7

# Advanced Communication between Distributed Objects

## Learning Objectives

We will acquaint ourselves with the primitives for distributed object communication. We have already understood synchronous communications between a client and a server object, which is the default way for object-oriented middleware to execute an object request. We will grasp the influence of non-functional requirements for distributed object systems, such as reliability or high performance, and understand why these requirements demand different forms of communication between distributed objects. We will ascertain the principles of non-standard synchronization of object requests, such as deferred synchronous, oneway or asynchronous communication between objects. We will also study request multiplicity and learn about object requests that involve more than one operation execution or more than one server object. We will then understand the different degrees of reliability with which distributed object requests can be executed and acquaint ourselves with the trade-offs between reliability and performance.

## Chapter Outline

Java/RMI, COM and CORBA (see Chapter 4) all enable a client object to request an operation execution from a server object. Requests in all these object-oriented middleware systems have the same properties: they are synchronous and only involve the client and the server objects; they are defined at the time a stub or proxy invocation is compiled into the client; and they may fail, in which case, the client is informed of the failure.

An object may have to meet requirements that demand non-standard requests from its server components. Such non-standard requests may have more advanced properties than the requests we have seen so far. One example is an object that has to broadcast information to more than one object.

In general certain non-functional requirements, such as performance, genericity or reliability, may demand the use of non-standard requests. It is, therefore, important that the non-functional requirements of a system are specified during the requirements analysis stage of a project. They can then be used to decide on the properties of requests used for the interaction between objects. During the course of discussing different properties of requests, we will indicate the influencing non-functional requirements that demand their use. Such advanced requests might be more expensive, either because they are more complicated to program or because they consume more resources at run-time. It is therefore the responsibility of the engineer to carefully consider the trade-off between using the right types of request in order to meet the requirements and producing objects within budget.

This chapter reviews requests from three perspectives. These are the *synchronization* of requests, the *multiplicity* of objects involved in requests and the degree of *reliability* with which requests are transferred and processed. These dimensions are orthogonal to each other. Hence, we can discuss the extent to which object-oriented middleware supports them independently. For all of these considerations, we will discuss the underlying principles and then show how they can be implemented. These implementations can be based on the standard object requests that we discussed above and can thus be achieved in Java/RMI, COM and CORBA alike. Sometimes, however, a more efficient implementation is provided by object-oriented middleware standards or products. OMG/CORBA is the most powerful standard in this respect. Thus in addition to discussing implementations with standard object requests, we show how CORBA supports advanced communication in a more efficient way.

# 7.1 Request Synchronization

## 7.1.1 Principles

Synchronous object requests block the client until the server has finished the execution.

The standard object requests that CORBA, COM and Java/RMI support, and which we discussed in Chapter 4 are all *synchronous*. This means that the client object is blocked while the server object executes the requested operation. Control is only returned to the client after the server has completed executing the operation or the middleware has notified the client about the occurrence of an error.

Non-synchronous requests are needed to avoid clients being blocked.

Although this synchronous request execution is appropriate in many cases, high performance requirements motivate different forms of synchronization. Depending on the time it takes to execute operations, it may not be appropriate for the client to be blocked. Consider a user interface component that requests operation executions from server objects. If these operations take longer than the response-time requirements permit, synchronous requests cannot be used. Moreover if several independent operations are to be executed from different servers it might be appropriate to execute them simultaneously in order to take advantage of distribution. With synchronous requests, single-threaded clients can, however, only request one operation execution at a time. There are different ways that synchronicity can be relaxed. An overview of these approaches is given in Figure 7.1 using UML sequence diagrams.

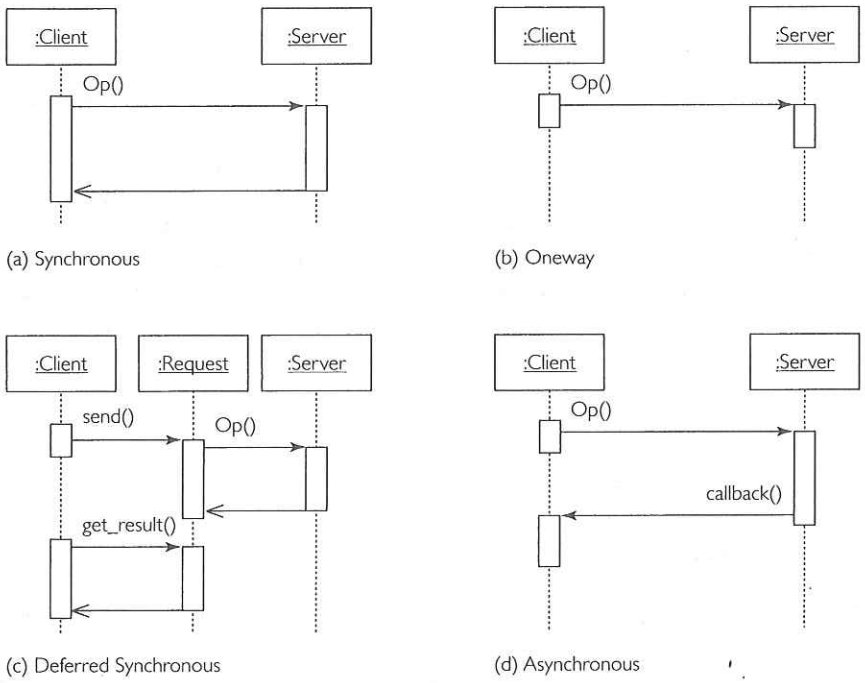


**Figure 7.1**
Request Synchronizations

Oneway requests return control to the client immediately and the client and the server do not synchronize.

