Chapter 5

Distributed Objects

Organizing a distributed application as a set of objects is a powerful architectural paradigm, which is supported by a number of infrastructures and tools. The development of middleware has in fact started with the advent of distributed object systems. This chapter presents the main patterns related to distributed objects, and illustrates them with two systems that are representative of current technology, Java RMI and CORBA. The main software frameworks used in the implementation of these systems are presented, with reference to open source implementations.

5.1 Distributing Objects

Organizing a distributed application as a (dynamically evolving) set of objects, located on a set of sites and communicating through remote invocations (2.2.2), is one of the most important paradigms of distributed computing, known as the distributed objects model. In this section, we first present an overview of this model (5.1.1). We next introduce the main interaction scheme between objects, the remote object call (5.1.2). We finally present the user's view of this mechanism (5.1.3).

5.1.1 Overview

The distributed objects model brings the following expected benefits.

- Application designers may take advantage of the expressiveness, abstraction, and flexibility of an object model (as described in 2.2.2).
- Encapsulation allows an object's implementation to be placed on any site; object placement may be done according to specific criteria such as access locality, administration constraints, security, etc.
- Legacy applications may be reused by encapsulating them in objects, using the Wrapper pattern (2.3.3).
- Scalability is enhanced by distributing processing power over a network of servers, which may be extended to accommodate an increasing load.

Note that, in the remote objects model, objects are the units of distribution, i.e. an individual object resides on a node in its entirety (Figure 5.1).

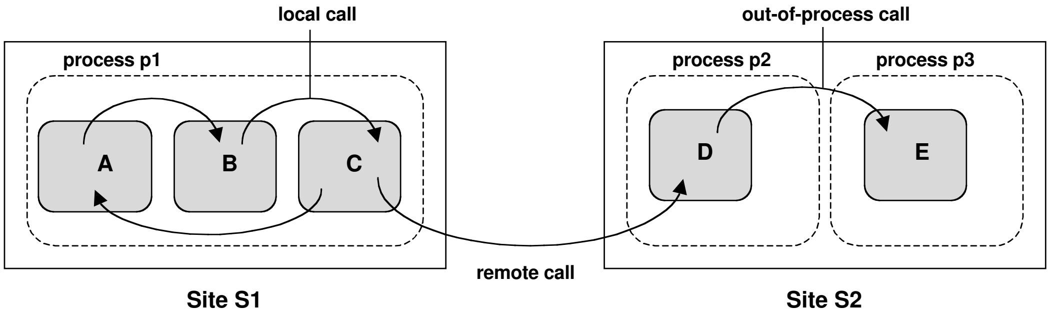

Figure 5.1. Distributed objects

Other models for distributing objects have been proposed, such as:

- The *fragmented objects* model, in which an object may be split in several parts, located on different nodes, and cooperating to provide the functionality of the object. An example of a fragmented object is a distributed binding object (3.3.2). An example of a system using this model is Globe [van Steen et al. 1999]. This model will not be considered further in this book.
- The *replicated objects* model, in which several copies, or replicas, of a given object may coexist. The motivation for replicated objects is to increase availability and to improve performance. However, the replicas of an object must be kept consistent, which entails additional cost. Replicated objects are examined in Chapter 11.
- The *migratory* (or *mobile*) objects model, in which an object may move from one node to another one. Object mobility is used to improve performance through load balancing, and to dynamically adapt applications to changing environments.

These models may be combined, e.g. fragmented objects may also be replicated, etc.

A distributed application using remote objects is executed as a set of processes located on the nodes of a network. An object's method is executed by a process or a thread (in some models, objects may also be shared between processes), and may include calls to other objects' methods. For such inter-object method calls, three situations may occur (Figure 5.1).

- The calling and called objects are in the same process (e.g. objects A and B): this is a *local invocation*.
- The calling and called objects are executed by different processes on the same site (e.g. objects D and E): this is an *out-of-process invocation*.
- The calling and called objects are on different nodes (e.g. objects C and D): this is a *remote invocation*.

Local invocations are done like in a non-distributed object system. Non-local forms of invocation rely on an *object request broker* (ORB), a middleware that supports distributed objects. This term has been introduced for the CORBA architecture, but it applies to other object systems as well. An ORB has the following functions.

- Identifying and locating objects.
- Binding client to server objects.
- Performing method calls on objects.
- Managing objects' life cycle (creating, activating, deleting objects)

In the rest of this section, we give a first outline of the operation of a non-local invocation. More details on the internals of an ORB are given in Sections 5.2 and 5.3.

5.1.2 Remote Object Call: a First Outline

An application using remote objects is typically organized using the client-server model: a process or thread executing a method of a client object sends a request to a (possibly remote, or out-of-process) server object in order to execute a method of that object.

The overall organization of a method invocation on a remote object, shown on Figure 5.2, is similar to that of an RPC, as described in Chapter 1. It relies on a stub-skeleton pair. In contrast with RPC, the stub and the skeleton are objects in their own right. Take the example of the call from object C to the remote object D. The stub for D, on client C's site, acts as a local representative, or proxy, of object D. It therefore has the same interface as D. It forwards the call to D's skeleton on D's site, which performs the actual method invocation and returns the results to C, via the stub.

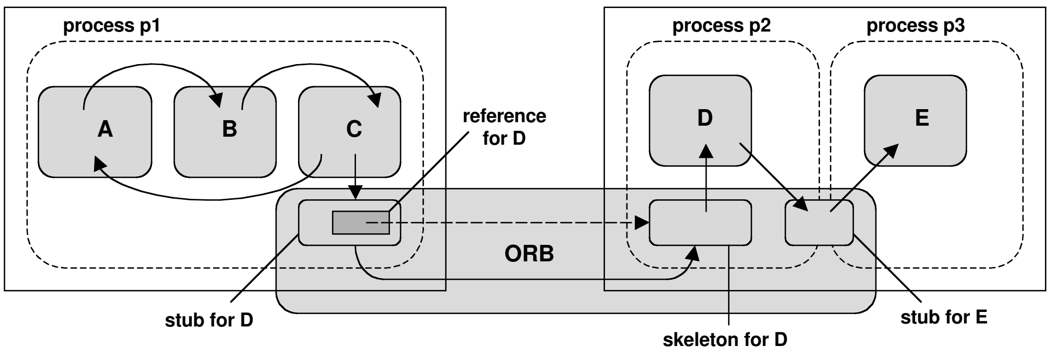

Figure 5.2. Performing non-local calls

In order to be able to forward the invocation, D's stub contains a reference to D (more precisely, to D's skeleton). A *reference* to an object is a name that allows access to the object (cf. 3.1); this name is therefore distribution-aware, i.e. it contains information allowing the object to be located (e.g. network address and port number). Object references are further developed in 5.2.1.

An out-of-process call on the same node (e.g. from object D to object E) could in principle be performed as a remote invocation. However, it is possible to take advantage of the fact that the objects are co-located, using e.g. shared memory. Thus an optimized stub-skeleton, bridging the client and server address spaces, is usually created in that case.

Let us go into the details of a remote method invocation. The client process invokes the method on the local stub of the remote object (recall that the stub has exactly the same interface as the remote object and contains a reference to that object). The stub marshalls the parameters, constructs a request, determines the location of the remote object using its reference, and sends the request to the remote object (more precisely, to the object's skeleton). On the remote object's site, the skeleton performs the same function as the server stub in RPC: unmarshalling parameters, dispatching the call to the address of the invoked method, marshalling returned values, and sending them back to the stub. The stub unmarshalls the returned values and delivers them to the client process, thus completing the call. This is represented on Figure 5.3.

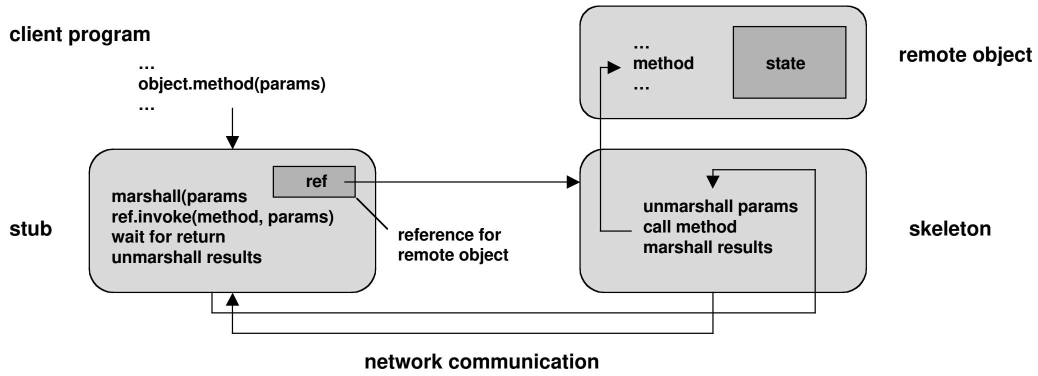

Figure 5.3. Invoking a remote object

An important difference with RPC is that objects are dynamically created (details in next section).

5.1.3 The User's View

Remote object infrastructures attempt to hide distribution from the user, by providing access and location transparency. However, some aspects of distribution remain visible, e.g. object creation through factories.

