3. Broker Part One

3.1 Learning Objectives

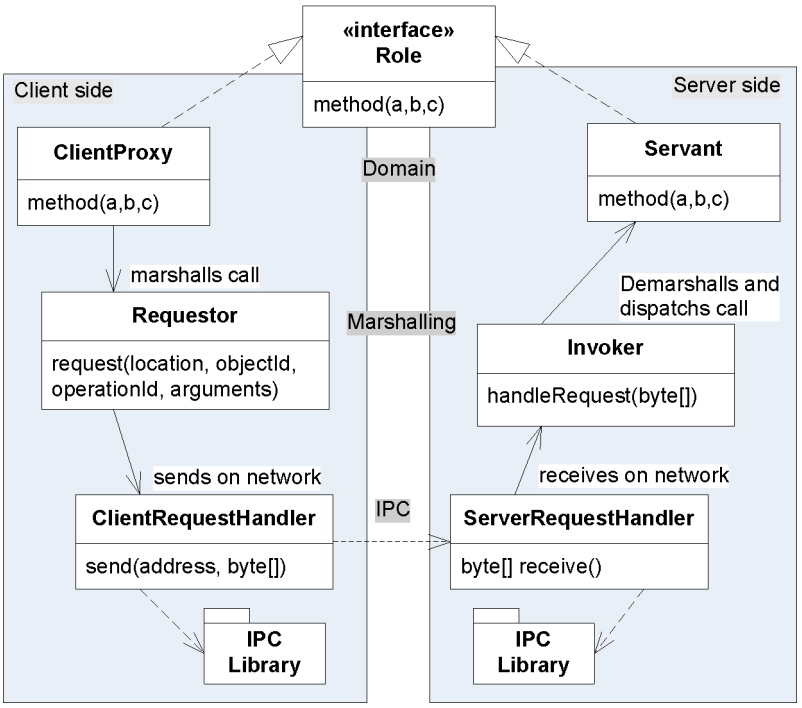
The learning objective of this chapter is the *Broker* pattern which is an architectural pattern that allows methods to be called on remote objects while hiding much of the complexity involved in the fact that these objects are indeed located on different computers connected in a network.

3.2 The Problem

Distributed systems consist of objects that are located on different computers. In the client-server architectural style, distribution is organized with shared objects stored in a single, centralized, server that a large number of clients query and update.

We want to program such client-server architectures in a programming model that is object-oriented, that is, support calling methods in clients on remote objects. The remote method calling shall behave as closely as possible to normal method calls: they are synchronous, they take parameters, and so forth, without the need to use low level distributed programming calls like `send()` and `receive()` methods.

