Object-Oriented Remoting - Basic Infrastructure Patterns

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

Michael Kircher
Siemems AG
Corporate Technology
Software and System Architectures
Germany
michael.kircher@siemens.com

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

This pattern language describes the building blocks of typical distributed object frameworks, such as Java RMI, CORBA, .NET Remoting, web object systems, or web services. The patterns cover the basic infrastructure of such distributed object frameworks in a rather abstract manner, as it can be observed by developers using a distributed object systems for object-oriented remoting. The patterns presented in this paper are used in almost every distributed object framework application.

Introduction: Remoting Applications

Distributed systems are probably the most common way of building complex software systems today. Many major systems – those that are really large, complex, and expensive – are distributed systems. They are used for many different purposes, including the Internet, reservation systems, in-vehicle software, telecommunication networks, air traffic control, video streaming, and many more.

Many critical issues, such as performance, predictability, parallelism, scalability, partial failure, etc., have to be taken into account (see [TS2002]). Three fundamentally different remoting paradigms are used in today's software systems: there are those systems that use the metaphor of a remote procedure call, those that use the metaphor of posting and receiving messages, and those that use continuous streams of data. Note that this paper looks mostly at the first of these three different paradigms.

In remote procedure call (RPC) systems, two different roles are distinguished: clients and servers. A server provides a (more or less well-defined) set of operations which the client can invoke. These operations look like normal local operations: they typically have a name, parameters and a return type, as well as a way to signal exceptions in some systems. The goal of remote procedure call middleware is to provide clients with the illusion that a remote invocation is the same as a local one. Here, we concentrate on object-oriented remote procedure call systems, where a server application hosts a set of objects that provide the operations to clients as part of their public interface.

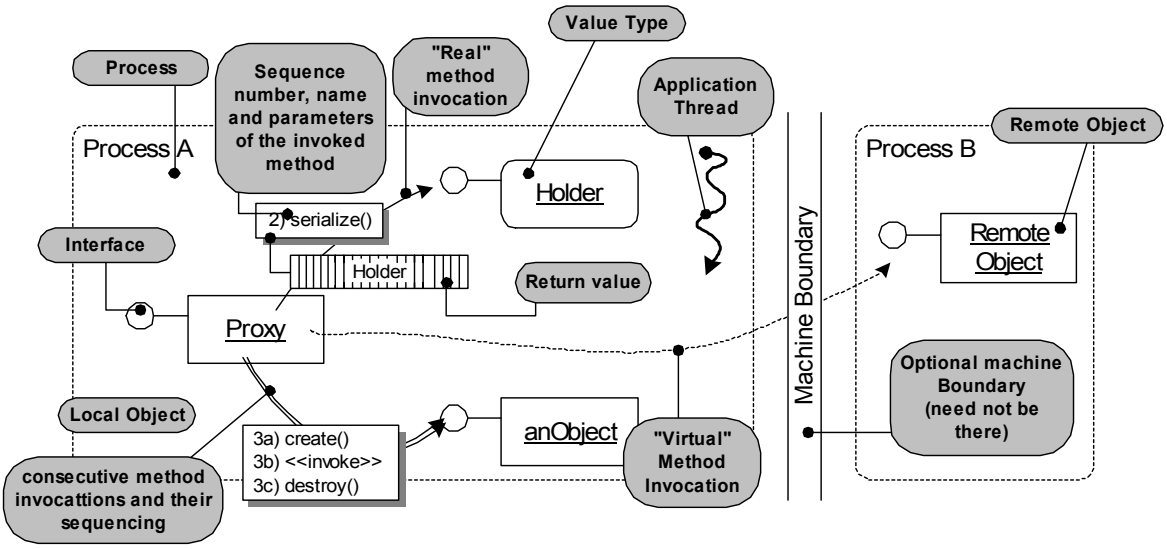
Whichever remoting paradigm is used today in an application, the application developer should be shielded from the details of the underlying metaphor. That is, in (OO-)RPC systems, the developer should only see a local method call, and not need to care about locating the server object, marshalling the request, or detect certain remoting-specific error conditions. A middleware provides a simple, high-level programming model to the developers, hiding all the nitty-gritty details as far as possible (but no further). Middleware specifications and products are available for all remoting paradigms.

In this paper, we will discuss a pattern language consisting of the basic infrastructure patterns that one has to know and understand when working with (or constructing) an (OO-)RPC middleware (called a "distributed object framework").

Pattern Form

The form of our patterns is Alexandrian, without examples. That is, each pattern starts with a name. It is followed by the context of the pattern in the language, and then three stars follow. After that, the problem is described in bold face, and then in plain face, the problem is described in more detail with the forces of the pattern. Then, following the word "Therefore" follows the solution, again in bold face. A detailed solution with emphasis on the related pattern in the language comes next (in plain face). Finally, after another three stars, the consequences of the pattern are given.

Each pattern is illustrated with a "collaboration diagram." As we also display containment structures in these diagrams, the illustrations are not really UML diagrams. The following example illustration is a legend annotated with comments.



Basic Distributed Object Patterns: Overview

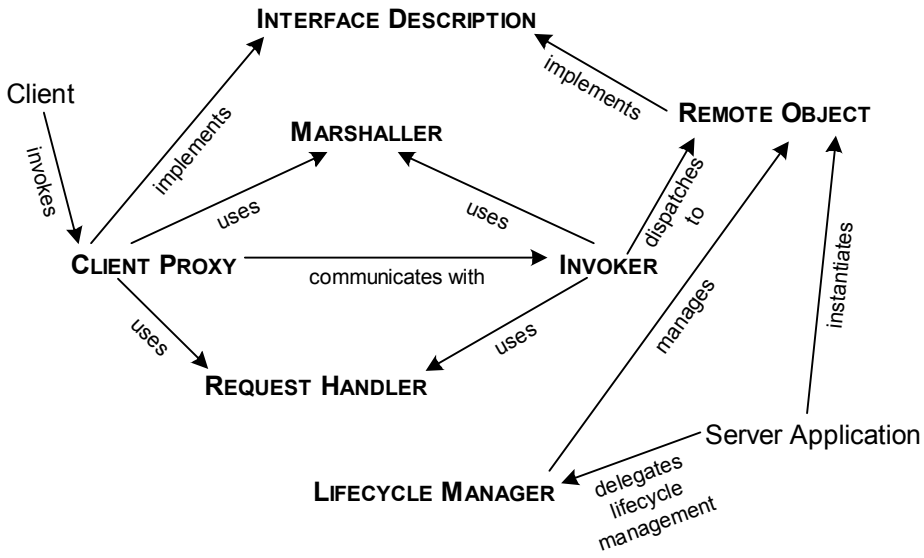
The reason for building or using distributed object frameworks is to allow clients to communicate with objects on a remote server. On the server side the invoked functionality is implemented as a REMOTE OBJECT. The client invokes an operation of a local object and expects it to be executed by the REMOTE OBJECT. To make this happen, the invocation crosses the machine boundary, the correct operation of the correct REMOTE OBJECT is obtained and executed, and the result of this operation invocation is passed back across the network. These basic communication tasks between client and REMOTE OBJECT are handled by the patterns described in this chapter.

A CLIENT PROXY is used by a client to access the REMOTE OBJECT. The CLIENT PROXY is a local object within the client process that offers the REMOTE OBJECT'S interface. This interface is defined using an INTERFACE DESCRIPTION.

The client can use a CLIENT REQUEST HANDLER to handle network communication. On the server side, the remote invocations are received by a SERVER REQUEST HANDLER. It handles the message reception and forwards invocations to the INVOKER, after the message is received completely. The INVOKER dispatches remote invocations to the responsible REMOTE OBJECT using the received invocation information.

On client and server side complex types are serialized and de-serialized using a MARSHALLER.

The LIFECYCLE HANDLER manages lifecycle issues of a group of REMOTE OBJECTS.



In most distributed object frameworks, these patterns are integrated with a few typical components. A SERVER APPLICATION instantiates and controls the REMOTE OBJECTS. The FRAMEWORK FACADE shields the distributed object framework and provides a simple API to developers using the distributed object framework.

The COMMUNICATION FRAMEWORK implements the layer beneath the distributed object framework, and it handles the low-level details of network communication. In object-oriented systems it is usually built using a set of common patterns for concurrent and networked objects. In the distributed object framework, the COMMUNICATION FRAMEWORK is primarily used by the REQUEST HANDLER both on client and server side. A PROTOCOL PLUG-IN can be used by developers to exchange or adapt the protocol implemented by the REQUEST HANDLER.

Remote Object

You are using (or building) a distributed object framework – a framework to access objects remotely. Clients access functionality provided by a remote SERVER APPLICATION.

❋ ❋ ❋

In many respects, accessing an object over a network is different from accessing a local object. Machine boundaries, process boundaries, network latency, network unreliability, and many other distinctive properties of network environments play an important role and need to be "managed." How to access an object in a distant process, separated by a network?