According to the encapsulation principle, an object, be it local or remote, is only accessible through an interface. Object interfaces are described by an *Interface Description Language* (IDL). The function of an IDL for objects is similar to that of the IDL used in RPC systems. A set of IDL descriptions is used

- to describe the interfaces of the objects being used by an application, thus helping the design process, allowing consistency checks, and providing useful documentation;
- to serve as input for stub and skeleton generation, both static (before execution) and dynamic (during execution).

There is no single format for an IDL. The syntax of most IDLs is inspired by that of a programming language. Languages that include interface definitions, such as Java and C#, define their own IDL.

An example of a general purpose IDL is the OMG IDL, used for programming in CORBA. This IDL may be "mapped" on various programming languages by using appropriate generation tools. For example, using the IDL to C++ mapping tools, stubs and skeletons in C++ may be generated from IDL description files. Client and server executable programs may then be generated, much like in an RPC system (1.3.3).

The main distribution-aware aspect is object creation and location. Creating a remote object cannot be done through the usual object instantiation mechanism, which involves memory allocation and is not directly applicable to a remote node. Creating a remote object is done through an *object factory* (2.3.2), which acts as a server that creates objects of a specified type. The reference of the created object is returned to the client (Figure 5.4 (a)).

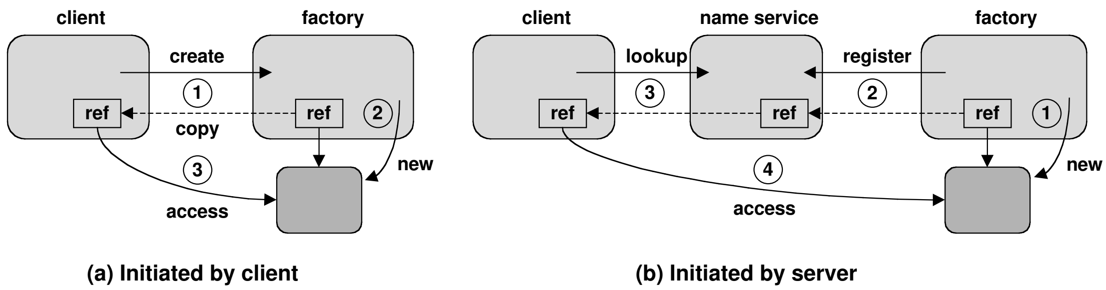

Figure 5.4. Object creation

An object may also be created at the server's initiative. In that case the server usually registers the object (more precisely, a reference to the object) in a name service, to be later retrieved by the client (Figure 5.4 (b)). This is an instance ot the general binding mechanism described in 3.3.4.

As a conclusion, a remote object may only be accessed by a client program through a reference, which in turn may be obtained in several ways: as a return parameter of a call, through an object factory, through a name service. The first two ways imply access to existing objects that in turn have to be located; thus, ultimately, retrieving an object relies on a name service. Examples are described in the case studies.

5.2 Remote Object Call: a Closer View

A first outline of the execution of a remote object call is given in Section 5.1.2. A few points need to be further clarified in this scheme.

- What information should be contained in an object reference?
- How is the remote object activated, i.e how is is associated with a process or thread that actually performs the call?
- How are parameters passed, specially if the parameters include objects?

These points are considered in the rest of this section.

5.2.1 Object References

In Chapter 3, we mentioned that the two functions of a name (identification and access) have conflicting requirements, which leads to use different forms of names for these functions. We consider here the means of providing access to an object; an information that fulfills this function is called an *object reference*.

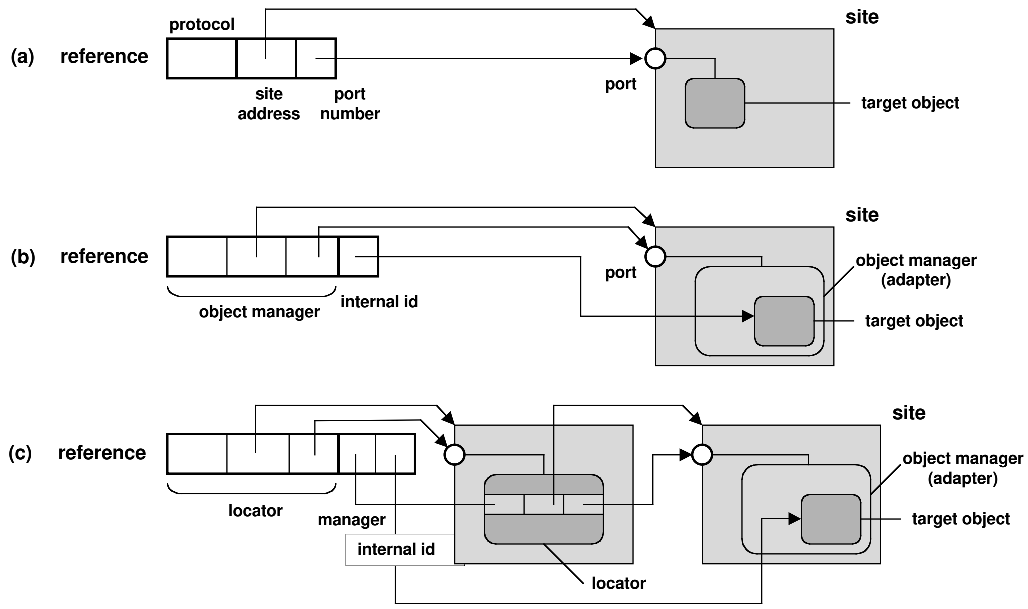

Figure 5.5. Object references

Recall that an object may only be accessed through its interface. Therefore an object reference must also refer to the object's interface. Since an object may be remote, the reference must provide all the information that is needed to perform a remote access to the object, i.e. the network location of the object and an access protocol.

Figure 5.5 shows three examples of reference implementations. In the simplest case (a), the location is a network address (e.g. host address and port number). However, this scheme lacks flexibility, since it does not allow the target object to be relocated without changing its reference. Therefore, indirect schemes are usually preferred. In example (b), the reference contains the network address of a server that manages the object (an object adapter), together with an internal identification of the object on this adapter – more on this in Section 5.2.3. To provide additional flexibility (e.g. allowing a reference to remain valid after the shutdown and restart of a server), an additional level of indirection may be introduced. In example (c), the reference contains the network address of a locator for the adapter, together with the identity of the adapter and the internal object identification. This allows the adapter to be transparently relocated on a different site, updating the locator information without changing the reference.

Recall that a stub, i.e. a proxy (2.3.1) for a remote object, contains a reference to the object it represents. Actually, the stub itself may be used as a reference. This has the advantage that the reference is now self-contained, i.e. it may be used anywhere to invoke the remote object. In particular, stubs may be used for passing objects as parameters. This aspect is further developed in Section 5.2.4.

Object references may be obtained through several mechanisms.

- When an object is created, the object factory returns a reference to the object;
- The `bind` operation on the name of an object (Chapter 3), if successful, returns a reference to the bound object;
- As a special case of the above, looking up an object in a name server or trader returns a reference to the object (if found).

Two points should be noted regarding object references.

The validity domain of a reference. A reference is not usually universally valid in space and time.

In a closed system relying for instance on a local area network using a fixed protocol suite, object references may use a restricted format, in which some elements are implicit (for example the protocols used). Such references may not be exported outside the system. However, if references are to be passed across interconnected systems using different protocols or representations, then their format must accommodate the specification of the variable elements. For example, the Interoperable Object Reference (IOR) format, used in CORBA systems (5.5), allows for communication between objects supported by different ORBs.

In addition, a reference to an object is only valid during the lifetime of the object. When an object is removed, any reference to it becomes invalid.

Reference vs identity. Recall that an object reference is not intended to identify an object, but only to provide a means of accessing it. Therefore a reference cannot in general be used as an identifier (e.g. different references may give access to the same object). Object identification must be dealt with separately (e.g. an object may carry a unique identifier as part of its state, and a method may be provided to return this identifier[1]).

One could imagine including a unique identification for an object in a reference. However, this would defeat the primary purpose of a reference, which is to give access to an object that provides a specified functionality. To understand why, consider the following scenario: Suppose *ref* is a reference for object O; it contains the network address and port number of a location server, plus a key for O on this server. The key is associated with the actual network location of O (e.g. again network address and port number). Suppose the server that supports O crashes. A fault tolerance mechanism is activated, which updates the location server with a new copy of O (say O_1). While O and O_1 are different objects, as regards identity, they are equivalent from the client's point of view (provided consistency is indeed ensured), and the reference *ref* may at different instants give access to either of them.

[1]note that, in this case, we need access to the object in order to check its identity.

5.2.2 The Object Invocation Path

Remote object invocation is actually somewhat more complex than described in Section 5.1.2, because of two requirements.

- Abstraction. The actual service described by the remote invocation interface may be provided by a variety of concrete mechanisms, which differ by such aspects as activation mode (whether a new process or thread needs to be created), dynamic instance creation (whether the target object already exists or needs to be created at run time), persistence (whether the target object survives through successive executions), etc. This results from the encapsulation principle, as presented in 2.2.2: a service described by an abstract interface is implemented by an object whose specific (concrete) interface depends on the details of the service provision mechanism. In other words there is a distinction between a server (a site that provides a set of services to remote clients), and a *servant* (an object, managed by a server site, that actually implements a specific service).

- Portability and Interoperability. A distributed application may need to be ported to various environments, using ORBs provided by different vendors. An application may also involve communication between different ORBs. Both situations call for the definition of standards, not only for the interface between client and server, but for internal interfaces within the ORB.