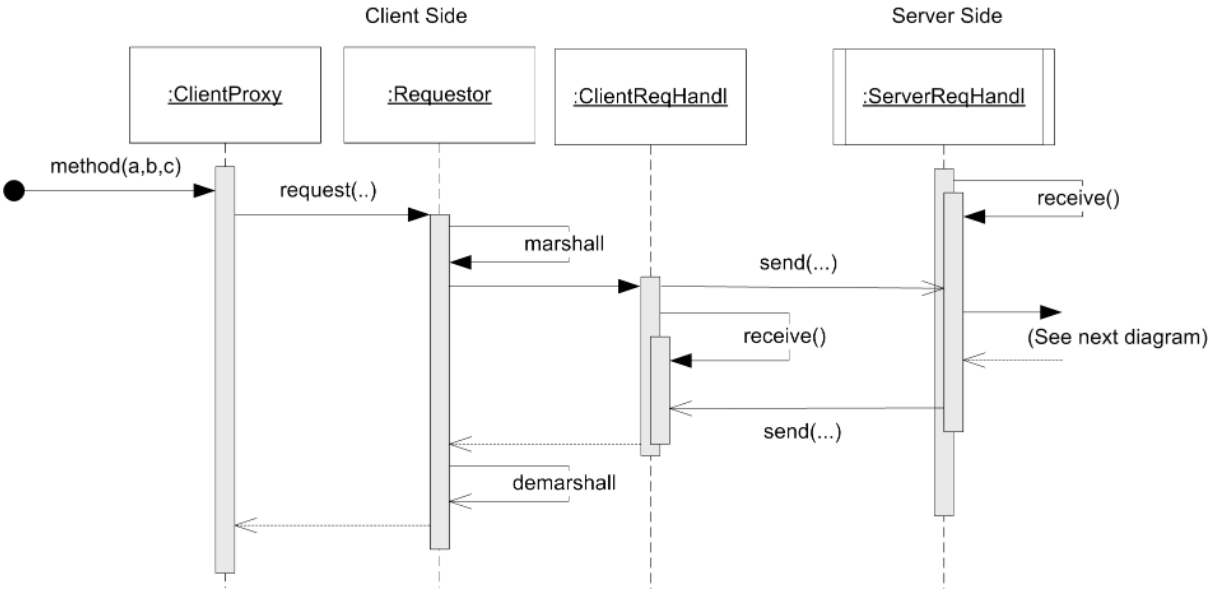
3.3 The Broker Pattern



Broker role structure.

The *Broker* pattern solves the problem of the mismatch between the abstraction wanted: method calls; and the abstractions given: raw byte array message sending and receiving; by defining a chain of six roles, three on the client side and three on the server side, as shown in the figure above.

These roles are at execution time connected by a protocol in which a method call is made on a **ClientProxy** on the client side that acts as a placeholder object for the real object, the **Servant**, on the server side. The protocol dictates that the method call is marshalled first, and next sent to the server, as outlined in the figure below.



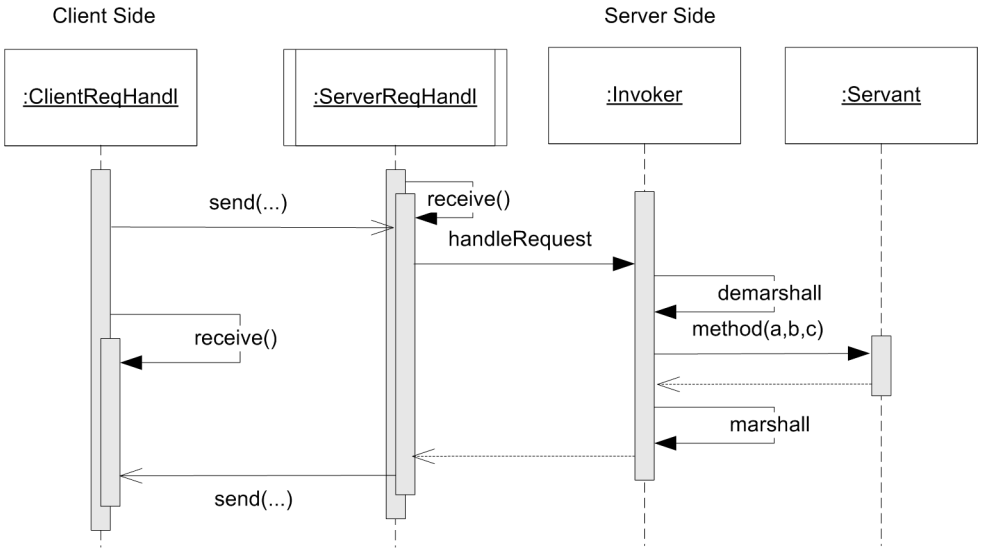
Broker client side dynamics.

The **Requestor**'s responsibility is to do marshalling and execute the send/receive calls - essentially implementing the client side of the request-reply protocol.

If you review the Proxy section in the previous chapter you will note that the send/receive calls are in the **Requestor** in this presentation, not in the **ClientProxy** as outlined there. This is because the algorithm can be expressed in general we thus avoid quite a lot of code duplication.

Its central method is `request(location, objectId, operationName, arguments)` which tells the requester which computer (`location`) to send messages to, which servant object to call (`objectId`), which method to call (`operationName`), and finally the list of all parameters (`arguments`).

Upon reception at the server side, the matching transformation occurs as shown in the figure below.



Broker server side dynamics.

The incoming message is received by the **ServerRequestHandler**, then forwarded to the **Invoker** (method `handleRequest`) that handles the demarshalling to produce the identity of the **Servant** object, the method to call, and the original parameters, and then invoke the method. Finally, a similar process is employed in order to return the computed answer from the **Servant**, the return value of the method, back to the client side and the calling object.