For a remote invocation, machine boundaries and process boundaries have to be crossed. An ordinary, local operation invocation is not sufficient, because additionally the operation invocation has to be transferred from the local process to the remote process, running within the remote machine.

Object identities, unique in the address space of one process, are not necessarily unique across process boundaries and machine boundaries.

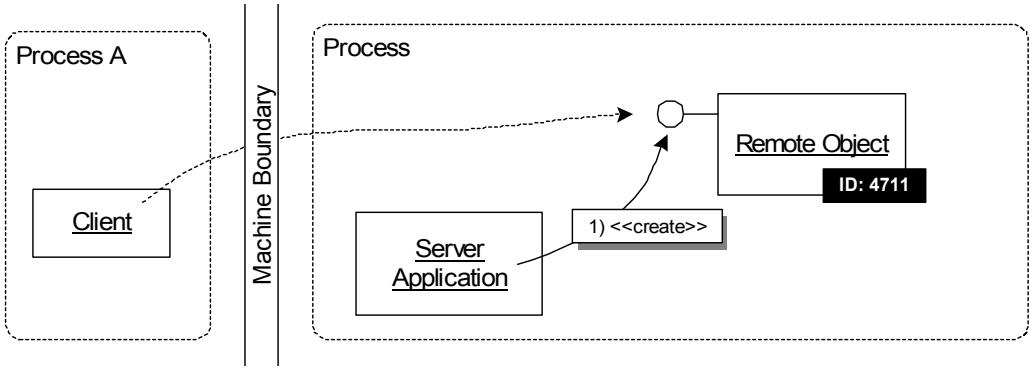
Compared to local invocations, invocations across a network involve delay and unpredictable latency. Because networks must be considered to be unreliable, clients must deal with new kinds of errors. Also, you want to minimize the number of (slow and thus expensive) network hops.

The interface of an object that is provided remotely is different to the local interface. The local interface can contain additional operations that should not be invoked by remote clients, but only by local clients. Thus an interface has to be defined on which remote clients can rely.

The distributed object framework should provide solutions for these fundamental issues of accessing objects remotely.

Therefore:

Provide a distributed object framework on the client side and server side. This framework transfers local invocations from the client side to a REMOTE OBJECT, running within the server. These REMOTE OBJECTS are used as the building blocks for 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 (some of) its operations.



The SERVER APPLICATION instantiates a REMOTE OBJECT. Then clients can access the REMOTE OBJECT using the functionality provided by the distributed object framework.

❋ ❋ ❋

The client and the REMOTE OBJECT usually reside within different processes and possibly also within different machines. The SERVER APPLICATION manages the lifecycle of the REMOTE OBJECTS.

It is responsible for instantiation and destruction. The distributed object framework provides the infrastructure to let clients access REMOTE OBJECTS.

The distributed object framework provides a REMOTE OBJECT type or interface. This is used to distinguish REMOTE OBJECTS from other, local objects. It also provides means to access the functionalities of the distributed object framework from the REMOTE OBJECT.

REMOTE OBJECTS have an unique *object ID* in their local address space, as well as means to construct a *global object reference*. The *global object reference* is used to reference and subsequently access a REMOTE OBJECT across the network.

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 framework. Thus each REMOTE OBJECT type defines or declares its remotely accessible interface. Often this definition or declaration is given as an INTERFACE DESCRIPTION.

The REMOTE OBJECT'S interface is also supported by the CLIENT PROXY that handles remote communication on client side. CLIENT PROXIES communicate with an INVOKER on server side that dispatches invocations to the addressed REMOTE OBJECT.

REMOTE OBJECTS are a solution 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. Different distributed object frameworks hide these issues to a different degree. When designing distributed object frameworks, there is always a trade-off between possible control and ease-of-use.

Client Proxy

You want to access a REMOTE OBJECT, running in a server, from a remote client.

❋ ❋ ❋

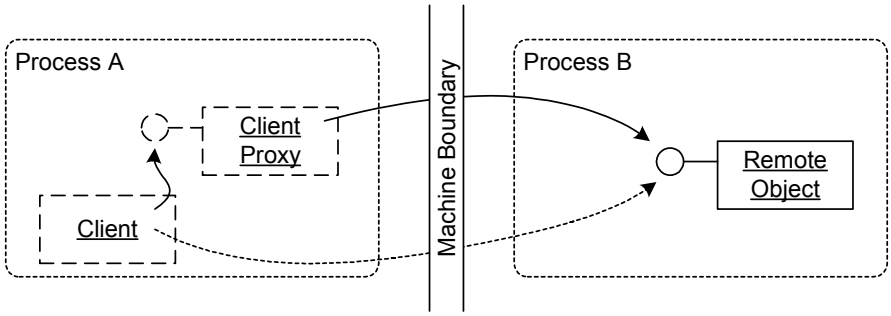
A primary goal of using REMOTE OBJECTS is to support a programming model for REMOTE OBJECTS that is similar to accessing local objects. A client developer should not have to deal with issues like access to the network, transmission of invocations, marshalling, and similar basic remoting functionalities. Of course, some fundamental properties of remote programming, such as unreliability and slowness of network calls, cannot be completely hidden. These should be presented to clients with the proper abstractions from the exploited object-oriented programming model.

The main purpose of distributed object frameworks is to ease development of distributed applications. Thus, developers should not necessarily leave their accustomed "way of programming." In the ideal case, they can simply invoke operations of the REMOTE OBJECTS just as if they were local objects.

Of course, some issues of remote activation and remote error conditions have to be considered by the client developer. Fundamental network properties such as network unreliability and latency also make an invocation of remote operations different to a local invocation. In cases where the client developer needs control over some remoting properties, appropriate APIs should be provided and these should integrate well with the accustomed programming model. In general, however, a remote invocation should be very similar to a local invocation. Issues of transporting an invocation across the network, such as mapping invocations to the REMOTE OBJECT'S address or marshalling the invocation data, should be hidden from client developers.

Therefore:

In the client application, use a CLIENT PROXY object for accessing the REMOTE OBJECT. The CLIENT PROXY object supports the interface of the respective REMOTE OBJECT. For remote invocations, clients only interact with the local CLIENT PROXY object. The CLIENT PROXY primarily forwards invocations to the REMOTE OBJECT. It is responsible for the details of accessing the REMOTE OBJECT via the distributed object framework. Only those remoting details that cannot be handled automatically are exposed to the client developer.



To invoke an operation of a REMOTE OBJECT, the client invokes a operation of the CLIENT PROXY that supports the REMOTE OBJECT'S interface. The CLIENT PROXY forwards this invocation to the REMOTE OBJECT. It handles the details of crossing the machine boundary and process boundary.

❋ ❋ ❋

A CLIENT PROXY is a variant of the *proxy* pattern [GHJV95] which is also documented in a variant supporting remoting [BMR+96]. CLIENT PROXIES do not access the REMOTE OBJECT directly. Instead they call an INVOKER on the server that dispatches the request to the correct target object. Thus

the INVOKER is responsible for transforming the remote invocation into a local invocation inside the REMOTE OBJECT'S server process.

Developers of clients cannot assume that the REMOTE OBJECT is reachable all the time, for instance, because of network delays, network failures, or server crashes. The CLIENT PROXY abstracts these remoting details for clients using *remoting errors*.

Instantiation of REMOTE OBJECTS requires the involvement of the SERVER APPLICATION. Sometimes client and CLIENT PROXY can also be involved, if a client-dependent instance is required. A related issue that involves the CLIENT PROXY is distributed garbage collection. In the context of client side failures, it has to be ensured that client-dependent instances are cleaned up, when they are not required anymore. *Leases* provide a possible solution to this problem.

The CLIENT PROXY is required on client side, and thus it has to be deployed to the client somehow. The simplest solution are CLIENT PROXIES that are hard-wired in the client code. The liability of this simple solution is that a complete, new client has to be deployed every time some interface or remote access operation of one REMOTE OBJECT changes. In some systems a client does not even know before runtime to which REMOTE OBJECT it connects, and thus the CLIENT PROXY cannot be hard-wired in the client. How to allow clients to use server objects without knowing about them at compile time? To resolve these problems, two CLIENT PROXY variants provide a solution:

- *Exchangeable client proxy:* The CLIENT PROXY class can be designed to be exchangeable in the client. Then a CLIENT PROXY class that corresponds to a specific INVOKER is delivered to the client at runtime. This usually happens during startup of the client. Distributing the CLIENT PROXY can be automatically handled by *naming*. This solution has the advantage that CLIENT PROXIES can be exchanged transparently, but incurs the liability that it requires the CLIENT PROXY classes to be sent (in binary form) across the network.
- *Dynamic invocation interface:* Alternatively, a more generic CLIENT PROXY can be used. That means, the CLIENT PROXY does not provide the REMOTE OBJECT'S interface as its own interface, but invocations are dynamically constructed. That means, the interface of the REMOTE OBJECT is not known in advance (before the invocation reaches the CLIENT PROXY), and thus has to be looked up for each invocation. Dynamic invocations on the CLIENT PROXY are very flexible but they incur some performance overhead for lookup of the REMOTE OBJECTS interface.

