacks` require a client with an event-based programming model.

Conclusion

The Remoting pattern language provides a guide for using and developing distributed object middleware applications. We have provided several projections of how the patterns apply in popular distributed object middleware systems.[6] In the future, we will further work on related patterns such as those in the areas of aspect-oriented and model-driven software development and service-oriented architectures.

Acknowledgments

This article is adapted from the authors' recent book, *Remoting Patterns: Foundations of Enterprise, Internet, and Real-Time Distributed Object Middleware* (Wiley & Sons, 2004).

References

1. D.C. Schmidt and F. Buschmann, "Patterns, Frameworks, and Middleware: Their Synergistic Relationships," *Proc. IEEE/ACM Int'l Conf. Software Engineering*, IEEE CS Press, 2003, pp. 694–704.
2. D.C. Schmidt et al., *Pattern-Oriented Software Architecture: Patterns for Concurrent and Distributed Objects*, vol. 2, Wiley & Sons, 2000.
3. D. Lea, *Concurrent Programming in Java: Design Principles and Patterns*, Addison-Wesley, 1996.
4. B. Gröne and P. Tabeling, "A System of Patterns for Concurrent Request Processing Servers," *Proc. 2nd Nordic Conf. Pattern Languages of Programs* (VikingPLoP), Microsoft Business Solutions, 2003.
5. M. Völter, A. Schmid, and E. Wolff, *Server Component Patterns*, Wiley & Sons, 2002.
6. M. Völter, M. Kircher, and U. Zdun, *Remoting Patterns: Foundations of Enterprise, Internet, and Real-Time Distributed Object Middleware*, Wiley & Sons, 2004.
7. R. Stevens, *Unix Network Programming*, Prentice-Hall. 1998.
8. E. Gamma et al., *Design Patterns*, Addison-Wesley, 1995.
9. F. Buschmann et al., *Pattern-Oriented Software Architecture: A System of Patterns*, vol. 1, Wiley & Sons, 1996.
10. M. Fowler, *Analysis Patterns: Reusable Object Models*, Addison-Wesley, 1996.
11. C. Alexander et al., *A Pattern Language: Towns, Buildings, Construction*, Oxford Univ. Press, 1977.
12. M. Kircher and P. Jain, *Pattern-Oriented Software Architecture: Patterns for Resource Management*, vol. 3, Wiley & Sons, 2004.

Uwe Zdun is an assistant professor in the Department of Information Systems at the Vienna University of Economics and Business Administration. His research interests include software patterns, scripting, object-orientation, software architecture, and Web engineering. Zdun received a PhD in informatics from the University of Essen. He is a member of the IEEE and the ACM. Contact him at zdun@acm.org or http://wi.wu-wien.ac.at/~uzdun/.

Michael Kircher is a senior software engineer at Siemens AG Corporate Technology. He focuses mainly on distributed object computing, software architectures, and design patterns; he has written several patterns, papers, and books on those topics. Kircher received a master's degree (Dipl.-Inform.) in computer science from the University of Stuttgart. Contact him at michael@kircher-schwanninger.de or www.kircher-schwanninger.de/michael.

Markus Völter works as an independent consultant for software technology and engineering. He focuses on software architecture, middleware, and model-driven software development. Völter received a Dipl. Ing. degree from the University of Applied Sciences, Weingarten. He has authored several magazine articles, patterns, and books, and he regularly speaks at conferences. He is a member of the ACM. Contact him at voelter@acm.org or www.voelter.de.

Related Work in Pattern Languages

An important aspect of pattern languages is that they are domain-specific with language-wide goals. For instance, the goal of the Remoting pattern language described in this article is to help developers use and develop distributed object middleware systems. As more pattern languages emerge and mature, developers will be able to use them to systematically integrate solutions from related but independent domains by describing their links to other pattern languages and patterns, documented elsewhere.

Numerous researchers have explored patterns in recent years. Figure A summarizes the relationship between the Remoting pattern language described in this article and other patterns and pattern languages.