*Oneway* requests return control to the client as soon as the middleware has accepted the request. The requested operation and the client are then executed concurrently and they are not synchronized. Oneway operation requests can be used if there is no need for the client to await the completion of the operation. This implies that the semantics of the client must not depend on the result of the requested operation. This is the case if the operation does not produce a result that is used by the client and if the operation cannot violate the server's integrity and therefore does not need to raise exceptions.

Deferred synchronous requests give control back to the client and the client polls for results later.

Similar to oneway requests, *deferred synchronous* requests return control to the client as soon as the distribution middleware has accepted the request. Unlike oneway operations, deferred synchronous requests can also be used in situations when a result needs to be transferred to the client. Deferred synchronous requests are dynamic and explicit request objects are used to represent ongoing requests. The client object is not blocked between requests and it can

perform other computations. In particular, it can use other request representations to issue several requests concurrently. In that way a client can request more than one concurrent operation without using multiple threads of control. In order to obtain the result, clients invoke an operation from the request representation to get the result. A slight disadvantage of deferred synchronous requests is that clients are in charge of synchronizing with the server when they obtain the result. In particular, the result may not yet be available at the time the client invokes the operation to get the result. Then the client is either blocked or the operation needs to be invoked again at a later time. This technique is also known as *polling*.

Asynchronous requests give control back to the client immediately and the server calls the client to provide the result.

The need for polling is avoided with *asynchronous* requests. When a client uses an asynchronous request it regains control as soon as the middleware has accepted the request. The operation is executed by the server and when it has finished it explicitly calls an operation of the client to transfer the operation result. This operation is referred to as a *callback*. The callback is determined by the client when it makes the asynchronous request.

Non-synchronous requests can be achieved by synchronous requests and multi-threading.

Synchronous requests are supported by all object-oriented middleware. In the remainder of this section, we review how the other three non-synchronous forms can be implemented. We will see that they can all be implemented using synchronous requests and multi-threading. These considerations are independent of the particular middleware as CORBA, COM and Java/RMI can all be used in a multi-threaded way. Moreover, CORBA provides direct support for oneway, deferred synchronous and asynchronous requests. It is probably reasonable to assume that these implementations are more easy to use and execute more efficiently than the multi-threading implementation that is built by an application designer on top of other middleware.

## 7.1.2 Oneway Requests

### Using Threads

Threads are used to implement concurrency, usually within one operating system process. It is much faster to start a thread than to launch an operating system process. Thus, threads are more lightweight than processes. There are a number of basic operations for threads. Spawning a new thread creates a child thread that is executed concurrently with its parent thread. Terminating a thread terminates the execution and deletes it. A thread may be suspended, thereby interrupting its execution. Suspended threads may resume execution and remain in the states in which they were suspended. If a thread is joined with a second thread, the first thread waits for the second thread to terminate and then continues execution. Threads are made available to programmers either by means of programming language constructs, such as class `Thread` in Java or co-routines in Ada, or by means of libraries, as in C++ or C.

Blocking the main thread can be avoided by making the synchronous call in a new thread

The designer of a client object may use threads in order to implement a oneway request. The general idea is that the client object spawns a child thread and executes a synchronous request using the child. Hence, the child is blocked but the client continues processing within the parent thread. Because oneway requests never synchronize the client and server

objects, there is no need to synchronize the two threads; we can simply let the child thread die as soon as the request is finished.

Example 7.1 shows the implementation of oneway operations using Java/RMI. We have chosen Java because of the nice integration of multi-threading into the Java programming language. However, we note that the same strategy can be used for implementing CORBA and COM oneway requests. Depending on the programming language that is used in these systems, it may be slightly more complicated to use threads, but it is still feasible. However, the use of threads implies quite a significant performance penalty and we now discuss the implementation of oneway requests in CORBA, which provides more efficient primitives.

Threads can be used with COM, CORBA and Java/RMI to avoid blocking clients.

## Example 7.1

Using Java Threads for Oneway Requests

```
class PrintSquad {
  static void main(String[] args) {
    Team team;
    Date date;
    // initializations of team and date omitted ...
    OnewayReqPrintSquad a=new OnewayReqPrintSquad(team,date);
    a.start();
    // continue to do work while request thread is blocked...
  }
}

// thread that invokes remote method
class OnewayReqPrintSquad extends Thread {
  Team team;
  Date date;
  OnewayReqPrintSquad(Team t, Date d) {
    team=t; date=d;
  }
  public void run() {
    team.print(date); // call remote method
  }
}
```

Threads are used to make a oneway request to a remote Java object `team` that prints a formatted representation for the squad to be used at a particular `date`. Class `PrintSquad` is the client object. `OnewayReqPrintSquad` extends the Java `Thread` class in order to implement the oneway request to print the squad data. Creation of a new `OnewayReqPrintSquad` object effectively spawns a new thread. The parameters passed to the constructor identify the remote server object `team` and the parameter for the remote method invocation `date`. A newly-created thread is started using the `start` method. This invokes the `run` method and the thread dies when `run` terminates. The remote method invocation of `print` is actually made as soon as the thread `a` has been started, i.e. the `run` method is executed. Then the thread `a` is blocked until the remote method has terminated.