For dynamic invocations, there are again two variants; either the CLIENT PROXY or the INVOKER can lookup the interface of the REMOTE OBJECT:

- *Interface repository:* If the CLIENT PROXY looks up the interface dynamically, it has to query an interface repository (a variant of the pattern INTERFACE DESCRIPTION) to retrieve the operation signature.
- *Dynamic dispatch on the invoker:* Alternatively, the CLIENT PROXY can send a symbolic invocation (like strings containing *object ID*, operation name, and arguments) across the network. Then it is the burden of the INVOKER to dynamically look up the interface. Note that this CLIENT PROXY variant has to deal with a special *remoting error* type that is raised when the dynamic dispatch was not successful.

The CLIENT PROXY uses a MARSHALLER to marshall the invocation and to de-marshall the result received.

CLIENT PROXY code for accessing the COMMUNICATION FRAMEWORK and invoking the MARSHALLER can be reused to a large degree. Only interface-dependent parts are custom (for instance per REMOTE OBJECT type). These interface-dependent parts can be generated or retrieved from the REMOTE OBJECT'S INTERFACE DESCRIPTION automatically.

Invoker

A remote invocation reaches the server side and should be delivered to a specific REMOTE OBJECT.

❋ ❋ ❋

When a CLIENT PROXY forwards 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. For large numbers of REMOTE OBJECTS this solution does not work. For one, there may be not enough network addresses (that is: network ports) for each REMOTE OBJECT. The client developer would have to deal with the network addresses to select for the appropriate REMOTE OBJECT, which is cumbersome and too complex. There is no way for the SERVER APPLICATION to control the access to its REMOTE OBJECTS (centrally). How to avoid these server side problems of invoking REMOTE OBJECTS?

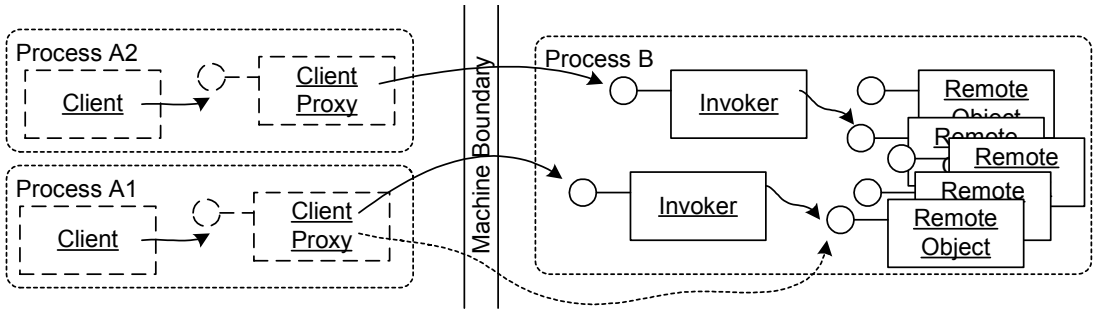
Consider a SERVER APPLICATION providing a large number of sensors as REMOTE OBJECTS to remote clients. If each of the sensor instances would be addressed over the network directly, a first problem might be that the possible number of sensors might exceed the number of free network ports. Then the sensors cannot be provided anymore by directly binding the REMOTE OBJECTS to network ports. Thus, if REMOTE OBJECTS are directly addressed over the network, scalability might be limited, due to the number of free ports.

But even if there is only a limited number of clients, there are still other issues left open, when directly associating REMOTE OBJECTS with network ports. Client developers would have to be aware of (the large number of) sensor network addresses. This is a considerable complexity that should be avoided. Instead, a client should only have to provide the necessary information required to select the appropriate sensor's REMOTE OBJECT, and the SERVER APPLICATION should have to deal with finding and invoking that object.

In many application scenarios a SERVER APPLICATION requires some central control over the invocation process of its REMOTE OBJECTS. Consider you want to implement some functionality that operates for a set of REMOTE OBJECTS of a SERVER APPLICATION. Examples are logging remote invocations or access control of REMOTE OBJECTS. In both cases it is required to deal with the invocation before or after it reaches the REMOTE OBJECT. The developer of the REMOTE OBJECT should not have to implement code for these issues; instead, these issues should be handled by the SERVER APPLICATION solely.

Therefore:

Provide an INVOKER that is remotely accessed by CLIENT PROXIES. The CLIENT PROXIES send invocations across the network, containing the actual operation invocation information and additional contextual information. The INVOKER has to de-marshal the operation invocation information to obtain the ID of the targeted REMOTE OBJECT and an identifier of operation to be invoked. Then the INVOKER dispatches the invocation to the REMOTE OBJECT; that is, it looks up the correct local object and operation implementation, corresponding to the remote invocation, and invokes it. The INVOKER can also be used to extend or control the invocation process.



The CLIENT PROXY sends a request to the INVOKER with invocation information for a REMOTE OBJECT. The INVOKER dispatches the local object address and operation implementation. Then it invokes the correct operation implementation for this object.

❄ ❄ ❄

The INVOKER is part of the SERVER APPLICATION. Possibly one SERVER APPLICATION can provide more than one INVOKER. The task of receiving messages via the network is handled by a REQUEST HANDLER. The REQUEST HANDLER is responsible for the details of receiving remote messages, such as threading, connection pooling, and accessing the operating system APIs. It hands over control to the INVOKER after a message is received completely.

INVOKERS specifically handle the message dispatching task for a group of REMOTE OBJECTS. Such a group of REMOTE OBJECTS might, for instance, consist of all REMOTE OBJECTS in a SERVER APPLICATION, or of the object in a specific REMOTE OBJECT configuration groups.

To reduce the amount of memory resources needed, the server can temporarily evict REMOTE OBJECTS instance from memory or use different *pooling* strategies. The INVOKER is used to implement this functionality.

Message dispatching means to de-marshal the symbolic information in the message, then to use this information to determine the target object and operation, and finally invoke this operation. De-marshalling is handled by a MARSHALLER.

For determining and invoking the target object and operation, different context information can be used, such as the *object ID*, an operation identifier, and the REMOTE OBJECT type. Context information of a remote invocation is implemented using *invocation contexts*. The *invocation context* at least contains the *object ID* and operation to be invoked.

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

- *Server stubs* (also called skeletons) are a static dispatch mechanism. The part of the INVOKER that is responsible for actually invoking the object is generated from the INTERFACE DESCRIPTION. For each REMOTE OBJECT type one server stub is generated. Thus the INVOKER „knows“ its REMOTE OBJECT types, operations, and operation signatures in advance. The INVOKER can directly invoke the called operation, and it does not have to find the operation implementation dynamically. Of course, the corresponding CLIENT PROXY has to address the correct INVOKER and determine the operation implementation statically. In comparison to dynamic dispatch variants, static dispatch eliminates the performance overhead of looking up operation implementations at runtime.
- *Dynamic dispatch:* Consider a situation in which the REMOTE OBJECTS interfaces are not known at compile time. Then it is not possible to generate or write (static) server stubs for the INVOKER. Instead the INVOKER has to decide at runtime which operation of which remote object should be invoked. To do this, first the INVOKER has to extract the *object ID*, operation name, and arguments from the invocation information sent by the CLIENT PROXY. Next it has to find the corresponding REMOTE OBJECT and operation implementation in the local process,

for instance, using runtime dispatch mechanisms, such as reflection [Mae87] or dynamic lookup in a table. Finally, the target instance and operation are invoked. This form of dispatch is called dynamic dispatch, as the INVOKER dispatches each invocation at runtime (for more details of this implementation variant see the pattern *message redirector* [GNZ01]). Dynamic dispatch is more flexible than static dispatch but has a performance penalty. As an invocation can possibly contain a non-existing operation or use a wrong signature, a special *remoting error* has to be delivered to the CLIENT PROXY in case the invocation fails.

Note that for using an INVOKER it is necessary to create a corresponding CLIENT PROXY. The CLIENT PROXY has to provide the invocation information required by the INVOKER. However, it is not necessarily required that the dynamic dispatch variant of INVOKER is combined with a dynamic invocation interface of CLIENT PROXY, or that (static) server stub requires static CLIENT PROXIES. In existing systems, often matching variants (like server stubs together with static CLIENT PROXIES) are used merely out of practical reasons. For instance, a code generator can directly generate matching server stubs and static CLIENT PROXIES from an INTERFACE DESCRIPTION. Of course, dynamic and static dispatch can also be mixed, and many existing distributed object systems provide static and dynamic INVOKERS simultaneously.

There is a trade-off between dynamic and static dispatch variants: dynamic variants are more flexible because they can dynamically be modified or extended with new object and operation types. Static variants are more efficient but they usually require recompiling INVOKERS and CLIENT PROXIES. That means new CLIENT PROXIES have to be distributed to the client side. Deploying CLIENT PROXIES to clients can be a problem; in such cases, a (more) dynamic solution can help to implement modifications only on server side.