As a result, the overall remote invocation path is organized as shown on Figure 5.6. This is a general picture. More details are provided in the case studies.

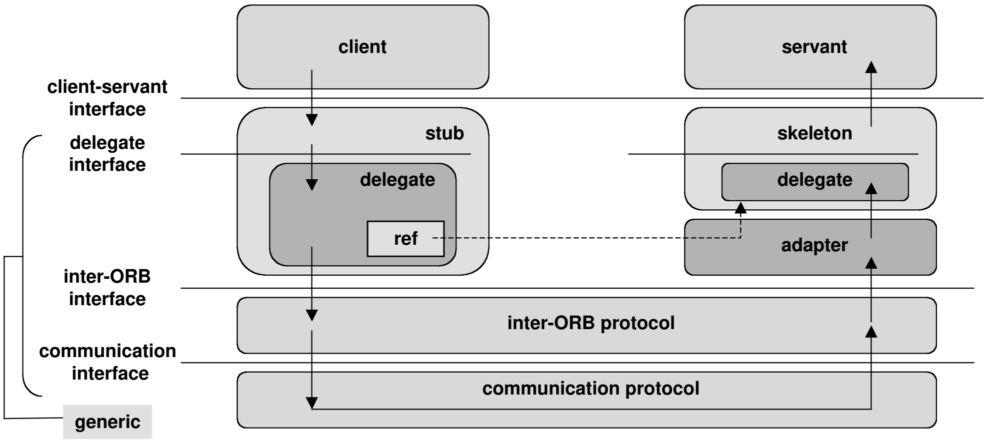

Figure 5.6. The invocation path and its main interfaces

The motivations for this organization result from the above stated requirements.

On the client side, a new interface is defined inside the stub. While the "upper" interface of the stub is application-specific (since it is identical to that of the remotely called object), this internal interface is generic, i.e. application-independent. This means that both the interface entry points and the format of the data may be standardized.

Typical methods provided by the delegate interface are `create_request` (constructing a invocation request in a standard form) and `invoke` (actually performing the invocation).

Likewise, the "lower" interface of the stub, connecting it to the network, is generic. An example of a generic operation is `SendRequest`, whose parameters are the reference of the called object, the name of the method, the description of the parameters of the called method. An example of a standard is GIOP (General Inter-ORB Protocol), further described in Section 5.5.1.

On the server side, the interface transformation between service (as described by a generic ORB interface) and servant (a specific, application-dependent interface) is again done through a server delegate, part of the skeleton. The skeleton itself (and thus the servant) is located through a piece of software called an *object adapter* (2.3.3). While delegates are part of the internal organization of a stub, and are never explicitly seen by a user, adapters are usually visible and may be directly used by applications.

5.2.3 Object Adapters

An adapter performs the following functions.

- to register servant objects when they are created or otherwise installed on the server;
- to create object references for the servants, and to find a servant object using its reference;
- to activate a servant object when it is called, i.e. to associate a process with it in order to perform the call.

There are three main ways of installing a servant object on a server. The first way consists of creating the servant, using an object factory. The second way consists of importing the servant from another site (assumed it is available there), by moving or copying it to the server site. The third way consists of constructing the object from its elementary parts, i.e. a state and a set of functions (methods) operating on the state. This latter mode is used when an existing (or *legacy*) application, not written in object style, must be reused in an object-oriented setting. In that case, the legacy application needs to be "wrapped up", i.e. its contents needs to be encapsulated to make only visible an object-style interface. This is another instance of interface transformation, which is again performed by an adapter (this explains why adapters are also called *wrappers*).

Several different adapters may coexist on a server, for example to provide different policies for process activation. A servant object may now be identified by providing an identification for its adapter together with an internal identification of the object within the adapter. This information is part of the object's reference.

The operation of a remote object invocation may now be refined as shown on Figure 5.7.

Starting from the stub as above, the call is directed to the adapter, providing the internal identification of the servant object within the adapter. The adapter must locate the object (more precisely, the object's skeleton), activate the object if needed, and forward the call to the skeleton. The call is then performed as described in Section 5.1.2.

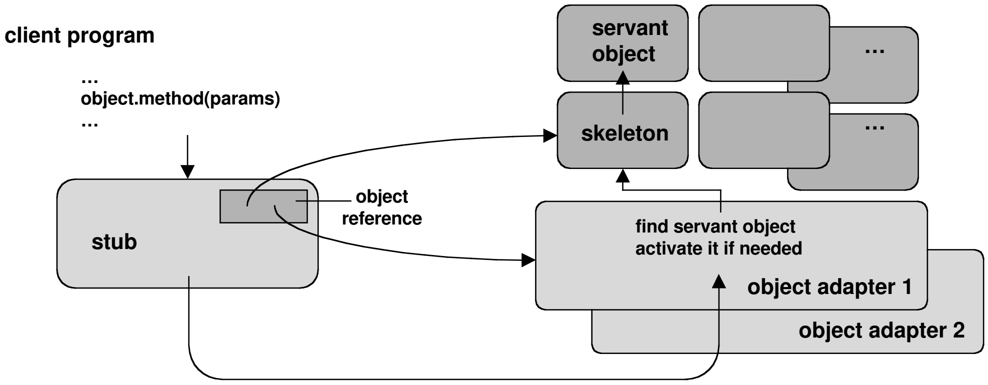

Figure 5.7. Invoking a remote object through an adapter

5.2.4 Parameter Passing

Passing parameters in a remote object system poses two kinds of problems.

- Transmitting values of elementary types through a network. This is the "marshalling-unmarshalling" problem.
- Passing objects as parameters. This raises the issue of the passing mode (reference vs value).

The marshalling problem is similar to that found in RPC systems (1.3.2). It is solved by creating marshallers and unmarshallers for the elementary types, using a serializable format (a byte sequence) as the intermediary form.

We now consider passing objects as parameters. Since the standard way of calling a method on a remote object is through a reference to it, the usual mode of passing an object as a parameter is by reference (Figure 5.8 (a)). The reference may have any of the formats presented in 5.2.1, including a stub. It needs to be in a serializable form to be copied on the network.

If the object being passed is on the site of the calling method (Figure 5.8 (b), then any access to that object involves a remote invocation from the called site to the calling site. If this access does not modify the object, one may consider passing the object by value, i.e. sending a copy of it on the called site (Figure 5.8 (c). However, this implies that the state of the object may indeed be marshalled and copied on the network[2]. Choosing the mode of passing for a (read-only) local object is a trade-off between the cost of marshalling and sending the copy of the object and the cost of remotely invoking the object from the called site, which depends on the size of the object's state and on the frequency of access. Some aspects of this trade-off are discussed in [Spiegel 1998].

[2] For example, in Java, an object must implement the `java.io.Serializable` interface to allow its state to be marshalled.

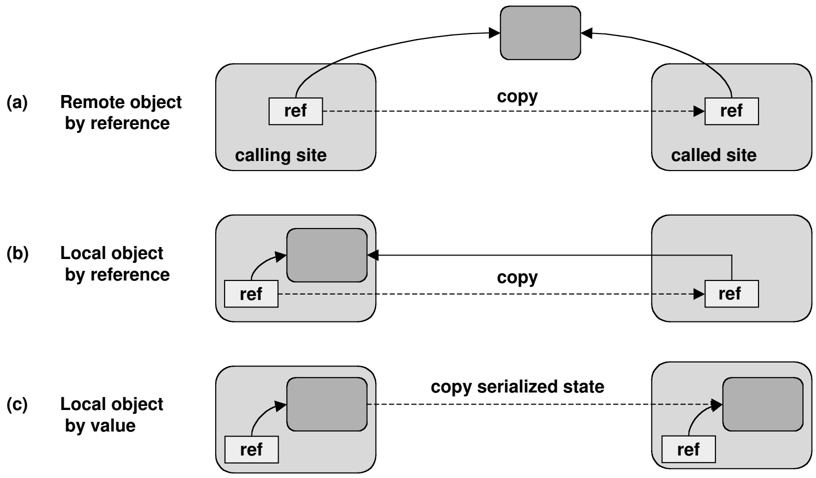

Figure 5.8. Passing an object as a parameter

5.3 Inside an Object Request Broker

In this section, we present a detailed view of the internal operation of an object request broker, emphasizing the binding and communication aspects. While this description is inspired by the organization of the Jonathan ORB, we attempt to present a generic view of the structure and operation of an ORB. Specific examples are presented in Sections 5.4 and 5.5.

A thorough treatment of the patterns involved in remote invocation may be found in [Völter et al. 2004].

5.3.1 The Mechanics of Object Invocation

The use of generic (i.e. application-independent) interfaces in the invocation path to a remote object has been motivated in 5.2.2. The main generic interface has been specified by the OMG in the General Inter-ORB Protocol (GIOP). While this specification is independent of the underlying transport layer, its dominant implementation, called IIOP (Internet Inter-ORB Protocol), is built over TCP/IP.

The initial purpose of GIOP was to allow different CORBA implementations to interoperate, leaving each ORB vendor free to choose an internal transport protocol. However, in practice, GIOP (essentially under the IIOP form), is also used for the internal implementation of ORBs (both CORBA and others such as Java RMI).

We do not intend to describe the full GIOP specification (see [OMG 2003]). We only identify the main elements that are needed to understand the basic mechanics of object binding and invocation.

The GIOP specification covers three aspects.