Essentially, the six roles are divided into three layers. The upper layer, the **Role**, **ClientProxy** and **Servant**, form the *domain layer* in which the abstraction level is at that of the domain: The **Role** is an interface representing some kind of role that embody a domain concept that reside on the server but accessed by the clients. Just like TeleMed in our case study.

At the next layer, the *marshalling layer*, the **Requestor** and **Invoker**, are dealing with marshalling, unmarshalling, and dispatching. And finally, the bottom layer is the *inter process communication (IPC) layer*, in which the **ClientRequestHandler** and **ServerRequestHandler** are bound to the operating system and the chosen inter process communication technology, responsible for sending and receiving messages on the network.

The six roles are also "mirrored" in the sense that the client side have the three roles **ClientProxy**, **Requestor**, and **ClientRequestHandler** that are mirrored in the server side's corresponding **Servant**, **Invoker**, and **ServerRequestHandler**.

The responsibilities of the roles involved in the broker are:

ClientProxy

- *Proxy* for the remote servant object, implements the same interface as the servant.

- Translates every method invocation into invocations of the associated **Requestor**'s `request()` method.

Requestor

- Performs marshalling of object identity, method name and arguments into a byte array.
- Invokes the **ClientRequestHandler**'s `send()` method.
- Demarshalls returned byte array into return value(s).
- Creates client side exceptions in case of failures detected at the server side or during network transmission.

ClientRequestHandler

- Performs all *inter process communication* send/receive of data on behalf of the client side, interacting with the server side's **ServerRequestHandler**.

ServerRequestHandler

- Performs all *inter process communication* on behalf of the server side, interacting with the client side's **ClientRequestHandler**.
- Contains the event loop thread that awaits incoming requests from the network.
- Upon receiving a message, calls the **Invoker**'s `handleRequest` method with the received byte array. Sends the reply back to the client side.

Invoker

- Performs demarshalling of incoming byte array.
- Determines servant object, method, and arguments and calls the given method in the identified **Servant** object.
- Performs marshalling of the return value from the **Servant** object into a reply byte array, or in case of server side exceptions or other failure conditions, return error replies allowing the **Requestor** to throw appropriate exceptions.

Servant

- Domain object with the domain implementation on the server side.

3.4 Analysis

Invoking methods on remote objects is a technique that is well supported by many libraries and frameworks, and indeed it is directly supported by Java Remote Method Invocation (RMI). So why spend much time on describing the pattern here?

The answer is twofold. First, the pattern is the inner workings of any such framework and a deep knowledge of this pattern will increase your ability to use such frameworks correctly and efficiently. For instance, REST frameworks that we will discuss in more detail later provide an implementation of the IPC layer (using HTTP) and partial implementation of the marshalling layer (method and object names and parameters are encoded as URIs), and it is thus relatively easy to relate to the *Broker*.

Second, the broker pattern is all about loose coupling between the three levels of responsibility: *let someone else do the dirty job* of inter process communication, and *let someone else do the job* of marshalling. This separation of concern allows us as software architects to choose the right delegates for the task at hand. If I need high performance and horizontal scalability (that is: the server side is not just one server but perhaps 100 servers to cope with very high demands), I will probably pick a message queue system or a web framework with a load balancer in front as basis for my implementation of the **ClientRequestHandler** and **ServerRequestHandler** pair. In a gaming domain, I would pick a binary format for requests and replies and develop the **Requestor** and **Invoker** roles accordingly. For doing TDD or testing, I can use Fake Object IPC implementations and make comprehensive testing without spawning a server - actually I will do just that in the next chapter. And all the while, my **ClientProxy** and **Servant** role implementations would stay unaffected of the underlying implementations. This is obviously also beneficial in the context of agile development of a growing system that may start out simple and easy, but will allow more complex and higher performant variants to be substituted in at later stages in the product's life cycle. These benefits are not available by a standard implementation like Java RMI.

Limitations

The *Broker* pattern outlined above can be used to design client-server systems similar to modern web based designs. However, it is not a full broker system as it is implemented in for instance Java RMI. This is due to a couple of limitations.

- The structure is *pure client-server*, that is, the server can never invoke a method on an object on a client. It is always the client that makes a call.