An INVOKER bundles the REMOTE OBJECTS for which it dispatches messages, for instance, all objects of a specific REMOTE OBJECT type. Using INVOKERS instead of direct network connections to the REMOTE OBJECTS, reduces the number of required network addressable entities, and the REQUEST HANDLER can use this information to share connections from the same client. The INVOKER can be used to implement behavior affecting a group of REMOTE OBJECTS.

Request Handler

You are providing REMOTE OBJECTS in a SERVER APPLICATION, and INVOKERS are used for message dispatching. A client invokes REMOTE OBJECTS using a CLIENT PROXY.

❋ ❋ ❋

For sending invocations from the client to the server side, many tasks have to be performed on client side: connection establishment and configuration, result handling, timeout handling, and error detection. On server side similar tasks have to performed before the responsible INVOKER can dispatch the respective operation to the targeted REMOTE OBJECT: the server has to listen to the respective port, handle (socket) communication via the communication protocol, and manage the communication resources. Especially for larger and more performance-critical systems it is necessary to allow for global optimization and coordination of the communication resources. That means to coordinate communication setup, resource management, threading, and concurrency in a central fashion. If more than one INVOKER or CLIENT PROXY is used, the request or reply has to be forwarded to the responsible INVOKER or CLIENT PROXY.

Consider a typical scenario in the embedded systems domain: a SERVER APPLICATION manages a large number of different types of controller objects. Each controller receives measurements from sensor objects in clients and forwards actions to actuator objects. Consider further that TCP/IP is used for network communication. In a naive implementation, each controller object type would have its own INVOKER, which would also manage connections with the clients. If a client connects to several different controller objects on the same server, an unnecessarily high number of TCP/IP connections needs to be maintained. If the SERVER APPLICATION would use a (blocking) thread to handle requests for each connection, a high number of threads would be necessary, impeding server performance.

In this scenario, similar problems can occur on client side, when the CLIENT PROXY that handles every detail of the network communication on its own. This would only work for simple, single-threaded clients with only a few requests. In more complex clients, possibly a large number of requests are sent simultaneously. Each CLIENT PROXY may have to handle multiple invocations at the same time.

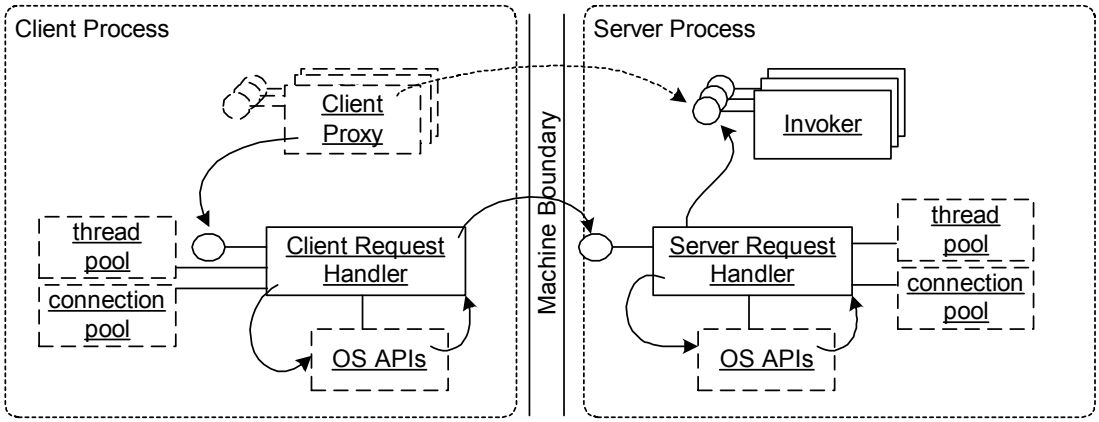
In general, a way to coordinate communication handling and management is required to arrange the network traffic and resource consumption effectively. In most applications it cannot be predicted at which time a particular REMOTE OBJECT or INVOKER (or CLIENT PROXY respectively) has which workload. In such cases, the fair allocation of shared communication resources, as well as quality of service (QoS) constraints, should be handled centrally. Thus the allocation of the communication resources should not be realized by individual CLIENT PROXIES, REMOTE OBJECTS, or INVOKERS. This is also important for reducing the number of used network connections: potentially, network connections to the same client can be shared for the whole server process.

Configuration of network protocols, add-on services, and general asynchrony has to be done on a per invocation basis. For instance, only some operations may be logged, need access control, or need a special Quality of Service. Often client and server side have to work in sync to handle these tasks.

Therefore:

Provide a SERVER REQUEST HANDLER together with a respective CLIENT REQUEST HANDLER. The CLIENT REQUEST HANDLER is used for client side invocations by the CLIENT PROXIES. It is responsible for opening and closing the network connection. It also sends the invocation across the network, waits for results, and dispatches them to the CLIENT PROXY. Additionally, it needs to cope with timeouts and errors on a per invocation basis. The SERVER REQUEST HANDLER is responsible for dealing with the network communication on server side. Messages

are received via network connections (incrementally). The SERVER REQUEST HANDLER waits until the full message has arrived, and then, if required, the message is (partially) demarshalled to find the responsible INVOKER. Next, the message is forwarded to this INVOKER for further processing. Finally, the REQUEST HANDLER has to clean up the connection and other resources it has used. The REQUEST HANDLER typically uses efficient, optimized mechanisms of the underlying operating system. Both CLIENT and SERVER REQUEST HANDLER can use *pooling* for connection handles and request threads.



Connections are handled by the REQUEST HANDLER on client and server side. These are used by CLIENT PROXIES and INVOKERS. The REQUEST HANDLER makes effective use of the communication resources and the operating system APIs, and it pools connections handles and threads.

❋ ❋ ❋

The primary responsiblity of a REQUEST HANDLER is to handle network communication. For this purpose, it instantiates a connection handle per connection. The connection handle is responsible for opening and closing the socket connection and storing the file descriptor of the conncection.

The CLIENT REQUEST HANDLER has to manage network connection on client side. It sends invocation requests and informs the client when the result arrives. For a synchronuous invocations this simply means to return to the waiting CLIENT PROXY operation. For asynchronuous invocations the CLIENT REQUEST HANDLER can invoke a *result callback*, a *poll object*, or use other asynchrony strategies.

When timeouts have to be supported, the CLIENT REQUEST HANDLER informs the CLIENT PROXY of timeouts, as it already does this for general network errors. For this purpose, the CLIENT REQUEST HANDLER sets up a timeout event when the invocation is sent to the REMOTE OBJECT. If the reply does not arrive within the timeout period the CLIENT PROXY is informed.

A SERVER REQUEST HANDLER generically handles the communication across the network. INVOKERS, in contrast, are specific for a group of REMOTE OBJECTS. Therefore, SERVER REQUEST HANDLER and INVOKERS are implemented as separated components because the SERVER REQUEST HANDLER should handle the communication resources independently of particular INVOKERS or REMOTE OBJECTS. Implementing both INVOKER and SERVER REQUEST HANDLER as one component makes only sense for small SERVER APPLICATIONS or if there is just one INVOKER per SERVER APPLICATION.

Both CLIENT and SERVER REQUEST HANDLER have to deal with network events: the CLIENT REQUEST HANDLER has to wait for the result of its invocations, the SERVER REQUEST HANDLER has to wait for arriving invocations. Both REQUEST HANDLERS typically use the *reactor* [SSRB00] pattern for demultiplexing and dispatching events from the network. The REQUEST HANDLER receives the events dispatched by the *reactor* and handles them. The same event dispatching infrastructure

usually can be used on client and server side (for more details see the description of the COMMUNICATION FRAMEWORK at the end of this chapter), whereas CLIENT and SERVER REQUEST HANDLER are different implementations.

The REQUEST HANDLER can also use *half-sync/half-async* [SSRB00] and/or *leader/followers* [SSRB00] to manage network connections and threading efficiently. The patterns CLIENT PROXY, REQUEST HANDLER, and INVOKER together build a *broker* as described in [BMR+96]. REQUEST HANDLERS are also responsible for making effective use of the operating system APIs.

For each particular connection, the REQUEST HANDLER has to instantiate a connection handle. To optimize resource allocation for connections, the connections can be shared in a pool as well (using the pattern *pooling* [KJ02]). If one particular client process sends or receives more than one message to the SERVER APPLICATION at the same time, the network connection can be shared among these messages.

If a client communicates with a REMOTE OBJECT in a SERVER APPLICATION, in some application scenarios it can be expected that this client will potentially communicate again with the same SERVER APPLICATION. Thus the connection can be held open for a certain period of time and used in case of a continued communication. This form of continued communication is called a persistent connection [FGM+99] and, if required, it is implemented by the REQUEST HANDLER. Persistent connections eliminate the overhead of establishing and destroying connections for continuous client requests.