- A common format for the data being transmitted, called Common Data representation (CDR);

- A format for the messages used for invoking remote objects;
- Requirements on the underlying transport layer.

The IIOP specification defines a mapping of GIOP on the TCP/IP transport protocol. In particular, it defines a general format for object references, called Interoperable Object References (IOR).

Eight message types are defined for object invocation, The two most important are `Request` (from client to server), and `Reply` (from server to client). The other messages fulfill auxiliary functions such as aborts, error detection, and IOR management.

A `Request` message includes a reference for the target object, the name of the invoked operation, the parameters (in marshalled form), and, if a reply is expected, an identifier for a reply holder (the endpoint on which the reply is expected). After sending the message, the calling thread is put to wait. At the receiving end, the servant is located using the reference and the request is forwarded to it.

A `Reply` message, only created if the invoked operation returns a value, contains that value, in marshalled form, together with an identification of the reply holder. When this message is received by the client, the waiting thread is activated and may retrieve the reply from the reply holder.

Note that the `Request` message may either be created by a stub, itself generated prior to the invocation, or created dynamically by directly putting its parts together at invocation time. Here we only consider the first case; dynamic request creation is examined in Section 5.6.

An outline of the invocation path may be found on Figures 5.9 and 5.10, which describe the client and server sides, respectively. The operation of the transport protocol used by the GIOP layer is not detailed (see Chapter 4).

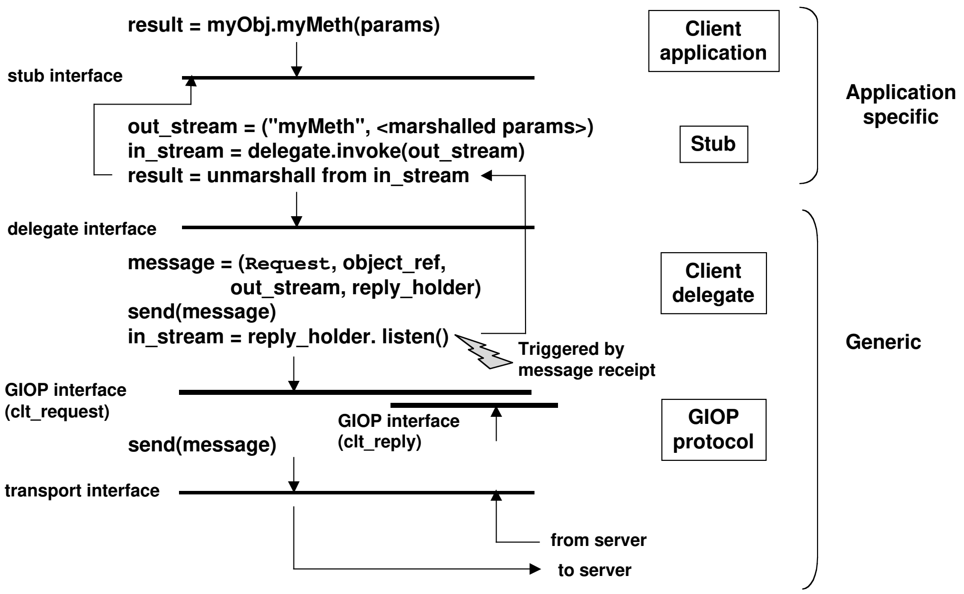

Figure 5.9. Object invocation: client side

Above the GIOP layer, the invocation path goes through a stub (at the client end) and a skeleton (at the server end). The upper interface of the stub and the skeleton is application specific. Inside the stub and the skeleton, it is convenient to define an additional generic (application-independent) interface, to improve code reusability, leaving the application-specific part to a minimum. Therefore, in several ORB organizations, the stub and the skeleton are separated into an (upper) application-specific part (the stub or skeleton proper) and a (lower) application-independent part called a *delegate*. The interface of the delegate reifies the application-dependent interface, giving an explicit representation of the operation name and the (marshalled) parameters.

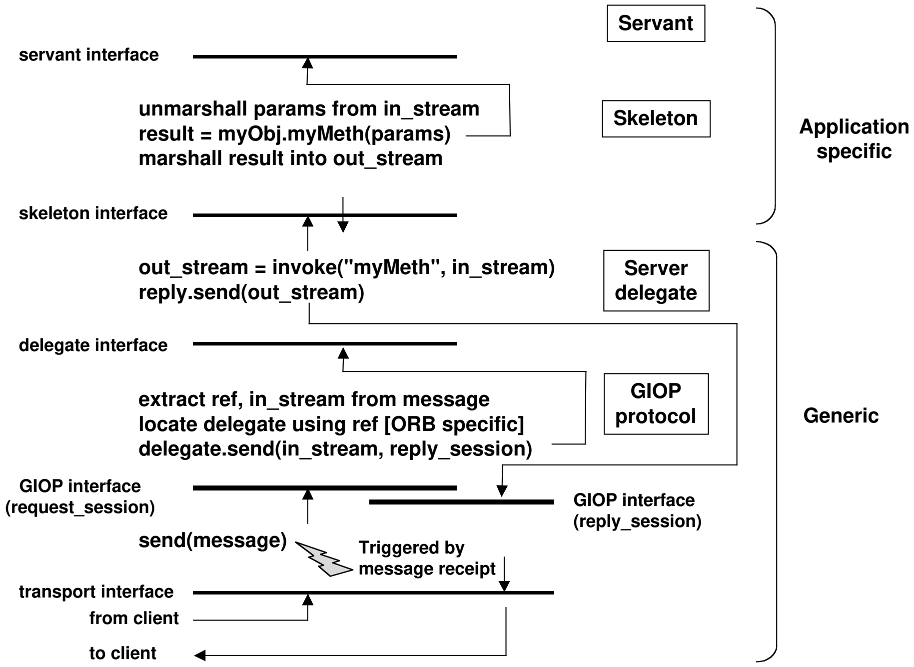

Figure 5.10. Object invocation: server side

An important aspect is locating the servant at the server's end, using the object reference sent in the `Request` message. This an ORB-specific issue, which is linked to the organization of object references. For example, Java RMI directly uses a [host address, port number] reference, while CORBA goes through an object adapter. More details on servant location are provided in the case studies.

5.3.2 Binding in an ORB

In the previous section, we have described the (statically generated) invocation path in an ORB. The components of this path (stub, skeleton, delegates, transport protocol endpoints) collectively form a binding object between the client and the servant.

Classes for the stub and skeleton of a remotely used object are generated from a description of the interface of this object. This description is expressed in an Interface Description Language (IDL), as explained in Section 5.1.3.

Instances of these classes are created before invocation, using appropriate stub, skeleton and delegate factories. The main information to be provided is the object reference. This is done using the `export-bind` pattern.

- `export`: a servant object is created, directly or through a factory; it is registered by the server (possibly using an adapter), thus providing a reference. This reference is then registered for future use in a name service. Parts of the binding object (the skeleton instance, and possibly the delegates) are also created at this time.
- `bind`: the client retrieves a reference for the servant, either through the name service or by any other means such as direct transmission from the server. It uses this reference to generate an instance of the stub, and to set up the path from client to server by creating the end points (sessions) of the communication path (see Chapter 4).

This process is described on Figure 5.11.

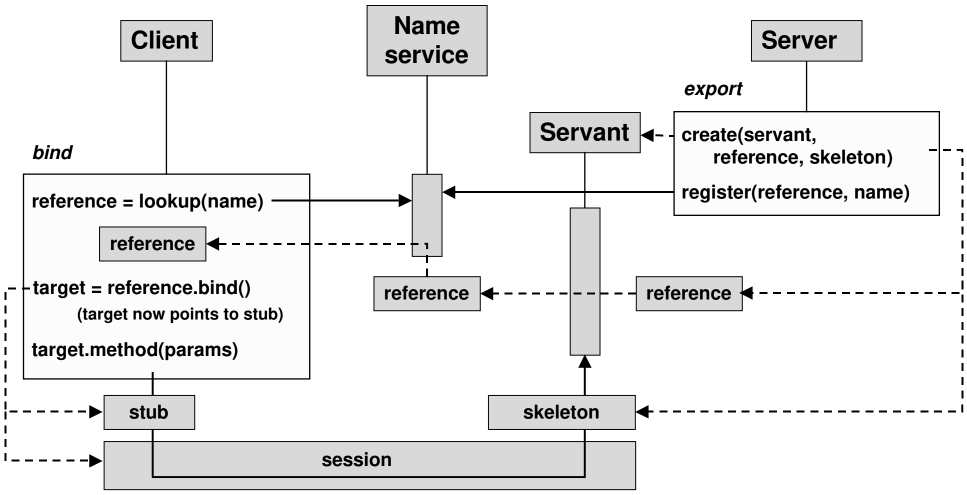

Figure 5.11. Binding for remote invocation

Specific instances of this process are described in more detail in the case studies.

5.3.3 Introduction to the Case Studies

We illustrate the internal working of an ORB with two case studies, based on open source implementations of Java RMI and CORBA, two widely used middleware systems based on the remote objects model.

We do not intend to give a detailed description of these systems, but to show the application of common design patterns. The two implementations are "personalities" built on top of the Jonathan kernel (3.4). Recall that Jonathan provides a framework for communication and binding. The core of both ORB implementations essentially consists of a binding factory, or binder (3.3.2), which itself relies on lower level tools and other binders:

- a set of tools for stub and skeleton generation;
- a binder for the specific remote invocation model;
- a binder for the communication protocol used between client and server;
- various auxiliary tools for managing elementary resources.

