Oneway Requests in CORBA

As shown in Figure 4.2 on Page 97 there are two ways in which clients can make requests: by invoking a client stub or by using the dynamic invocation interface (DII). Both of them support oneway requests.

CORBA oneway operations implement oneway requests more efficiently than threads.

As we have seen in Chapter 3, the client stubs are generated by the IDL compiler from an IDL interface definition. CORBA/IDL has a language construct for the definition of *oneway operations*. An operation exported by an object type can be defined as oneway, provided that the operation has a `void` return type, does not have any `out` or `inout` parameters and does not raise any type-specific exceptions. These preconditions for oneway requests imposed by CORBA are reasonable as they imply that the semantics of the client execution does not depend on the oneway operation. When an operation is declared as oneway, clients requesting that operation will not await completion of the operation. They will continue with the instruction after the request as soon as the request has been accepted by the object request broker.

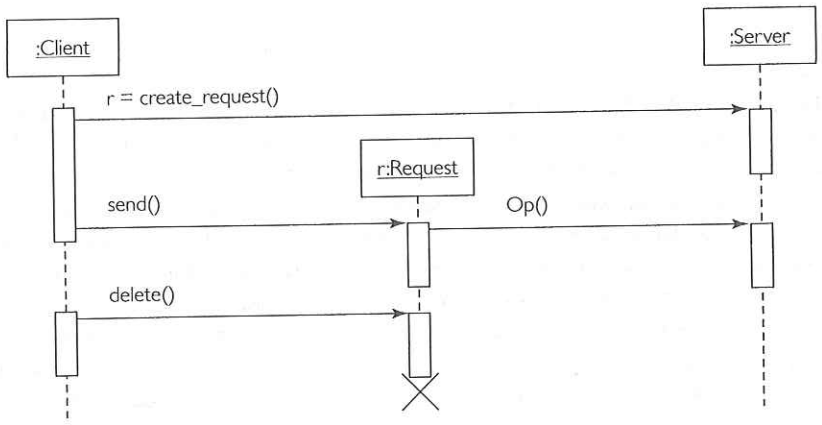
Example 7.2
Oneway Operations in CORBA

```
interface Team {
  oneway void mail_training_timetable(in Date date);
  ...
}
```

The trainer desires an automated facility to mail training timetables to all players. This can be implemented quite easily as an operation of the object type `Team`. Instances of `Team` have references to all `Player` objects from which the player's addresses can be obtained. To format letters and have them printed, however, takes a while and it would be inappropriate to block the trainer's management application until the printing is finished. Hence, we define the operation as a oneway operation.

CORBA oneway operations are implemented by the client stub that is derived from the IDL oneway operation declaration. This means that it is not the designer of a client who chooses this form of synchronization but the designer of the server. Moreover, the synchronization is determined when the interface definition is compiled, which is when the stub is generated.

Figure 7.2
Using the DII to Implement Oneway Requests



In the above example, for instance the client designer could not request synchronous execution of the operation to mail the training timetable. This restriction does not hold if oneway requests are issued using the dynamic invocation interface.

The CORBA Dynamic Invocation Interface defines `Request`, an object type that is used as an explicit representation of requests. The `send` operation exported by `Request` has a parameter that controls whether or not the client wants to wait for the result to become available. With a no-wait parameter, `send` is, in fact, a oneway request because the client regains control while the server executes the requested operation, as shown in Figure 7.2. The client can then delete the request object.

Oneway requests can also be implemented with CORBA's DII.

Comparison

In a comparison of the three approaches, oneway operations implemented with threads can be achieved in any middleware system. The implementation requires a thread to be started and deleted, which may involve a significant performance penalty. To avoid this and also to make oneway operations available in programming languages that do not have multi-threading support, CORBA provides direct support for oneway requests. The CORBA/IDL `oneway` operations are less flexible than oneway requests with the DII because a synchronous execution cannot be requested for them. On the other hand, an implementation with the dynamic invocation interface is less safe, because it is not validated whether the requested operation produces a result or raises exceptions to report integrity violations. Moreover oneway requests using the DII are certainly less efficient because `Request` objects need to be created at runtime. Hence, when deciding whether to use a client stub or the DII, the designer has to consider the trade-off between flexibility and lack of safety and performance.

7.1.3 Deferred Synchronous Requests

Using Threads

Oneway requests cannot be used if the client needs to obtain a result. In this case, the client and the server have to be synchronized. With deferred synchronous requests, the client initiates the synchronization. In order to achieve deferred synchronous requests with threads, we have to modify our multi-threaded oneway implementation in such a way that the client waits for the request thread to terminate before it obtains the result. We show an application in Example 7.3.

We note again, that the primitives that we use to achieve deferred synchronous invocation are multi-threading and synchronous requests and that this approach can therefore be used in COM and Java/RMI as well as CORBA. We now discuss a mechanism of CORBA that achieves more efficient deferred synchronous requests due to the lack of threading overhead.

Threads and synchronous requests can be used to implement deferred synchronous requests in CORBA, COM and Java/RMI.