The REQUEST HANDLER provides framework functionality for PROTOCOL PLUG-INS. Also other add-on services, such as access control or logging, can be implemented by REQUEST HANDLER. Note that many of these tasks require the CLIENT and SERVER REQUEST HANDLER to work in concert.

Although the REQUEST HANDLER is described as a single component that all invocations have to pass, usually it is not a bottleneck because with connection pools and thread pools it is highly concurrent. The pattern allows to effectively use the communication resources, independently of the REMOTE OBJECT, CLIENT PROXY, or INVOKER structures. Because of this, a REQUEST HANDLER is reusable for many different clients and SERVER APPLICATIONS. The same REQUEST HANDLER instance is shared by multiple CLIENT PROXIES on client side and multiple INVOKERS on server side. REQUEST HANDLERS hide connection and message handling complexity from the CLIENT PROXIES and INVOKERS.

However, REQUEST HANDLERS impede a slight overhead. In smaller applications with only few network connections and high performance requirements, the performance overhead may matter. Another liability may be the memory overhead of the connection and thread pools, especially in limited computing environments such as embedded systems. Using connection and thread pools in REQUEST HANDLERS does only make sense if the instantiation times for connection handles and threads may have a significant performance impact in terms of response times. This means that there should be a considerable number of messages expected to be received at the same time. For example, simplistic clients can also handle network communication on their own (for instance, simply with a blocking request).

Marshaller

You make REMOTE OBJECTS available to clients. The invocation data is transported over the network. CLIENT PROXY, REQUEST HANDLER, and INVOKER handle the basic communication issues.

❄ ❄ ❄

The data to describe operation invocations of REMOTE OBJECTS consist of the target object's *object ID*, the operation identifier, the arguments, and possibly other context information. All this information has to be transported over the network connection. For transporting it over the network only byte streams are suitable as a transport format.

For sending invocation data across the network there has to be some concept for transforming invocations of remote operations into a byte stream format. There are some exceptional cases that make this task complicated, as for instance:

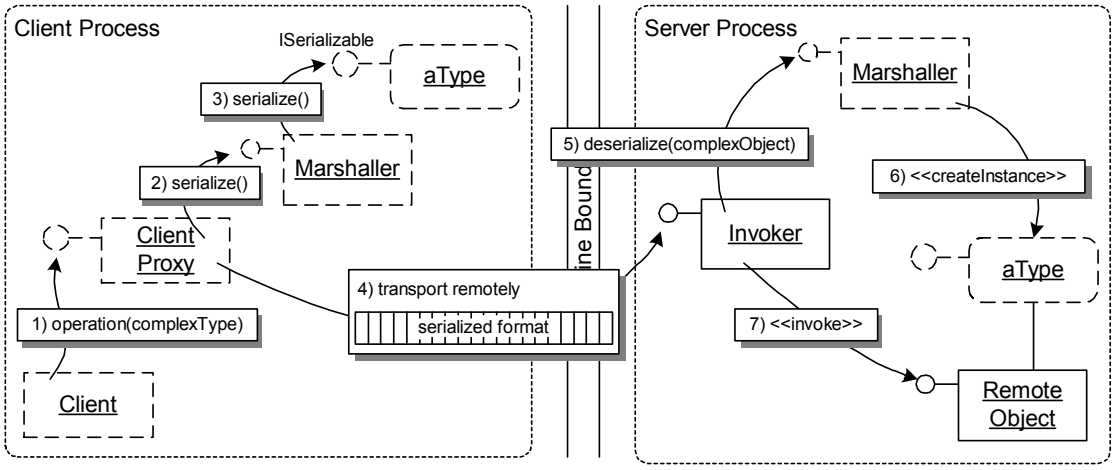
- How do you handle references occurring multiple times?
- How do you handle object identity for non-remote objects?
- How do you determine wether an attribute of an operation should be transformed into a byte stream or not (for instance, references to GUI objects or local resources will usually not be transmitted)?
- How do you handle domain-specific or application-specific specialties of the REMOTE OBJECTS (for example, for objects that are persistent in a database the relationships with the database have to be handled)?
- How do you handle complex, user-defined type objects? These 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.

Generating and interpreting a byte stream representation should not require additional programming efforts per instance, but should only be defined once per type.

Sometimes even the process of generating and interpreting transport formats should be extensible for developers. Additionally, sometimes the used data formats have to be extensible as well.

Therefore:

Require each type used within REMOTE OBJECT invocations to provide a way to serialize itself into a transport format that can be transported over a network as a byte stream. The distributed object framework provides a MARSHALLER (on client and server side) that uses this mechanism whenever a remote invocation needs to be transported across the network. A MARSHALLER also implements some scheme how to preserve object identities and deal with complex data types. In many systems, developers can provide custom MARSHALLERS to customize this scheme or to use other transport formats. The MARSHALLER also provides operations to de-marshal a given byte stream. Make sure the CLIENT PROXY, INVOKER, and REQUEST HANDLER invoke the respective MARSHALLER at the appropriate times.



An operation with an object as argument is invoked. The CLIENT PROXY uses the responsible MARSHALLER to serialize this object. The serialized format is transported across the network. This format is de-serialized by the MARSHALLER on server side and the instance of the respective type is created. Finally, the REMOTE OBJECT is invoked and uses this object.

❋ ❋ ❋

A MARSHALLER converts remote invocations into byte streams in a way that is non-specific for REMOTE OBJECT types.

Complex type object should not be referenced remotely, but marshalled by value. To transport such a 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 can provides a generic, built-in facility. This is often the case in interpreted languages which can use reflection to introspect the type's structure.
- Tools generate serialization code directly from the INTERFACE DESCRIPTION, assuming that the structure of such types is also expressed in the INTERFACE DESCRIPTION.
- It also can be the developer's burden to provide the 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 such serialized data depends on the distributed object framework used. In principle, everything is a byte stream as soon as it is transported over the network. It is often more convenient to use a structured format such as XML, CDR, or ASN.1 to represent complex, structured data.

A MARSHALLER can support a hook to let developers provide a custom MARSHALLER. Reasons are that generic marshalling may become a rather complex and performance-consuming operation, when you need to marshal complex data structures, such as graphs. Some serialization formats might be better than others for certain environments. Thus one generic marshalling format can never be optimal for each data structure. You might also need to optimize data packets for bandwidth.

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. As a consequence, exchanging a MARSHALLER is not necessarily transparent for the application on top.

Different serialization formats have their benefits and liabilities. For instance, the byte stream format can be application-specifically optimized for efficient processing or memory or bandwidth usage. But such formats are hardly human-readable, and harder to create and parse, because standard tools are usually missing. Standard representations such as XML or CDR can be created and parsed easily with standard tools, but may result in a less efficient (because more or less generic) representation. This means these formats require more processing power to be created or parsed. Some formats, especially XML, are very bloated, because they use a human-readable, text-based representation. While there is a large set of tools to process XML, it requires a lot of memory and significantly more network bandwidth than more condensed formats, such as binary data. Standard representations are usually more interoperable than application-specific formats.

Interface Description

A client invokes an operation of a REMOTE OBJECT using CLIENT PROXY and INVOKER.

❋ ❋ ❋

When a CLIENT PROXY and an INVOKER are used together for remote communication, you need to align the interface of CLIENT PROXY and REMOTE OBJECT. Also, you need to align marshalling and de-marshalling. Client developers need to know the interfaces of REMOTE OBJECTS they use.

If a CLIENT PROXY should be used as a local representative of a REMOTE OBJECT in the client process, it exposes the interface provided by the REMOTE OBJECT. If the CLIENT PROXY is responsible for ensuring that the interface is used correctly, the type, operation, and signature information of the REMOTE OBJECT have to be known before a invocation is sent across the network.

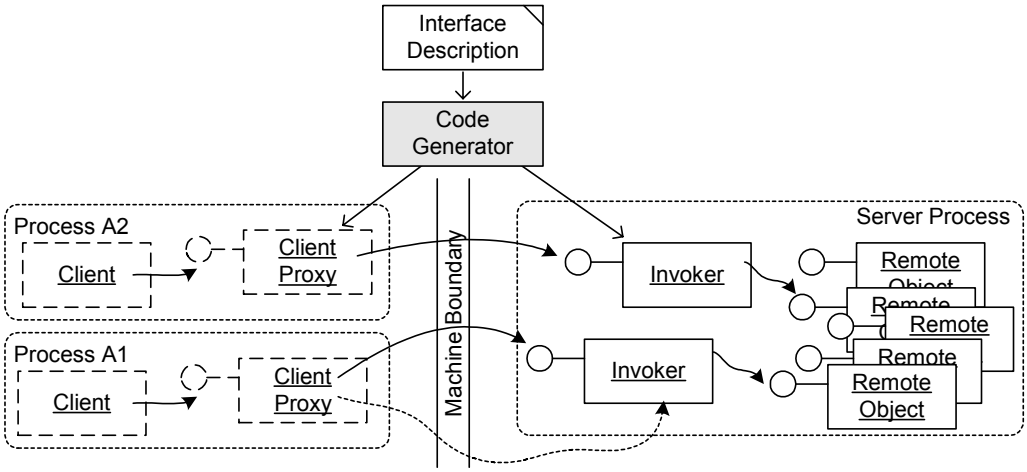
On server side, the INVOKER has to dispatch the invoked operation of the REMOTE OBJECT. If the INVOKER does not dispatch all type, operation, and signature information dynamically, it requires these information before an invocation takes place.