- The location and identity of the server object must be known to the client as my description above did not include any name service.
- The arguments to any method call are pass-by-value (see Sidebar *Pass-by-Reference and Pass-by-Value* below) as they must be able to marshall and demarshall to produce identical results on both client and server. Pass-by-reference cannot be handled as there is no name service to resolve remote references.

The first issue has the consequence that some designs/protocols cannot be implemented between client and server. A good example of a design our broker cannot implement is the *Observer* pattern in which the **Subject** is on the server side while **Observers** are on the client side(s). This is because the observer implements call-backs: the client makes a state change to the subject which then responds by invoking the `update()` method on all registered observers. This would require the servant object to call methods that are on the client side which our *Broker* cannot handle. Essentially this *Observer* behavior is not a true client-server architecture but a peer-to-peer architecture because suddenly the server acts as client that makes a call to a object hosted on the client, essentially making the client act as a server. The problem can be solved by mirroring the broker pattern roles so both client and server sides have all six roles implemented. But the code and design of course increase in complexity.

Pass-by-Reference and Pass-by-Value

Many programming languages distinguish strongly between *pass by reference* or *pass by value* when it comes to providing arguments to function and method calls. The difference is whether the argument is the value itself (like the integer value 42) or is a reference to a variable holding the value (like the reference to a variable holding the value 42).

The programming languages C and C++ allow both to be expressed:

```
1 void fooByValue(int value) { ... }
2 void fooByRef(int* value) { ... }
```

The call will pass the value 42 to the C function, and thus 'value' is a new variable whose value is a copy. Thus if the formal argument 'value' is modified with the function, it will have no effect on the value of 'v' in the calling code—it is just the copy that changes value. In contrast the call will pass the reference (memory address) of the 'v' variable, and thus if 'value' is changed within the function, it will also change the value of 'v'.

Java provides no such freedom but passes all arguments as values: *pass-by-value*; but with the complication that for class types, it is the object reference that is passed by value, not the object itself — you get a copy of the object reference. Thus if you call methods on this object reference, the original

object will potentially change state (the copied reference points to the same object), however if you change the copied reference itself, it will have no effect on the original object. Yes, a bit tricky indeed...

Exercise: Argue whether the following call String hello = "World"; addHello(hello); will change the 'hello' string, given this implementation:

```
1 private static void addHello(String s) {
2   s = "Hello " + s;
3 }
```

The next two issues are more or less two sides of the same coin. By avoiding a name service in my broker design it becomes a simpler design. However, now the client has to know the location/identity of the server object and the server machine. In the TeleMed case this could be knowing the name of the server, like 'www.telemed.org', as well as having a unique identity of the patient object, like "251248-1234". These two pieces of information are enough to make the call: the client request handler contacts the server at 'www.telemed.org' and the patient id can be used by the invoker to get the right object to review blood pressure for.

The missing name service also means that we cannot pass *object references* **to** the server side, we can only pass basic data types (int, double, arrays, etc.) and string objects ("hello world" is passed as a value to the server, not as an object reference). Remember that an object reference is a reference into a specific address in the computer's memory in which the object is stored. This address of course does not make sense on the server side. If I were to insist on being able to handle pass-by-reference, the requester on the client side could substitute the object reference by an object name that can be registered in the name service. Note that this requires this client side object to be remotely accessible for method calls in case the server decides to call a method on it. On the server side, the invoker should then use the name service to look up that particular object name, get hold of a *ClientProxy* for it as it needs to make remote calls, and then pass a reference to this proxy object up to the servant role. Thus, pass-by-reference both requires a name service and it requires calls to be made from server to client. Thus, this limitation again ties to the previous requirement of only supporting true client-server communication.

It is important to note that this limitation is only an issue with *client side object references*. As I will discuss in Chapter Broker II later, the server can create objects and return "references" back to the client, that it can use to make remote method calls on the server-created objects.

So, my *Broker* is limited compared to Java RMI, and asymmetric in a sense with more constraints on the clients than on the server. But the limitations

I have made are the same as those made in general by large scale web architectures, notably REST, which have achieved world wide success and that today are much more common than Java RMI systems, CORBA, .Net remoting and other full Broker systems. Large scale web systems based upon REST are also only able to pass-by-value, and (for the most part) are purely client-server. And there are good architectural reasons for doing so.