A global view of the structure of the ORBs is given on Figure 5.12. Some details such as the use of common resource managers (schedulers, storage allocators) are not shown. A more detailed view is provided for each case study.

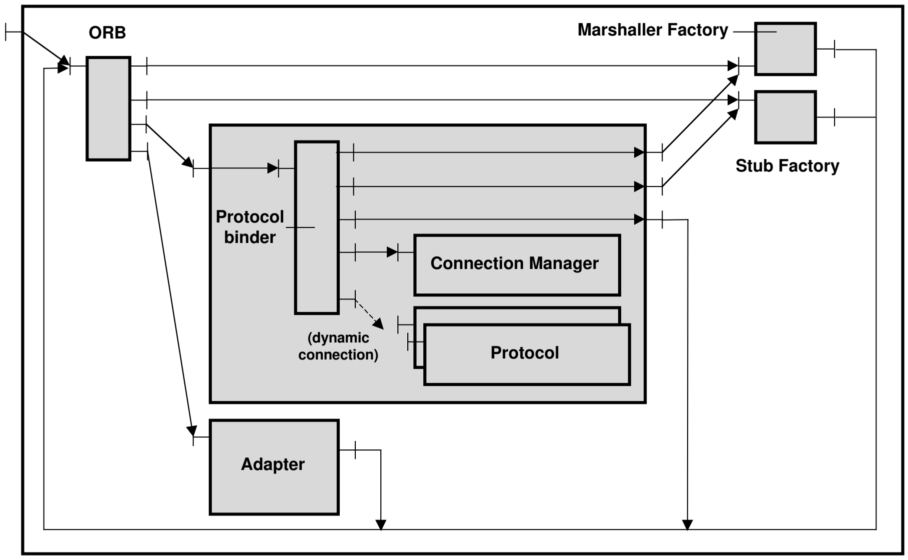

Figure 5.12. The common structure of Jonathan ORBs

The structure of the ORB is described in terms of "components" using a self-descriptive graphical formalism (the arrows connect "required" to "provided" interfaces – see Chapter 7 for more details).

5.4 Case Study 1: Jeremie, an Implementation of Java RMI

We present the Jeremie implementation of Java RMI in the Jonathan framework. After a brief introduction (5.4.1), we illustrate the principle of application development in Java RMI, through a simple example (5.4.2). We finally describe the inner working of the implementation (5.4.3).

5.4.1 Introducing Java RMI

Java Remote Method Invocation (RMI) [Wollrath et al. 1996] implements the remote objects model for Java objects. It extends the Java language with the ability to invoke a method on a remote object (a Java object located on a remote site), and to pass Java objects as parameters in method calls.

Since the Java language includes the notion of an interface, there is no need for a separate Interface Description Language (IDL). Stub and skeleton classes are generated from a remote interface description, using a stub generator (`rmic`).

Programming with remote objects is subject to a few rules of usage, as follows.

- A remote interface (the interface of a remote object) is defined like an ordinary Java interface, except that it must extend the `java.rmi.Remote` interface, which is nothing but a marker that identifies remote interfaces.
- A call to a method of a remote object must throw the predefined exception `java.rmi.RemoteException`.
- Any class implementing a remote object must create a stub and skeleton for each newly created instance; the stub is used as a reference for the object. This is usually done by making the class extend the predefined class `java.rmi.server.UnicastRemoteObject`, provided by the RMI implementation. Details are provided in Section 5.4.3.

Objects may be passed as parameters to methods. Local objects (residing on the caller's site) are passed by value, and must therefore be serializable. Non-local objects are passed by reference, i.e. a stub for the object is transmitted.

Thus, in accordance with the engineering principle discussed in 1.3.4 (see also [Waldo et al. 1997]), the Java RMI programming model does not attempt to be fully transparent, i.e. some modifications must be made to a centralized application when porting it to a distributed environment.

As mentioned in 5.1.3, a remote object system relies on a naming service. In Java RMI, this service is provided by a registry, which allows remote objects to be registered under symbolic names. The data that is registered is actually a reference for the remote object, i.e. a stub.

The registry may be located on any node. It is accessible on both the client and server node through a local interface called `Naming`. The interaction between `Naming` and the actual registry is described in 5.4.3.

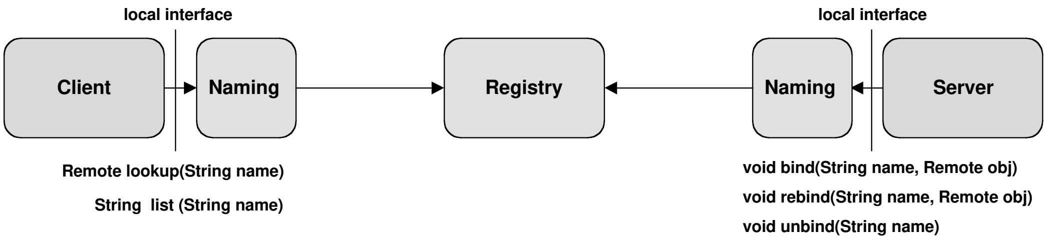

Figure 5.13. The RMI naming registry interface

The symbolic names have the form: `rmi://[host name][:portname]/local name` (the items between brackets are optional). A server registers references in the registry using `bind` and `rebind` and unregisters them using `unbind`. The client uses `lookup` to search the registry for a reference of a given name. The `list` method is used to list a registry's contents. The use of the registry is illustrated in the next section.

5.4.2 Developing an RMI Application

We present the main steps in developing an RMI application. Since emphasis here is on the internal working of the RMI system, we do not attempt to discuss a realistic application; we use the minimal "Hello World" example.

The centralized version of this example is presented below (the programs are self-explanatory).

```
// Hello Interface
     public interface Hello {
       String sayHello();}
     };
```

```
// Hello Implementation
     class HelloImpl implements Hello {
       HelloImpl() {   // constructor
         };
       public String sayHello() {
         return "Hello World!";
         };
     }
```

```
// Hello Usage
     ...
     Hello hello = new HelloImpl ();
     hello.sayHello();
```

In the distributed version, the client and the server run on possibly different machines. In order to allow this new mode of operation, two main problems must be solved: the client must find the location of the `hello` object (the target object of the invocation); the client must access the target object remotely.

Here is an outline of the distributed version (some details are omitted, e.g. catching exceptions, etc.). First the interface.

```
public interface Hello extends Remote {
   String sayHello() throws RemoteException;
}
```

The server program contains the implementation of the `Hello` interface and the main program. Objects are located through a naming service (the registry), which is part of the RMI environment. The server creates the target object and registers it under a symbolic name (5.4.1). The `rebind` operation does this, superseding any previous associations of the name. Note that the URL prefix of the name is `jrmi` (for the Jeremie version of RMI)

```
class HelloImpl extends UnicastRemoteObject implements Hello {
    HelloImpl() throws RemoteException {
    };
  public String sayHello() throws RemoteException {
      return "Hello World!";
  };
```

```
public class Server {
  public static void main (...) {
    ...
    Naming.rebind("jrmi://" + registryHost + "/helloobj", new HelloImpl());
    System.out.println("Hello Server ready !");
  }
}
```

The client program looks up the symbolic name, and retrieves a stub for the target object, which allows it to perform the remote invocation. The client and server must agree on the symbolic name (how this agreement is achieved is not examined here).

```
...
Hello obj = (Hello) Naming.lookup("jrmi://" + registryHost + "/helloobj");
System.out.println(obj.sayHello());
```

The registry may itself be a remote service (i.e. running on a machine different from that of the client and the server). Therefore, both the client and server use a local representative of the registry, called `Naming`, which locates and calls the actual registry, as described in 5.4.1. This allows the registry to be relocated without modifying the application.

The interaction diagram shown on Figure 5.14 gives a high-level view of the global interaction between the client, the server, and the registry. A more detailed view is presented in the following sections.

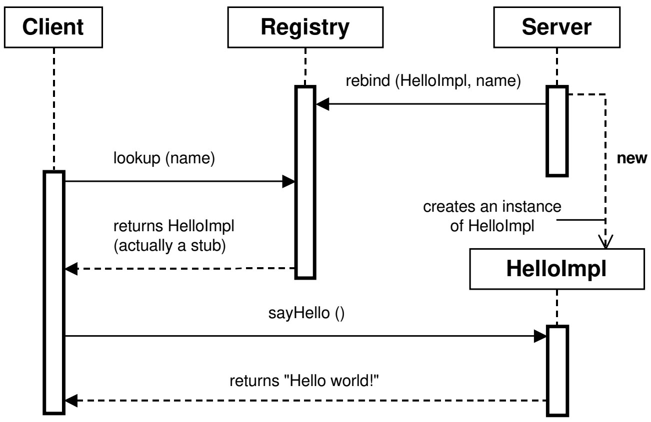

Figure 5.14. The “Hello World” application: overview

The actual execution involves the following steps.

1. Generate the stub and skeleton classes, by compiling the `Hello` interface definition on the server site, using the stub compiler provided by Jeremie.

2. Start the registry (by default the registry is located on the server site, but the system may be configured to make the registry reside on a different site).