Either CLIENT PROXY or INVOKER should ensure that no violation of the REMOTE OBJECT'S interfaces occurs, or, if violations are possible, CLIENT PROXY or INVOKER have to handle them.

It should not be required that client developers or REMOTE OBJECT developers have to deal with propagating and/or ensuring REMOTE OBJECT interfaces manually. Instead, the distributed object framework should provide suitable means for partly automating these issues.

Therefore:

Provide an INTERFACE DESCRIPTION in which you describe the interface of a REMOTE OBJECT type required for CLIENT PROXY and INVOKER, as well as information for marshalling and dispatching.From the INTERFACE DESCRIPTION interface-related parts of CLIENT PROXY and INVOKER can be derived (either with code generation or runtime configuration techniques). The INTERFACE DESCRIPTION also documents the remoting interfaces for client developers.



Using an INTERFACE DESCRIPTION in a separate file a code generator generates code for CLIENT PROXY and INVOKER. The CLIENT PROXY is then used by the client. It contacts the INVOKER that dispatches the invocation to the REMOTE OBJECT. This way, details of the distributed object framework are hidden from client and REMOTE OBJECT developers, as they only see client code, INTERFACE DESCRIPTION and REMOTE OBJECT code.

❋ ❋ ❋

Usually an INTERFACE DESCRIPTION contains interface specifications including their operations and signatures, as well as marshalling and dispatching information. The INTERFACE DESCRIPTION itself can be given in various forms:

- *Interface description language:* The INTERFACE DESCRIPTION is separated from the program text, for instance, in an additional file written by the REMOTE OBJECT developer. An interface description language is used to provide these information. A code generator generates the interface-related parts of CLIENT PROXY and INVOKER. That means (many) interface violations can be automatically detected when compiling client and CLIENT PROXY.
- *Interface repository:* The INTERFACE DESCRIPTION is provided at runtime to remote clients using an interface repository exposed by the SERVER APPLICATION (or by some external interface repository). Note that an interface repository is required for implementing the dynamic invocation interface variant of CLIENT PROXY. An interface repository is not sufficient for code generation purposes solely, as code generators require the information before runtime.
- *Reflective interfaces:* The INTERFACE DESCRIPTION is only provided on server side by means of reflection [Mae87]. The dynamic dispatch variant of INVOKER is used to obtain the interface information and handle the complete invocation process. A simplistic CLIENT PROXY only sends symbolic invocations to the INVOKER, but does not know the actual interface of the REMOTE OBJECT. Note that this variant requires the INVOKER to handle interface violations and other exceptions. This variant is not suited for code generation, as the INTERFACE DESCRIPTION is not exposed before runtime.

Note that the different variants of INTERFACE DESCRIPTION correspond to the used variants of CLIENT PROXY and INVOKER. Static variants of CLIENT PROXY and INVOKER can exploit code generation techniques and thus primarily use separated interface descriptions in interface description languages. 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 frameworks more than one INTERFACE DESCRIPTION variant is supported, as more than one variant of CLIENT PROXY and/or INVOKER are supported.

operation signatures offered to clients should generally be designed to stay stable. However, changes cannot be avoided. Especially, in a distributed setting, where SERVER APPLICATION developers have no control over client code (and deployment of it), there should be some common way to provide modified remoting interfaces to client developers. INTERFACE DESCRIPTIONS primarily separate interfaces from implementations. Thus the software engineering principle “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 exchanged.

Lifecycle Manager

Your SERVER APPLICATION provides different kinds of REMOTE OBJECTS. Each of these objects has a lifecycle.

❄ ❄ ❄

A SERVER APPLICATIONS has to handle the lifecycle of its REMOTE OBJECTS. In first place, that means to create each object when it is needed and ensure that objects are destroyed (using their destructor) when the server (or thread) terminates. At runtime, the SERVER APPLICATION has to control its resources. For instance, it should ensure that only those objects that are actually needed are active (in memory). All others should be either destroyed or passivated to a database (according to the REMOTE OBJECT'S activation strategy).

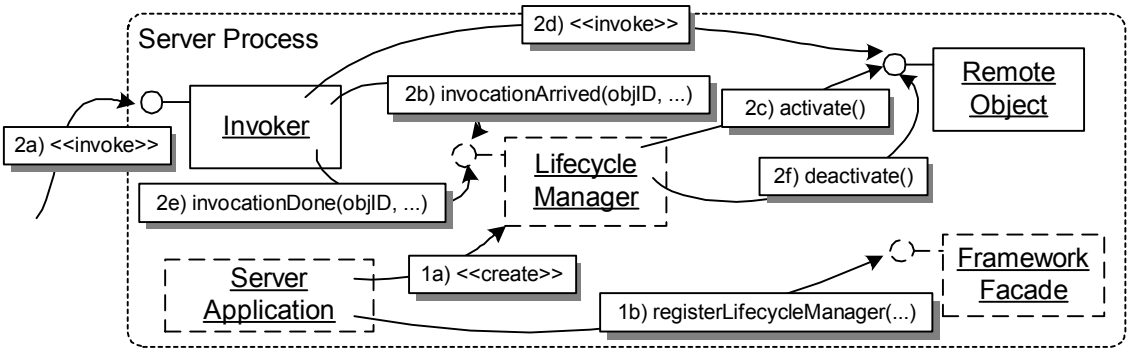
Consider a web community portal with about one hundred thousand registered users. Only a small number of these users will be active at the same time. Thus for reasons of scalability, only those user objects that are actually needed at a certain point in time should be active objects in memory. All other objects should be created or activated on demand, according to the activation strategy of their REMOTE OBJECT type.

Implementing standard activation, sharing, and eviction strategies requires triggering certain lifecycle events. A generic mechanism used for implementing these lifecycle events would also leverage the integration of these patterns and code reuse in their implementations.

Besides standard activation, sharing, and eviction strategies, sometimes developers require custom lifecycle *strategies*. Consider an application in which a REMOTE OBJECT can be paused; that is, the messages for this object are sent to a message queue and processed later on. Some instance has to redirect the message for certain *object IDs* to the message queue instead of the paused REMOTE OBJECT.

Therefore:

Provide a LIFECYCLE MANAGER to handle the lifecycle of a set of REMOTE OBJECTS. It also stores the current lifecycle state of each REMOTE OBJECT. The REMOTE OBJECTS implement lifecycle operations corresponding to the possible lifecycle events. These allow the LIFECYCLE MANAGER to modify the lifecycle state of an object. Different LIFECYCLE MANAGERS can be present in the same SERVER APPLICATION, implementing different lifecycle *strategies*. A custom LIFECYCLE MANAGER can extend both lifecycle states and lifecycle operations. Before and after an invocation the INVOKER calls the LIFECYCLE MANAGER to ensure that the invoked object is active.



The responsible LIFECYCLE MANAGER is created by the SERVER APPLICATION during startup and it is registered with the distributed object framework using the FRAMEWORK FACADE'S API. Before an invocation the LIFECYCLE MANAGER is informed by the INVOKER. If the REMOTE OBJECT is not active, the LIFECYCLE MANAGER activates it. Then

the invocation is performed. After it returns, the LIFECYCLE MANAGER is informed again and can deactivate the object.

❋ ❋ ❋

The INVOKER invokes the LIFECYCLE MANAGER before and after each invocation so that the LIFECYCLE MANAGER can handle the lifecycle events. Informing the LIFECYCLE MANAGER of events in the INVOKER can be hard-coded in the INVOKER code, or it can be implemented with an *invocation interceptor*. Often it is also necessary to let REMOTE OBJECTS invoke their associated LIFECYCLE MANAGER to inform it about certain lifecycle events. Thus it is necessary that each REMOTE OBJECT can obtain a reference to its LIFECYCLE MANAGER.

Note that it might be necessary to let the LIFECYCLE MANAGER work asynchronously, for example, to scan for objects that should be deactivated because some timeout has been reached. The LIFECYCLE MANAGER can, for instance, be informed of the timeout with a callback operation.

The possible lifecycle events of a LIFECYCLE MANAGER are implemented as lifecycle operations by the REMOTE OBJECTS. The REMOTE OBJECTS have to provide lifecycle operations that fit to the LIFECYCLE MANAGER'S lifecycle *strategy*. The LIFECYCLE MANAGER can invoke these lifecycle operations to change the lifecycle state of an object. Typical lifecycle states are: not existing, inactive, virtual, and active.