First, a pure client-server architecture scales horizontally: if you need more computing power you can add more servers and use a load balancer, and the clear distinction between the two roles (the client always calls, and the server reacts) are much easier to implement to be secure, highly available, and performant. The location problem is partially mitigated by using URI that uses DNS so servers *can* move to new locations even if their DNS names are hard coded.

Failure Modes

A remote method call is architecturally significantly different from a local method call. It is always *much* slower. And unless extreme measures have been taken, some day it will simply fail — no computer operates reliably indefinitely. To create highly available systems, such failure modes must be anticipated and their effects mitigated.

This means all method calls to a **ClientProxy** must be scrutinized and coded with attention — they are what Nygard[1] terms *integration points*, that is, points in the code where two systems integrate and thus will cause failures at some time. The same goes, of course, for the server side, that may not be able to transmit the reply because the client computer crashed after sending the initial request.

One simple complication is that our standard failure handling mechanism, i.e. throwing exceptions, of course does not propagate across networks. For example, if the TeleMed server tries to store a document in XDS but the connection is lost, an exception will be thrown in the server, but it will propagate only to the server request handler implementation. As a thrown exception is also a return value, it must be caught in the invoker, marshalled, and sent to the client side. In turn, the **Requester** must identify the reply as an exception, and then throw a suitable client side exception to inform the domain code there, that some unusual situation has occurred.

Failure handling is a large, architectural, topic and beyond the scope of this book, and I refer to the books outlined in the end of the chapter. The most basic recipe for handling failures is to analyze each remote call site (the *integration points*) thoroughly in order to catch and handle all exceptions

[1]Michael T. Nygard, "Release it - Design and Deploy Production-Ready Software, 2nd ed.", Pragmatic Bookshelf, 2018.

gracefully, to ensure all server side exceptions are handled and acknowledged to the client side, and importantly ensure that the server will survive.

Marshalling Robustness

The **Requestor** and **Invoker** must of course agree on the format of the transmitted message: the marshalling format. As mentioned earlier, many libraries exist for converting ordinary Java objects into, say, JSON and back again. This is fast and convenient, and I also use it in the TeleMed implementation in the next section. However, for production quality software it is important to adhere to Jon Postel's *robustness principle*:

> *Be conservative in what you do, be liberal in what you accept from others.*[2]

In other words, marshalling data to be read by other machines (or by other programs on the same machine) should conform completely to the specifications, but code that receives input should accept non-conformant input as long as the meaning is clear.

This principle requires a bit more design and programming at the onset but it pays back as time goes, and services are updated as new features are added. To see this, consider a situation in which developers continue work on an existing system, using version 1 of a marshalling format, that is in production. They have to change the marshalling format to a new version, version 2, as their development of new features and bugfixes progress. They of course test their development intensively, and it works fine in their staging environment, as they of course run both server and clients using the updated code base. But once they put it into production, they discover that servers crash. This is because the server now expects version 2 of incoming messages whereas all already deployed clients send messages using marshalling format version 1.

One important lesson is the following:

> **Key Point: Include Format Version Identity.** Always include version identity in the marshalling format.

Then the demarshalling code can do a first lenient demarshalling to look for the version identity in the received message; and then next based upon that, choose the appropriate demarshalling algorithm that fits the particular format. If no such algorithm exists, then at least report the error gracefully to the users or system administrators, so appropriate actions can be taken.

One obstacle is that marshalling libraries for convenience often marshalls to and from objects using their type as template. For instance, GSON allows you to write:

[2] Postel, Jon, "Transmission Control Protocol. IETF.", RFC 761, Jan 1980.

```
1       TeleObs a = new TeleObs("251248-1234", 128, 76);
2       String aAsJson = gson.toJson(a);
3       TeleObs b = gson.fromJson(aAsJson, TeleObs.class);
```

While this is effective and convenient for the programmer, it unfortunately also implies that once the marshalling format changes (because developers have change the `TeleObs` class in a new release), the `fromJson()` method call will throw an exception if the supplied JSON string uses the old format.

You will have to dig into the concrete library in order to explore its options for a more lenient handling of marshalling.