3. Start the server.
4. Start the client.

5.4.3 The Inner Working of Jeremie

The execution of the above application proceeds as follows: on the server side, export the servant object and generate the actual stub and skeleton objects (the stub compiler has only generated the corresponding classes); on the client side, set up the binding, i.e. the actual connection between client and server, and perform the call. We examine the detail of these operations in turn.

Exporting the Servant Object

Recall that the servant object's class `HelloImpl` inherits from the standard class `UnicastRemoteObject`. The goal of this extension is essentially to enhance the creation of a remote object: when `new` is called on such a class, a skeleton and a stub are created in addition to the actual instance (using the classes generated by `rmic`), and the stub is returned as the result of `new`, to serve as a reference for the object. Thus the instruction

```
Naming.rebind("jrmi://" + registryHost + "/helloobj", new HelloImpl());
```

in the server program creates a new instance of `HelloImpl` (the servant), together with a skeleton and a stub for this servant, and returns the stub. It then registers the stub in the naming registry under a symbolic name.

Technically, the creation phase is performed by the `UnicastRemoteObject` (Figure 5.15), which exports the new servant object `impl` by calling a servant manager through an `exportObject` method. This manager (called `AdapterContext` on the figure) is essentially a driver for an object adapter; it first gets an instance of a stub factory from the ORB, and uses it to create a skeleton `skel`. It then registers `skel` into an adapter (essentially a table of objects), in which the skeleton is associated with a `key`. The `key` is then exported to the ORB, which delegates its operations to an `IIOPBinder`. This binder manages the IIOP protocol stack; it encapsulates the `key` into an object `ref`, of type `SrvIdentifier`, which includes a [`host`, `port`] pair (the port number was passed as a parameter to `exportObject`; if missing, an available port is selected by the binder). Thus `ref` is actually a reference for the servant `impl`. This reference is then used to create a stub (again using the stub factory), which is finally returned.

The stub is then registered, which concludes the export phase. Note that the stub must be serializable in order to be transmitted over the network.

This framework is extensible, i.e. it identifies generic interfaces under which various implementations may be plugged in, such as the adapter interface and the binding protocol interface, which are shown on the figure.

Setting up the Binding

The client looks up the servant in the registry, using its symbolic name:

```
Hello obj = (Hello) Naming.lookup("jrmi://" + registryHost + "/helloobj");
```

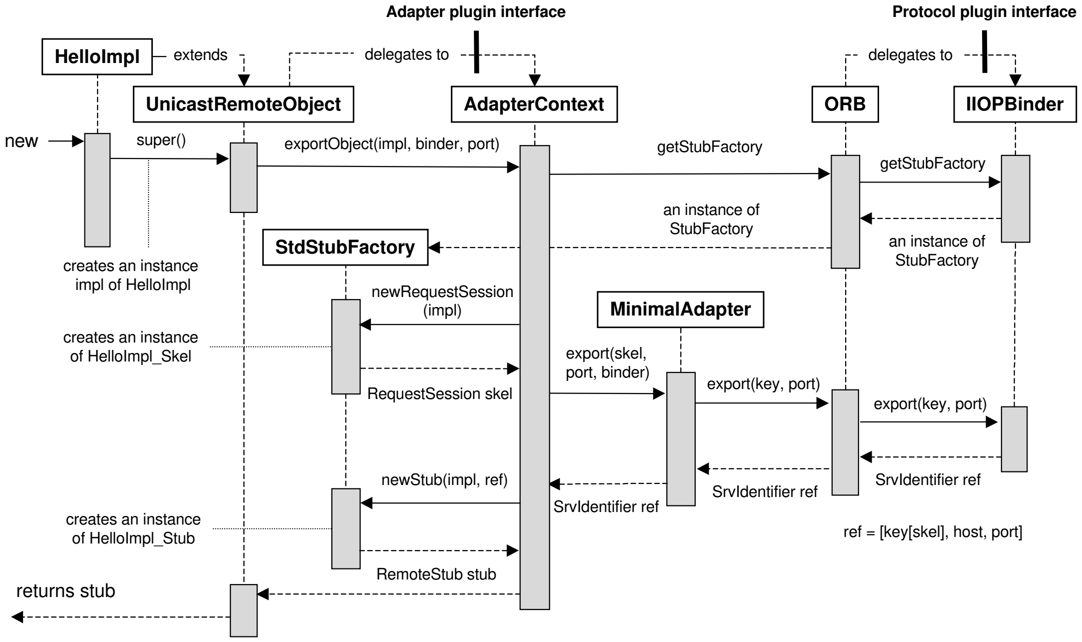

Figure 5.15. Remote Method Invocation: creating the stub and skeleton

This operation apparently retrieves a "servant", but what it actually does is setting up the binding, by first retrieving a stub and then binding this stub to the skeleton, and therefore to the actual servant. We now present the details of this operation (Figure 5.16).

The stub object stored in the registry is returned as a result of the `lookup` operation. This stub is still unbound, i.e. the calling path to the remote servant is not set up. The binding operation relies on the mechanisms of object transmission in Java:

- when an externalizable object is read from an `ObjectInputStream` by the `readObject` method, this method is superseded by a specific `ReadExternal` method;
- after an object has been de-serialized, a `readResolve` method is called on this object.

The first mechanism is used when the stub delegate (here called `RefImpl`) is read. Recall that the delegate contains a reference to the remote servant, including its IP address, port number and key in the adapter. The call to `ReadExternal` creates a client endpoint (called `CltId` on the figure), managed by the `IIOPBinder`. The second mechanism invokes the `bind()` method on this endpoint, which in turn sets up the binding, i.e. the path to the remote object, using the `IIOPBinder` (to set up a TCP/IP session), and a stub factory (to put the final stub together).

Performing the Call

Finally, the call is performed by the operation

```
System.out.println(obj.sayHello());
```

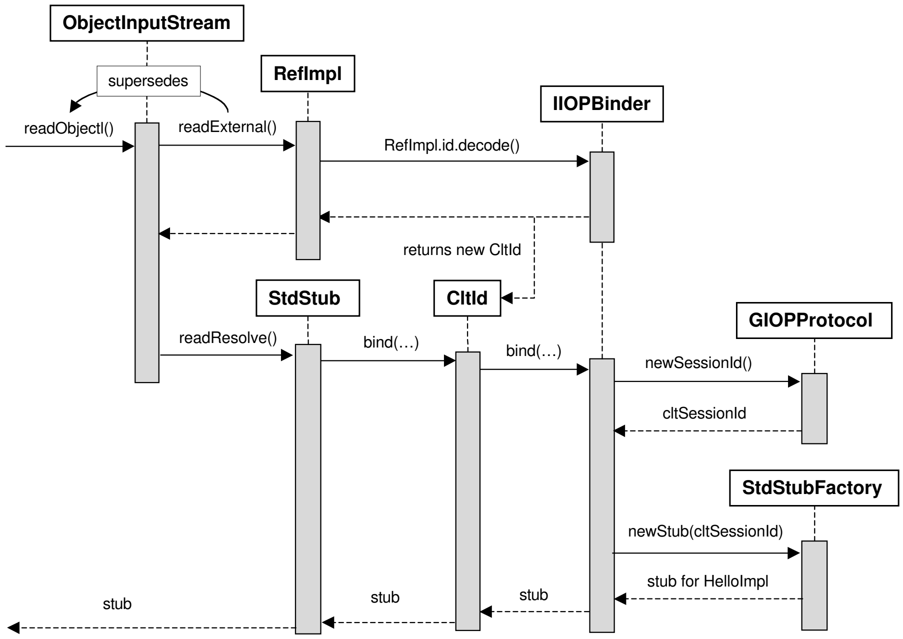

Figure 5.16. Remote Method Invocation: retrieving the stub on the client side

which actually invokes the `sayHello()` method on the stub. The skeleton is located using the reference contained in the delegate, the remote call is performed using the IIOP protocol, and the skeleton dispatches the invocation to the `sayHello` method of the servant. The return value is transmitted using the reverse path.

The Mechanics of Registry Invocation

Recall that the registry may be located on any node and is accessible through a local `Naming` interface. In order to reach the actual registry, `Naming` calls a `LocateRegistry`, giving the symbolic name under which the registry servant has been initialized (e.g. `RegistryImpl`).

`LocateRegistry` uses a binder (here `JIOP`) to retrieve a stub for `RegistryImpl` (Figure 5.17). The binding process is identical to that described above (i.e. using `IIOPBinder`), and details are not shown.

When the registry server is initialized on a host, it creates a `RegistryImpl` servant on a specified port. A precompiled stub class for this servant (`RegistryImpl_Stub`, needs to be present on each site using the registry.

5.5 Case Study 2: David, an Implementation of CORBA

We present the David implementation of CORBA in the Jonathan framework. After a brief introduction (5.5.1), we illustrate the principle of application development in CORBA,

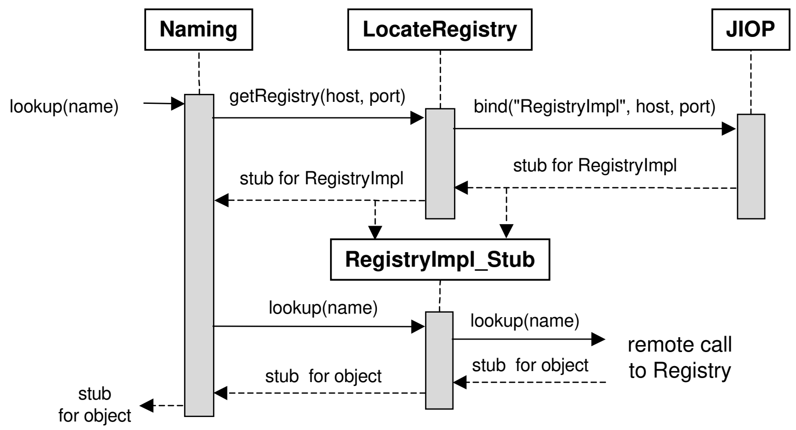

Figure 5.17. Invoking a registry method

through a simple example (5.5.2). We finally describe the inner working of the implementation (5.5.3).