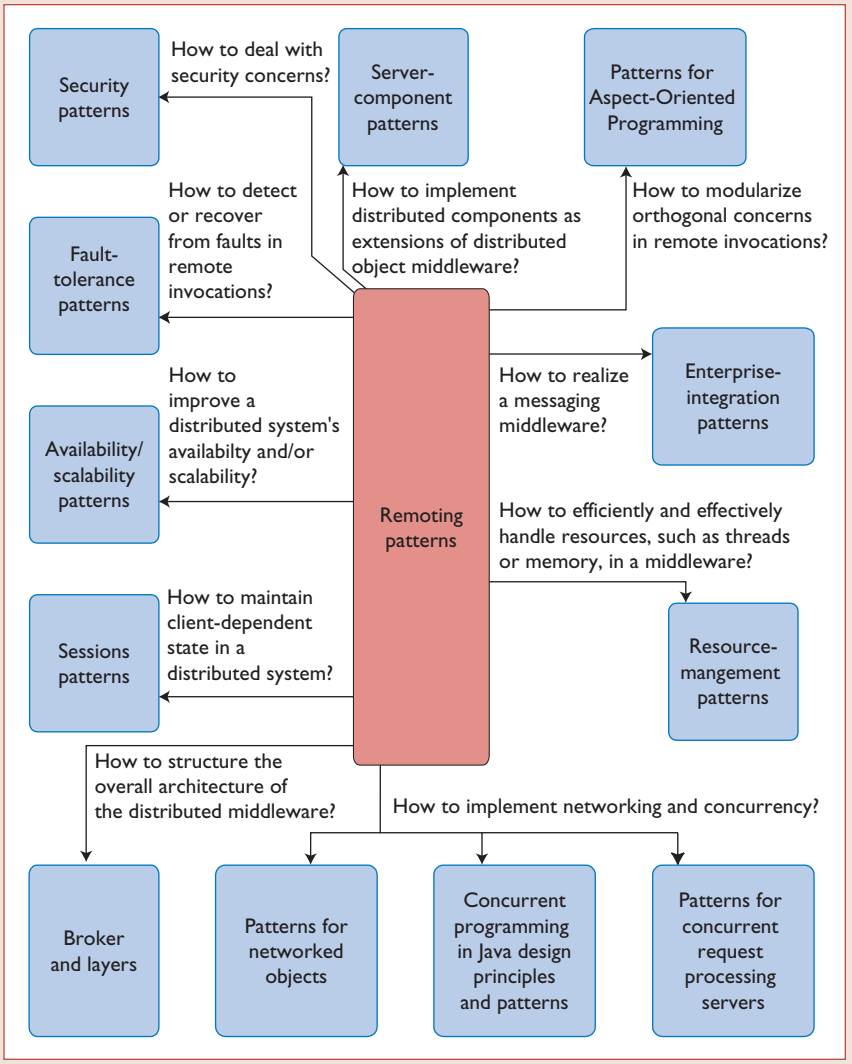

Figure A. Patterns and pattern languages related to the Remoting patterns. The main problems, shown in the form of questions, lead developers to consider other patterns or pattern languages.

Related Patterns

As documented by Buschmann and colleagues,[1] distributed object middleware follows two architectural patterns:

- The `broker` pattern mediates object invocations among communication participants. The `broker` architecture is the foundation of the remoting patterns described in this article.
- The `layers` pattern separates responsibilities by decomposing systems into groups of subtasks in which each group of subtasks operates at a particular level of abstraction. Distributed object middleware systems usually follow a `layers` architecture. For instance, remoting patterns operate at the communication-protocol, request-handling, invocation, and application-logic layers.

Schmidt and colleagues describe many patterns used to implement distributed systems, especially at the communication-protocol layer in distributed object middleware.[2] These patterns are used mainly as the elements of the `client-server-request handlers` and `protocol plug-ins`.

Gröne and Tabeling document several patterns for concurrent request handling in high-performance servers.[3] Lea also describes some concurrency patterns with a special focus on Java.[4]

Kircher and Jain deal with patterns for resource management and optimization with any kind of resource, ranging from typical operating system resources, such as threads or connections, to remote objects or application services.[5] Specifically, they document a pattern language on how to efficiently and effectively acquire, access, and release resources at different layers of abstraction. These patterns are important for managing a distributed object middleware's resources, especially with respect to request handling and remote-object life cycles.

Maintaining client-dependent state presents a common problem in distributed object middleware. The `session` pattern[6] provides a solution by letting the server-side of the distributed object middleware system maintain state between individual client requests, so that new requests can access previously accumulated data. A session identifier lets clients and remote objects refer to specific sessions. While sessions can exist at

continued on p. 68

Related Work in Pattern Languages continued

any protocol level, they are mostly independent of lower-level communication tasks, such as those that arise when multiple client objects share a physical network connection.

Server-side component infrastructures provide a distributed execution environment for software components.[7] The component container provides essential services to components, which can't be executed as stand-alone elements. These services handle applications' cross-cutting technical concerns, which vary according to application domain but typically aren't directly related to the application functionality implemented within the components. In an enterprise environment, these services handle issues such as transaction management, resource-access decisions, fail-over, replication, and persistence. Developers typically use distributed object middleware to facilitate remote access to the components.

Hohpe and Woolf document patterns on how to implement messaging middleware systems.[8] Messaging is inherently asynchronous, and extends distributed object middleware in several ways, including support for reliable message transport, message ordering, message expiration, and different kinds of message channels. Developers can implement messaging systems by extending distributed object middleware or using existing messaging systems inside `protocol plug-ins`.