Consider implementing *passivation*. A persistent object may have an additional lifecycles state *virtual,* meaning that the REMOTE OBJECT is currently not an active instance, but its *object ID* still can be passed to clients. Upon an invocation, some entity has to map the requested *object ID* to the serving, virtual REMOTE OBJECT, and it has to be re-activated on demand.

The LIFECYCLE MANAGER needs to know the REMOTE OBJECTS under its control. For this purpose, it can maintain an object map which associates *object IDs* with REMOTE OBJECTS, or it uses a list of objects maintained elsewhere in the SERVER APPLICATION. The current lifecycle state is also stored in the LIFECYCLE MANAGER'S object map.

The LIFECYCLE MANAGER can be used to implement activation, sharing, and eviction *strategies*. Especially, the activation patterns can be implemented this way. Providing new, custom activation policies makes SERVER APPLICATIONS customizable in terms of performance tuning and custom resource management.

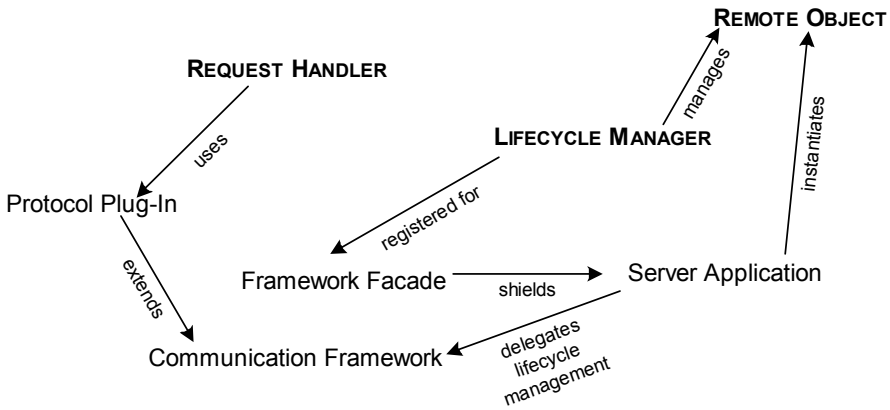
However, a LIFECYCLE MANAGER also incurs the liability of a slight performance overhead, as it has to be informed of every invocation and return of a message.

A LIFECYCLE MANAGER can be seen as a combination of *activator* [SSRB00] and *evictor* [Jai01].

Integrating the Patterns

There are many interactions among the patterns presented and with the patterns presented in later chapters. Components of the environment of a distributed object framework, as well as the use of the patterns in these components, are discussed in the remainder of this chapter. In particular:

- A SERVER APPLICATION is a central instance that manages its REMOTE OBJECTS. Its most important task is to bundle all the objects that belong to one application or service. During startup the SERVER APPLICATION initializes the distributed object framework, if necessary, and obtains references to well-known REMOTE OBJECTS, such as *naming*. Also, it instantiate REMOTE OBJECTS according to their activation strategy and delegates lifecycle management of REMOTE OBJECTS to its LIFECYCLE MANAGER.
- A FRAMEWORK FACADE shields the distributed object framework from direct access. The distributed object framework is a complex piece of software and only parts of it are relevant for developers of SERVER APPLICATIONS. Necessary configuration parameters are offered with a concise interface, the FRAMEWORK FACADE.
- The COMMUNICATION FRAMEWORK implements the low-level details of network connections. The distributed object framework patterns largely abstract these details. However, sometimes it is important to understand the COMMUNICATION FRAMEWORK elements. We explain them with a set of common pattern for concurrent and networked objects.
- A PROTOCOL PLUG-IN is plugged into the CLIENT and SERVER REQUEST HANDLER. It substitutes the communication protocol, used per default by the REQUEST HANDLER, if necessary. This can be used for optimizing the network protocols used for particular applications.



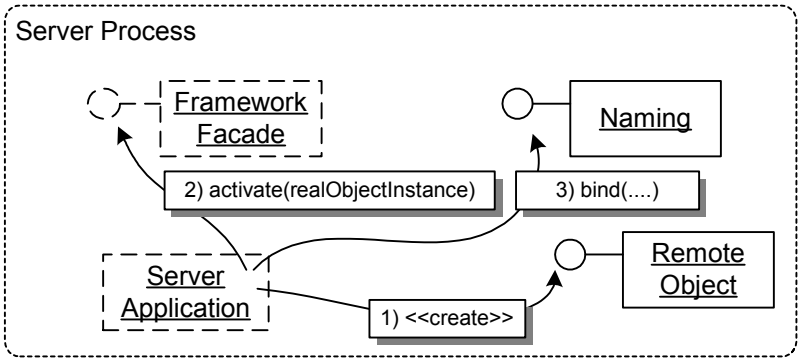
Server Application

The REMOTE OBJECT pattern describes how services are offered remotely. In order to achieve this, there are many tasks that have to be fulfilled in the server:

- The distributed object framework has to be initialized.
- Like any other object, a REMOTE OBJECT has to be instantiated. In a distributed object framework there are different strategies defined when and how remotely accessible objects are instantiated.
- When *naming* is used, a REMOTE OBJECT can be registered with it. To make this possible, a *global object reference* to *naming* has to be resolved in advance.
- A set of related REMOTE OBJECTS form together one remote application. In one server process more than one application may run.

- An application has not only a lifetime responsibility for creating the REMOTE OBJECTS, but also for destroying them. If the application terminates, its REMOTE OBJECTS have to be destroyed, too.

Thus, we provide a SERVER APPLICATION whose job it is to initialize and configure the distributed object framework. The SERVER APPLICATION uses the FRAMEWORK FACADE for this task. Moreover, it resolves initial, pre-configured references such as *naming*. Then it instantiates REMOTE OBJECTS that are static instances, or prepares for instantiating other REMOTE OBJECTS, according to their activation strategy.



For each REMOTE OBJECT to be instantiated, the SERVER APPLICATION creates the object according to the activation strategy, activates it in the distributed object framework, and optionally binds it with *naming*.

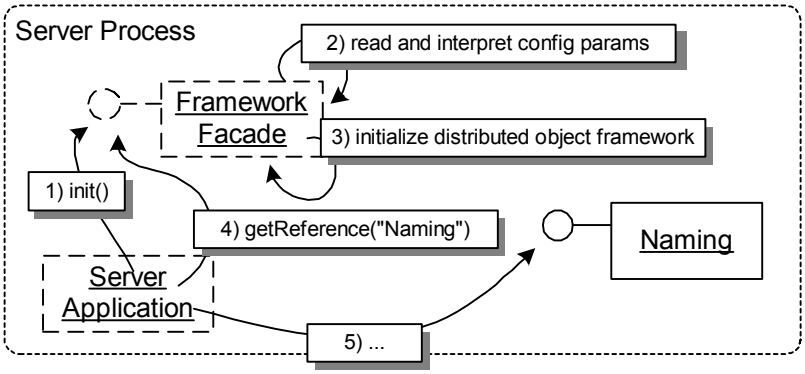
The SERVER APPLICATION bundles REMOTE OBJECTS that conceptually belong together, and handles all object management tasks with respect to the distributed object framework. If a lifecycle manager is used, the SERVER APPLICATION object delegates the actual management of the REMOTE OBJECT'S lifecycle to a LIFECYCLE MANAGER.

Framework Facade

The SERVER APPLICATION has to initialize the distributed object framework and provide access to its COMMUNICATION FRAMEWORK. A distributed object framework is a rather complex piece of software. It needs to coordinate several components and their interactions, all need to be initialized and configured properly and consistently.

Some (simple) way for developers using the distributed object framework is required for configuring the COMMUNICATION FRAMEWORK and interacting with it. Developers also need a way to resolve initial, well-known references to *pseudo objects*. References that must be initially resolved include *naming* and other well-known REMOTE OBJECTS available in a distributed object framework. These "well-known" REMOTE OBJECTS have to be configured somewhere.

As a solution, the distributed object framework provides a central FRAMEWORK FACADE. For developers using the distributed object framework, it serves as the single access point and administration API to the distributed object framework and its services.



A FRAMEWORK FACADE is initialized by a SERVER APPLICATION. It reads and interprets the configuration parameters. Then it initializes the distributed object framework. It also serves as the central place to obtain references to central services, such as *naming*.

The FRAMEWORK FACADE is typically a *pseudo object,* meaning that it behaves like any other managed REMOTE OBJECT but it is not remotely accessible.

A FRAMEWORK FACADE is used in server and client applications. You might want to provide different implementations, as the performance and scalability requirements for servers are usually significantly higher than those for pure clients.

A FRAMEWORK FACADE provides a single access point (a *facade* [GHJV95]) to the distributed object framework. It reduces complexity and shields the constituent parts from direct access. A central FRAMEWORK FACADE avoids multiple initialization and configuration of the same distributed object framework.