5.5.1 Introducing CORBA

The Object Management Group (OMG) is a consortium created in 1989 with the goal of making and promoting standards in the area of distributed applications development. CORBA (Common Object Request Broker Architecture) is one of these standards, covering the area of middleware and services for applications based on distributed objects.

CORBA is intended to allow the cooperation of heterogeneous applications, using different languages and operating environments. To that end, the standard defines a common way of organizing applications based on the remote objects model, a common Interface Definition Language (IDL), and the main components of an object request broker architecture. In addition, a number of common services are specified for such functions as naming, trading, transactions, persistence, security, etc. These services are accessible through IDL interfaces.

The global organization of an ORB, as defined by CORBA, is represented on Figure 5.18.

The usual invocation path from a client application to a method provided by a remote servant is through a stub and a skeleton generated from an interface description. CORBA defines a generic IDL, from which stubs and skeletons may be created for different languages, using the appropriate tools. The mapping of the entities defined in the IDL to constructs in a particular language is defined by a set of binding rules. Such bindings have been defined for all usual programming languages. Thus for instance a client written in C++ may call a servant written in Java: the client stub is compiled using tools based on the IDL to C++ binding, while the skeleton is compiled using tools based on the IDL to Java binding.

The ORB itself provides interfaces to both clients and servants, in order to fulfill a number of basic functions such as a primitive name service and a tool to map object references to strings and vice-versa, in order to facilitate the interchange of references.

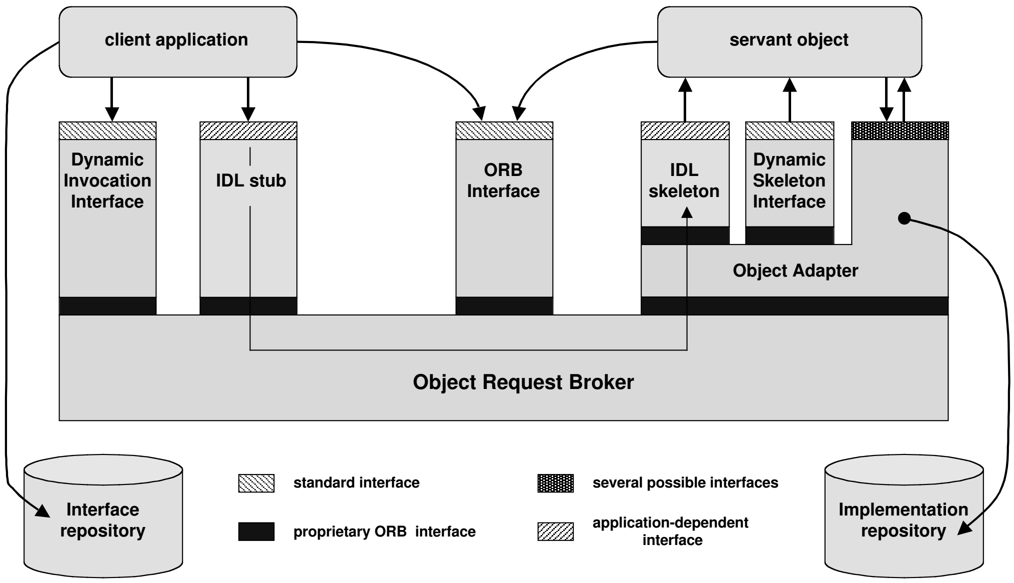

Figure 5.18. The global organization of CORBA

CORBA uses object adapters on the server side. The motivation for an adapter has been given in 5.2.3. The adapter manages servant objects, using an implementation repository. The use of adapters in CORBA is further developed in 5.6.

Finally, CORBA provides facilities for dynamically constructing the elements of an invocation path, by allowing a client application to discover a servant interface at run time (using an interface repository), and to create requests without using stub generation tools. These aspects are examined in 5.6.

5.5.2 Developing a CORBA Application

We use the same application (HelloWorld) as in Java RMI. The centralized version has been described in Section 5.4.2.

The IDL description of the interface of the remote object is:

```
interface Hello {
   string sayHello();
};
```

Since we intend to write the client and server programs in Java, this description must be compiled using an IDL to Java compiler, to generate the stub and skeleton files (`_HelloStub.java` and `_HelloImplBase.java`, respectively). The compiler also generates auxiliary programs (holders and helpers), whose function will be explained when necessary.

Here is the the program of the server.

```
import org.omg.CORBA.ORB;
```

```
import org.omg.CosNaming.NamingContext;
import org.omg.CosNaming.NamingContextHelper;
import org.omg.CosNaming.NameComponent;
import org.objectweb.david.libs.helpers.IORHelpers;
import idl.*;

class HelloImpl extends _HelloImplBase {

   HelloImpl() {}
   public String sayHello() {
      return "Hello World!";
   }
}

public class Server {
   public static void main (String[] args) {
      try {
         ORB orb = ORB.init(args,null);
         Hello hello = new HelloImpl();
         orb.connect(hello);
         IORHelpers.writeIORToFile(orb.object_to_string(hello),"hello_ior");
         org.omg.CORBA.Object ns_ref =
            orb.resolve_initial_references("NameService");
         NamingContext ns = NamingContextHelper.narrow(ns_ref);
         ns.rebind(new NameComponent[] { new NameComponent("helloobj","") },hello);
         System.out.println("Hello Server ready");
         orb.run();
      } catch (Exception e) {
         System.err.println("Hello Server exception");
         e.printStackTrace();
      }
   }
}
```

As may be expected, the program of the servant object is identical to that used in Java RMI. The difference lies in the server program.

The first instruction `ORB orb = ORB.init(args,null)` initializes the ORB, which is thereafter accessible as an object `orb`. Then an instance `hello` of the servant is created. The instruction `orb.connect(hello)` makes the servant known to the ORB for further use, by exporting it to an adapter.

The following instructions register the servant (actually a reference to it) in the CORBA name service. A reference to this service must first be retrieved, which is done using a primitive name service (`resolve_initial_references`) provided by the ORB. Note the use of the `narrow` operation performed by a "helper" program: the primitive name service returns references to type `Object`, and these references need to be cast to the actual type of the object. However, a language cast operation cannot be used here, because the references are not language references, but remote references managed by the ORB.

Finally, the `orb.run()` operations puts the server thread to sleep, waiting for client invocations.

Here is the client program.

```
import org.omg.CORBA.ORB;
import org.omg.CosNaming.NamingContext;
import org.omg.CosNaming.NamingContextHelper;
import org.omg.CosNaming.NameComponent;
import idl.*;

public class Client {
   public static void main(String[] args) {
      try {
         ORB orb = ORB.init(args,null);
         org.omg.CORBA.Object ns_ref =
            orb.resolve_initial_references("NameService");
         NamingContext ns = NamingContextHelper.narrow(ns_ref);
         org.omg.CORBA.Object obj_ref =
            ns.resolve(new NameComponent[] { new NameComponent("helloobj","") });
         Hello obj = HelloHelper.narrow(obj_ref);
         System.out.println(obj.sayHello());
      } catch (Exception e) {
         System.err.println("Hello Client exception");
         e.printStackTrace();
      }
   }
}
```

The ORB is initialized like in the server program. Then a reference for the servant is retrieved using the naming service. This reference must again be cast to its actual type, using the appropriate helper. Finally the remote operation is invoked.

CORBA uses a standard object reference format (Interoperable Object Reference, or IOR), defined by the IIOP specification. While these references are opaque (i.e. their contents is hidden), they may be used through primitives provided by the ORB. Thus, in the server program, the registration of the servant in the name service might be replaced by:

```
IORHelpers.writeIORToFile(orb.object_to_string(hello),"hello_ior_srv");
```

This instruction writes in the `hello_ior_srv` file a "stringified" form of the IOR. This string may be sent to the client (e.g. by mail), and copied in a file `hello_ior_clt`. The reference to the servant may then be retrieved in the client program by:

```
hello = orb.string_to_object(IORHelpers.readIORFromFile("hello_ior_clt"));
```

Executing the application involves exactly the same steps as in Java RMI: generating the stub and skeleton classes by compiling the IDL interface description; starting the name service; starting the server; starting the client.

CORBA allows additional facilities: dynamic request generation, adapter programming. These aspects are examined in Section 5.6.

5.5.3 The Inner Working of David

The compilation of the IDL interface description generates several files for each servant. For instance, from the `Hello` interface description, the `Idl2Java` compiler creates the Java source files for the following entities: `HelloHelper`, a class which carries auxiliary operations such as `narrow` (the reference casting operation); `HelloHolder`, a class which carries read and write operations of objects of type `Hello` on streams; `HelloOperations`, the interface of objects of type `Hello`; `_HelloStub`, the stub class; `_HelloImplBase`, the base class from which the servant object derives (this class extends a predefined `Skeleton` class).