When a software system is deployed on a single machine, availability and scalability can be problematic. Under increased load conditions, the system might not be able to provide the required performance levels, and the whole system fails if the machine goes down. To deal with such situations, Dyson and Longshaw introduce patterns for building highly available and scalable distributed software systems, especially Internet systems.[9] These patterns can be used to ensure the distributed object middleware system's availability and scalability, for instance, by introducing data replication measures and load balancing.

Saridakis presents basic fault-tolerance techniques, including fault detection, recovery, and masking, as a system of patterns.[10] These patterns have two relations to distributed object middleware. First, many fault-tolerant systems use replication on different hardware units, which require remote communication. Second, some safety-critical distributed systems also require fault tolerance in remote-object implementation.

Aspect-oriented programming is a technique for supporting the separation of concerns. In AOP, developers implement systems' various cross-cutting concerns as separate units (the aspects), which are composed automatically by an aspect-composition framework. AOP is an important future trend in object-oriented remoting because it avoids tangled solutions for cross-cutting design concerns.[11] The term AOP actually denotes several adaptation techniques, which can be implemented using several different aspect-composition frameworks and aspect languages. Zdun describes a pattern language[12] that explains how these aspect composition frameworks are realized internally. In other work, he shows how this pattern language applies in several popular aspect-composition frameworks.[13] These AOP patterns explain only how AOP can be realized, rather than how to build distributed AOP applications, but they also work in implementing aspect solutions for distributed object middleware.

Combining Remoting and Related Patterns

Clearly, other patterns and pattern languages capture most of the closely related domains described by the Remoting pattern language. Our language, in turn, acts as a "glue" between these other languages when applied to distributed object middleware or distributed application development.

There are many patterns for extending the core concepts of distributed object middleware to include functionality such as messaging, fault tolerance, scalability, and session management. We've also described a few best practices for combining the remoting patterns with security[14] — an important orthogonal concern when building distributed systems. Generally, however, domains such as security and transactions in distributed systems aren't well captured by patterns yet; a security patterns book is forthcoming but not available yet (for details, see www.securitypatterns.org). Pattern languages for systems built on top of distributed object middleware are also scant or missing in many domains, including peer-to-peer and grid computing. We expect patterns to emerge as these fields become more mature.

References

1. F. Buschmann et al., *Pattern-Oriented Software Architecture: A System of Patterns*, vol. 1, Wiley & Sons, 1996.
2. D.C. Schmidt et al., *Pattern-Oriented Software Architecture: Patterns for Concurrent and Distributed Objects*, vol. 2, Wiley & Sons, 2000.
3. B. Gröne and P. Tabeling, "A System of Patterns for Concurrent Request Processing Servers," *Proc. 2nd Nordic Conf. Pattern Languages of Programs* (VikingPlop), Microsoft Business Solutions, 2003.
4. D. Lea, *Concurrent Programming in Java: Design Principles and Patterns*, Addison-Wesley, 1996.
5. M. Kircher and P. Jain, *Pattern-Oriented Software Architecture: Patterns for Resource Management*, vol. 3, Wiley & Sons, 2004.
6. K.E. Sorensen, "Session Patterns," *Proc. European Conf. Pattern Languages of Programs* (EuroPlop 02), UKV Konstanz, 2002, pp. 301–322.
7. M. Völter, A. Schmid, and E. Wolff, *Server Component Patterns*, Wiley & Sons, 2002.
8. G. Hohpe and B. Woolf, *Enterprise Integration Patterns*, Addison-Wesley, 2003.
9. P. Dyson and A. Longshaw, *Architecting Enterprise Solutions: Patterns for High-Capability Internet-Based Systems*, Wiley & Sons, 2004.
10. T. Saridakis, "A System of Patterns for Fault Tolerance," *Proc. European Conf. Pattern Languages of Programs* (EuroPlop 02), UKV Konstanz, 2002, pp. 535–582.
11. G. Kiczales et al., "Aspect-Oriented Programming," *Proc. European Conf. Object-Oriented Programming* (ECOOP 97), LCNS 1241, Springer-Verlag, 1997, pp. 220–242.
12. U. Zdun, "Patterns of Tracing Software Structures and Dependencies," *Proc. European Conf. Pattern Languages of Programs* (EuroPlop 03), UKV Konstanz, 2003, pp. 581–616.
13. U. Zdun, "Pattern Language for the Design of Aspect Languages and Aspect Composition Frameworks," *IEEE Proc. Software*, vol. 151, no. 2, 2004, pp. 67–83.
14. M. Völter, M. Kircher, and U. Zdun, *Remoting Patterns: Foundations of Enterprise, Internet, and Real-Time Distributed Object Middleware*, Wiley & Sons, 2004.