3.5 Summary of Key Concepts

The *Broker* architectural pattern defines the architecture for allowing client objects to invoke methods on server objects which are located on different machines or in different processes. The benefit is that we can program using the object-oriented paradigm instead of having to program using low level network `send()` and `receive()` calls. Still, a remote method call is likely to fail, and thus attention must be made to handling failure situations.

The *Broker* pattern dictates six roles organized in three layers of abstraction: the domain layer (roles **ClientProxy** and **Servant**) which is defined by the objects that implement the domain interface **Role**; the marshalling layer (roles **Requestor** and **Invoker**) which handle converting objects, methods, and parameters to and from a marshalling format that is suitable for transmission on a network; and the inter process layer (roles **ClientRequestHandler** and **ServerRequestHandler**) which handles network traffic and threading issues.

The outlined *Broker* is the core of Java RMI and .NET remoting, but is simpler in the sense that it is purely client-server: no method call from the server side to an object on the client side is possible.

My presentation of the is highly influenced by the patterns described in "Pattern-Oriented Software Architecture, Volume 4"[3]. Patterns to deal with failure modes are presented by Nygard[4].

3.6 Review Questions

Draw the UML diagram of the *Broker* pattern. Explain what each role's responsibilities are. Draw a sequence diagram of how a single method call on

[3]Frank Buschmann, Kevin Henney, and Douglas C. Schmidt, "Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing Volume 4 Edition", Wiley, 2007.

[4]Michael T. Nygard, "Release it - Design and Deploy Production-Ready Software, 2nd ed.", Pragmatic Bookshelf, 2018.

the client side is executed using the *Broker* patterns. Explain the limitations of the present discussion of the pattern.

Architecture Pattern: Broker

Intent

Define an loosely coupled architecture that allows methods to be called on remote objects while having flexibility in choice of operating system, communication protocol, and marshalling format.

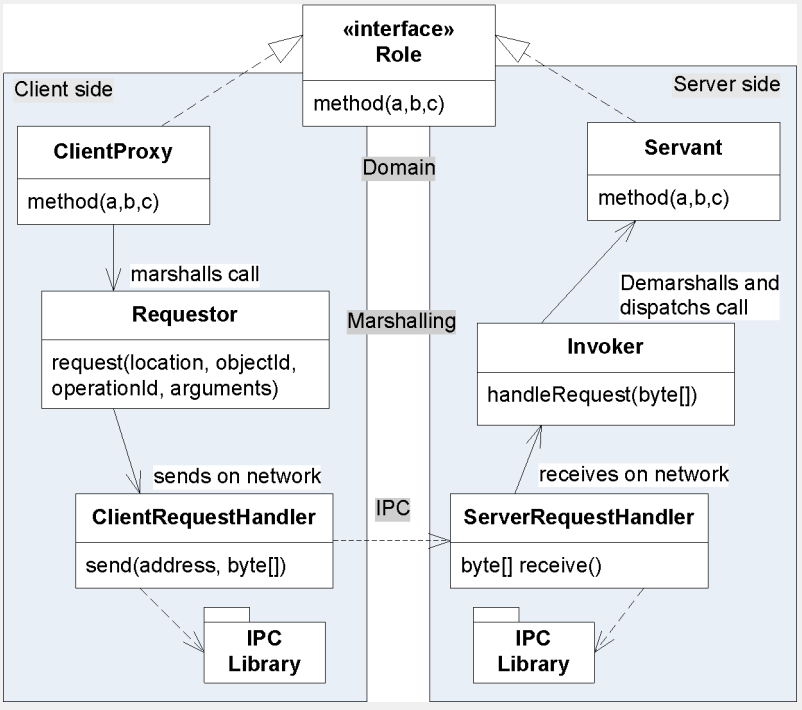
Problem

We want to program using the object-oriented paradigm but some objects are distributed on a remote machine.

Solution

The remote object must implement an interface. The domain implementation of it on the server is the **Servant**, while the client communicate with it using a **ClientProxy** implementation. The **ClientProxy** and **Servant** implementations are coupled with layers that handle *marshalling* and *inter process communication*.

Structure



Broker role structure.

Cost-Benefit

The benefits are: *loose coupling between all roles* that provides flexibility in choosing the marshalling format, the operating system, the communication

protocol, as well as programming language on client and server side. The liabilities are: *complex structure* of the pattern.