The overall working of the CORBA ORB is similar to that described for Java RMI. The main differences are the use of the adapter on the server side, and the reflective operations described in 5.6.

The first step is to obtain a reference on the ORB as an object. This is done by the operation `ORB orb = ORB.init()`, which creates an ORB as an instance of a singleton class (a class with a single instance). This ORB provides a number of primitive operations, some of which are described in the rest of this section.

Exporting the Servant Object

The servant object's class `HelloImpl` inherits from the `_HelloImplBase` generated class, which itself inherits from `Skeleton`. This extends the servant's class constructor: when a new servant is created (an instance `hello` of `HelloImpl`), a new server delegate (5.3.1) is created and associated with that servant.

The operation `orb.connect(hello)` exports the new servant to the ORB. The server delegate is registered in the adapter (which returns a key to subsequently retrieve the servant), and then exported to the IIOP binder (which provides host address and port). At this stage, the IOR (containing host, port, and key) is ready, and it is actually registered in the servant itself (here, as an instance of `SrvIdentifier`). This is shown on Figure 5.19.

The next step consists in registering the servant in the name server. The name server must first be located, which is done as follows through a primitive naming service provided by the ORB:

```
org.omg.CORBA.Object ns_ref =
    orb.resolve_initial_references("NameService");
NamingContext ns = NamingContextHelper.narrow(ns_ref);
```

Retrieving a reference for the name server uses an internal association table (a context) managed by the ORB, in which a reference for the server has been initially registered. From this reference, a stub for the name server is generated, using the binding mechanism explained in the next subsection. The function of `narrow` (reference cast) has been explained in 5.5.2.

The servant is then registered in the name server:

```
ns.rebind(new NameComponent[] { new NameComponent("helloobj","") },hello);
```

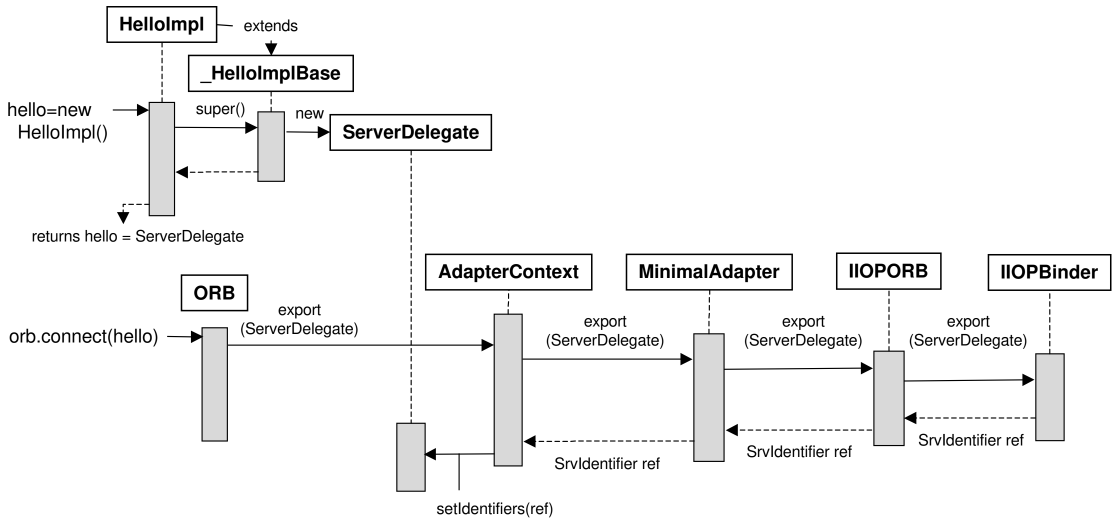

Figure 5.19. Exporting a servant in CORBA, phase 1

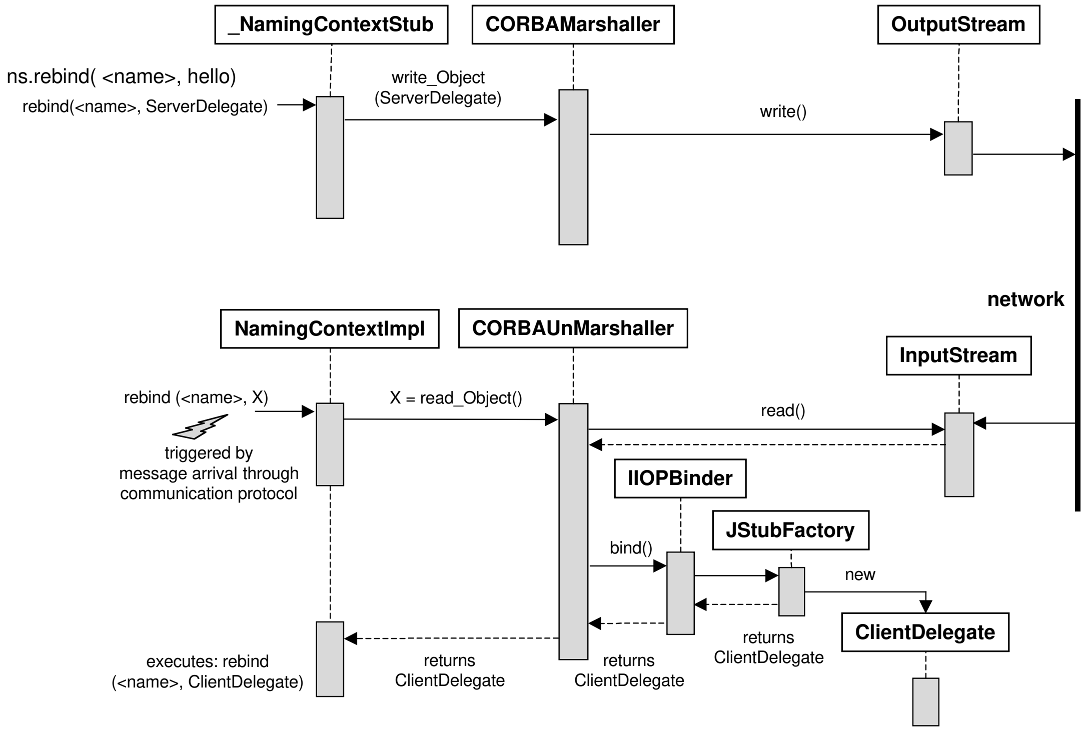

Figure 5.20. Exporting a servant in CORBA, phase 2

The effect of this operation is the following (Figure 5.20).

The reference of the servant (i.e. the `SrvIdentifier`) is sent to the name server, using the encapsulation mechanism described in 3.4.3. In this process, the identifier is unmarshalled from an `ObjectInputStream`, using the `ReadObject` operation. This operation invokes `bind` on the decoded identifier, in the `IIOPBinder` context, which in turn invokes a `JStubFactory`. This latter constructs a `ClientDelegate` (5.3.1), an access point to the servant using a generic (invoke) interface. This delegate is registered in the name server, to be later retrieved by the client, as described below.

Completing the Binding

After initializing the ORB and locating the name server (like above), the client binds to the servant by calling the name server:

```
ns.resolve(new NameComponent[] { new NameComponent("helloobj","") });
Hello obj = HelloHelper.narrow(obj_ref);
```

Like in the registration phase, the `ClientDelegate` is transmitted in an encapsulated form and unmarshalled through `ReadObject`. The `IIOPBinder` now constructs a stub using this delegate, using the mechanism illustrated in the lower-right quarter of Figure 5.20. The binding is now complete, including the communication path (session) between the client and the server.

Performing the Call

The call may now proceed using the object invocation scheme described in 5.3.1 and illustrated in Figures 5.9 and 5.10. The mechanics of passing objects as parameters or results by reference is again illustrated by the operation of the `Naming.rebind` primitive (Figure 5.20, i.e. recreating a delegate on the receiving site as the core of the object's stub.

5.6 Complements and Extensions

Currently not available (should cover reflective features, semi-synchronous invocations, etc.)

5.7 Historical Note

The historical evolution of object middleware has been outlined in the Historical Note section of the Introduction Chapter (1.5).

References

[OMG 2003] OMG (2003). CORBA/IIOP Specification. Object Management Group. http://www.omg.org/technology/documents/formal/corba_iiop.htm.

[Spiegel 1998] Spiegel, A. (1998). Objects by value: Evaluating the trade-off. In *Proceedings Int. Conf. on Parallel and Distributed Computing and Networks (PDCN)*, pages 542– 548, Brisbane, Australia. IASTED, ACTA Press.

[van Steen et al. 1999] van Steen, M., Homburg, P., and Tanenbaum, A. (1999). Globe: A Wide-Area Distributed System. *IEEE Concurrency*, pages 70–78.

[Völter et al. 2004] Völter, M., Kircher, M., and Zdun, U. (2004). *Remoting Patterns: Foundations of Enterprise, Internet, and Realtime Distributed Object Middleware*. John Wiley & Sons.

[Waldo et al. 1997] Waldo, J., Wyant, G., Wollrath, A., and Kendall, S. (1997). A Note on Distributed Computing. In Vitek, J. and Tschudin, C., editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 49–64. Springer-Verlag.

[Wollrath et al. 1996] Wollrath, A., Riggs, R., and Waldo, J. (1996). A Distributed Object Model for the Java System. *Computing Systems*, 9(4):265–290.