Deferred Synchronous Requests in CORBA

Unlike CORBA oneway operations, deferred synchronous requests cannot be issued using client stubs. They are only available through the Dynamic Invocation Interface. The reason for this restriction is the fact that client stubs are effectively methods or procedures in the

Deferred synchronous requests are available through the DII.

Example 7.3
Using Java Threads for Deferred Synchronous Requests

```
class PrintSquad {
  public void print(Team team, Date date) {
    DefSyncReqPrintSquad a=new DefSyncReqPrintSquad(team,date);
    // do something else here.
    a.join(this); // wait for request thread to die.
    System.out.println(a.getResult());//get result and print it
  }
}
// thread that invokes remote method
class DefSyncReqPrintSquad extends Thread {
  String s;
  Team team;
  Date date;
  DefSyncReqPrintSquad(Team t, Date d) {team=t; date=d;}
  public String getResult() {return s;}
  public void run() {
    String s;
    s=team.asString(date);// call remote method and die
  }
}
```

This example extends the oneway request implementation by re-synchronizing the two threads. Rather than printing on a remote host, we want a formatted result of the training schedule as a string, achieved by `asString`. This string is the result that requires the two threads to be synchronized. The thread that executes the request then stores the result so that it can be retrieved. Before the client thread retrieves the result, it uses the `join` operation to wait for the request thread to die.

programming language of the client and are not suitable for representing multiple threads of control. The approach to implementing deferred synchronous requests is the same as with oneway requests. The `Request` object represents a thread of control that is ongoing on some host of a server object. The difference between oneway requests and deferred synchronous requests is that the client invokes a `get_response` operation from the `Request` object some time after having sent the request. This synchronizes the client and server objects and returns control to the client as soon as the result is available. Then the operation result is available in a data structure that was passed during the `create_request` operation. Once the operation result has been obtained the request object is deleted. Figure 7.3 shows how the dynamic invocation interface is used for deferred synchronous requests.

The `get_response` operation supports two ways of synchronizing client and server. This is necessary to give clients a chance not to be blocked if the operation result is not yet available. A parameter of `get_response` controls whether or not the client will wait for the result to become available. If set to 'no wait', `get_response` returns immediately indicating whether or not the result is available. If it is not available, the client can continue with some other processing and poll later by calling `get_response` again. The need for polling is removed with asynchronous requests, which we discuss next.

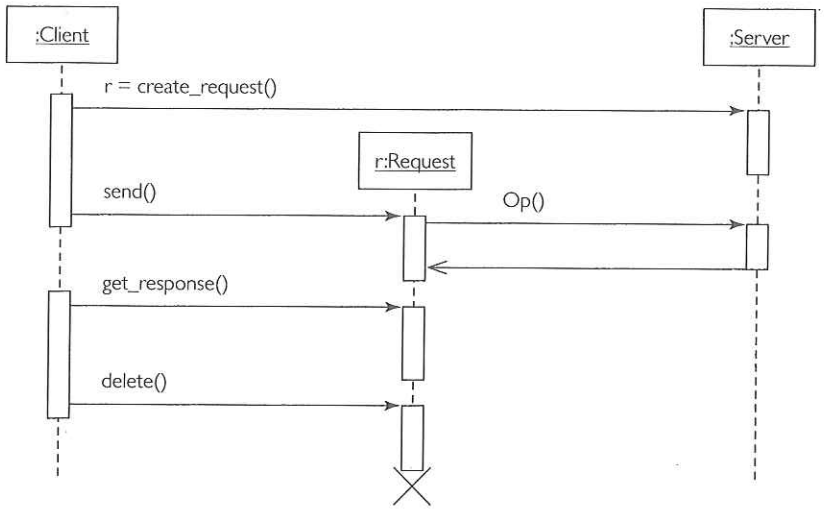


Figure 7.3
Using the DII to Implement Deferred Synchronous Requests

7.1.4 Asynchronous Requests

Using Threads

The general idea of implementing asynchronous requests with threads and synchronous requests is very much the same as with the deferred synchronous request implementation. The difference is that it is the request thread that initiates the synchronization rather than the main thread. As soon as the request is completed, the new thread invokes a callback operation in order to pass on the result and then joins the main thread as in Example 7.4.

Note that Example 7.4 does not synchronize the two threads. The child thread dies after the callback has been completed. This is appropriate in this example as the callback prints some information on the console and does not modify the state of the client. If at some stage the client needs to access the result of the request or review whether the request has been completed, synchronization would be necessary. This can be done using the thread synchronization primitives that are available in Java. The client, for instance, could suspend its execution and then the callback could resume it, or the parent thread simply suspends its execution. Other options include the `join` operation, which makes the parent thread wait for the child thread to die, or the use of `synchronized` methods.

Although we have explained the use of threads for asynchronous requests using Java/RMI, the considerations are again directly applicable to CORBA and COM. If we used a Java programming language binding to CORBA/IDL or MIDL, we would just have to replace the remote method invocation with a call of a client stub generated by the IDL compiler.

Asynchronous requests can be implemented using synchronous requests and threads in COM, CORBA and Java/RMI.