However, situations that require (unforeseen) access to the distributed object framework's intricacies are hard to handle, as bypassing the *facade* would be the only way for application developers to deal with such situations.

Communication Framework

In this section we give a brief overview of (patterns of) the COMMUNICATION FRAMEWORK shielded by the distributed object framework. On server side interaction with the COMMUNICATION FRAMEWORK is especially handled by the pattern REQUEST HANDLER.

Usually a COMMUNICATION FRAMEWORK is designed using *layers* [BMR+96]. The lowest *layer* is an adaptation *layer* to the operating system and network communication APIs. Using an adaptation *layer* has the advantage that higher layers can abstract from platform details, and therefore use an platform-independent interface.

The operating system APIs are (often) written in the procedural C language; thus, if the COMMUNICATION FRAMEWORK is written in an object-oriented language, *wrapper facades* [SSRB00] are used for encapsulating the operating system's APIs. The higher layers only access the operating system's APIs via the *wrapper facades.* Each *wrapper facade* consists of one or more classes that contain forwarder operations. These forwarder operations encapsulate the C-based functions and data within an object-oriented interface. Typical *wrapper facades* provide access to threading, socket communication, I/O event handling, dynamic linking, etc.

On top of the *wrapper facades* the COMMUNICATION FRAMEWORK is defined. Connection establishment is handled by the *acceptor/connector* pattern [SSRB00]. The pattern separates the connection initialization from the use of established connections. A connection is created, when the connector on client side connects to the acceptor on server side. On the basis of *acceptor/connector*, connect strategies and concurrency strategies become exchangeable. Once a connection is estab-

lished, further communication is done based on a connection handle, returned from successful connection establishment.

The SERVER APPLICATION starts an event loop, and new connections are handled as events. A *reactor* [SSRB00] reacts on the communication events raised on the connection handles. Its task is to efficiently demultiplex the events and dispatch all requests to connection handler objects in the SERVER APPLICATION.

Concurrency is dealt with using concurrency patterns [SSRB00] (see also [Lea99]). There are two alternatives for synchronizing and scheduling concurrently invoked remote operations:

- *Active object* decouples the executing from the invoking thread. In the case of a distributed object framework, the invoking thread belongs to the REQUEST HANDLER, whereas the executing thread belongs to the INVOKER. A queue is used between those threads to exchange requests and responses.
- A *monitor object* ensures that only one operation runs within an object at a time by queueing operation executions. It applies one lock per object to synchronizes access to all operations.

The mentioned patterns are used within a *half-sync/half-async* architecture. The pattern decouples asynchronous and synchronous processing by defining an asynchronous and a synchronous service processing layer. A queue between these layers maps asynchronous invocations to synchronous execution.

A *component configurator* [SSRB00] supports component configuration and dynamic reconfiguration of components in an application at runtime. Typical components include reusable implementations of common services used in distributed applications, such as *naming* or logging. A *component configurator* uses a *factory* [GHJV95] to create the service objects according to configuration parameters.

CLIENT PROXIES, REQUEST HANDLER, and INVOKERS build together a *broker*, as it is documented in [BMR+96].

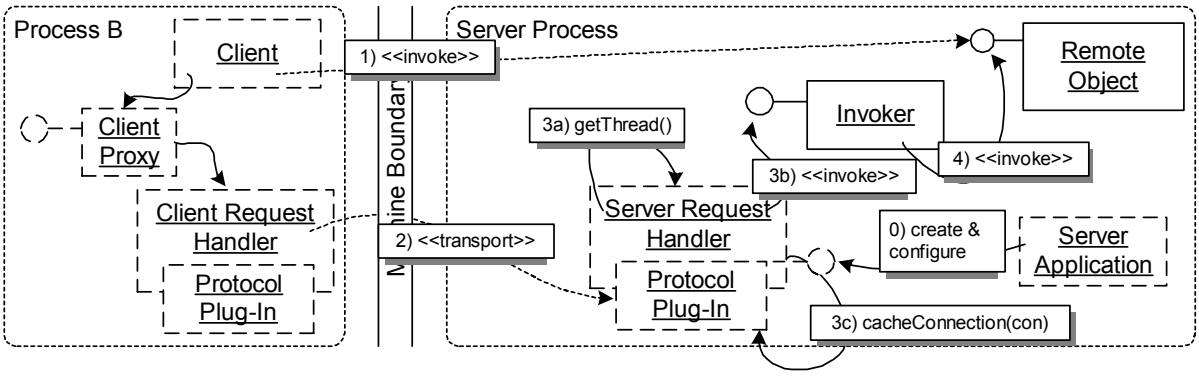
Protocol Plug-in

Usually, a developer using the distributed object framework can abstract from the implementation details of the COMMUNICATION FRAMEWORK. However, there are some situations in which these details matter and it is necessary to provide some way to influence them:

- Sometimes the same application should be able to operate with different protocols. Thus the protocol realization should be exchangeable.
- The developer provides a custom MARSHALLER that provides an optimized serialization mechanism. It is important to make sure that the protocol used by the COMMUNICATION FRAMEWORK can actually transport the serialized data.
- The SERVER APPLICATION needs to fulfil varying QoS requirements. To effectively fulfil these QoS requirements, the facilities provided by the network protocol have to be used differently and perhaps need to be optimized at a rather low level.
- You have to serve many clients. Thus it might be beneficial to optimize the protocol to open a new connection each time a request is transported, because this reduces resource consumption of the SERVER APPLICATION.

Other low-level aspects you might need to take care of are threading, invocation priorities, or caching of certain data. Such issues should be handled transparently for the application logic.

PROTOCOL PLUG-INS are provided as an extension mechanism of the REQUEST HANDLER. The PROTOCOL PLUG-INS handle the low-level networking issues in cooperation with the COMMUNICATION FRAMEWORK and operating system.



The SERVER APPLICATION creates and configures the PROTOCOL PLUG-IN for the REQUEST HANDLER. For an invocation, the CLIENT PROXY transports the message to this PROTOCOL PLUG-IN. The REQUEST HANDLER then forwards the invocation to the INVOKER. It may cache the connection in the PROTOCOL PLUG-IN.

Often, a PROTOCOL PLUG-IN and a custom MARSHALLER go hand in hand, because a specific protocol requires specific marshalling or vice versa.

Although we focused on the server side, the client obviously also requires something that adapts its CLIENT PROXY to the networking details. The protocol used by client and server must match, obviously.

A PROTOCOL PLUG-IN offers an API to customize low-level protocol details of the COMMUNICATION FRAMEWORK. There is no need to integrate these low-level details into the application logic. In principal, protocols can be exchanged transparently. However, it is not always possible to plug-in new protocols without changes in REQUEST HANDLER, INVOKER or SERVER APPLICATION. If more than one protocol are used together, the provided APIs either can only provide a common denominator of these protocols, or the other components have to be aware of the used protocol.

Conclusion

In this paper, we have presented a pattern language consisting of basic infrastructure pattern of distributed object frameworks. These patterns are required for building almost any RPC-based distributed object framework. Moreover, they have to be understood to build applications with these frameworks as well. Note that, for more complex distributed object framework applications many other patterns have to be applied, say, to achieve high performance, scalability, and multi-threading.

Acknowledgements

We wish to thank our Viking Plop 2002 shepherd Kristian Elof Sørensen for his valuable comments on this paper, as well as the participants in the Viking Plop 2002 writers workshop: Kevlin Henney, Michael Pont, Valter Cazzalo, Mikio Aoyama, Juha Pärsinen, and Lars Grunske.

References

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

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frysyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext transfer protocol – HTTP/1.1. RFC2616*, 1999.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GNZ01] M. Goedicke, G. Neumann, and U. Zdun. Message Redirector. In *Proceedings of EuroPlop 2001*, Irsee, Germany, July 2001.

[Jai01] P. Jain. Evictor Pattern , In *Proceedings of 8th Pattern Languages of Programs Conference*, Illinois, USA, Sep. 11-15, 2001

[KJ02] M. Kircher, and P. Jain. Pooling Pattern, In *Proceedings of EuroPlop 2002*, Irsee, Germany, July, 2002.

[Lea99] D. Lea. *Concurrent Java: Design Principles and Patterns*, Second Edition, Addison-Wesley, 1999.

[Mae87] P. Maes. *Computational reflection*. Technical report 87-2, Free University of Brussels, AI Lab, 1987.

[RSB+98] Dirk Riehle, Wolf Siberski, Dirk Bäumer, Daniel Megert, and Heinz Züllighoven. Serializer. In *Pattern Languages of Program Design 3*. Edited by Robert Martin, Dirk Riehle, and Frank Buschmann. Reading, Massachusetts: Addison-Wesley, 1998. Chapter 17, page 293-312.

[SSRB00] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. Patterns for Concurrent and Distributed Objects. Pattern-Oriented Software Architecture. John Wiley and Sons, 2000.

[TS2002] A. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 1995.

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