Example 7.8
Multiple Request Implementation with Threads and Tuple Spaces

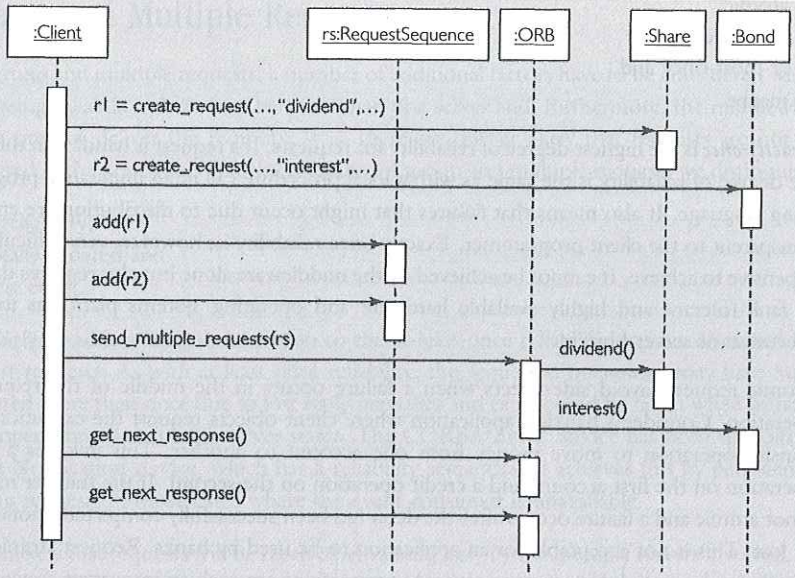
```
class BondRequest extends Thread {
   TupleSpace bag; Bond bond
   BondRequest(TupleSpace ts, Bond b){ bag = ts; bond=b; }
   public void run () {
      try {
          bag.out("result",new Double(bond.interest()));
      } catch (Exception e){}
   }
}

class ShareRequest extends Thread {
// analog to BondRequest
}

public class Client {
  static void main(String[] args) {
   Bond b; Share s;
   // ... initialization of b and s omitted
   double portfolio_revenue=0.0;
   Double result;
   TupleSpace ts = new TupleSpace();
   ShareRequest sr=new ShareRequest(ts,b); sr.start();
   BondRequest br=new BondRequest(ts,b); br.start();
   System.out.println("Do something else during requests");
   try {
    for (int i=0; i<2; i++){
    result=(Double)ts.in("result");
     portfolio_revenue=portfolio_revenue+result.floatValue();
     System.out.println(result.floatValue());
    }
   } catch (InterruptedException e){}
    System.out.println(portfolio_revenue);
  }
}
```

Figure 7.7
Excerpt of ORB Interface that Supports Multiple Requests

```
module CORBA {
  interface ORB {
    typedef sequence<Request> RequestSeq;
    Status send_multiple_requests(in RequestSeq targets);
    ...
  };
};
```



Example 7.9
Multiple Deferred Synchronous Requests in CORBA

A client creates a number of `Request` objects and concatenates them to a `RequestSeq`. Then the client invokes `send_multiple_request` from the `ORB` object for requesting the respective operations all at once. After that, the client invokes `get_next_response` as many times as there are request objects in the `RequestSeq`. When the operation returns, a new operation result is available and can be processed.

As Example 7.9 suggests, the use of multiple deferred synchronous requests is less difficult than when using threads, but is still rather involved. The client has to create a request object for every object that participates in the multiple request. After successful completion of the request, request objects have to be deleted again. Moreover, the client has to manage associations between request objects and the results of the requested operation. Hence the improvement of run-time performance is paid for by higher development costs.

7.3 Request Reliability

For the implementation of several non-functional requirements, in particular, fault-tolerance, reliability and availability it is important to be aware of the reliability with which middleware handles operation execution requests. When discussing *request reliability*, we separate unicast requests, where only one client and one server interact, from multiple and group requests.

7.3.1 Unicast Requests

For unicast requests, a number of different degrees of reliability can be distinguished. These are:

1. exactly-once,

2. atomic,
3. at-least-once,
4. at-most-once and
5. maybe.

Exactly-once requests are guaranteed to execute once and only once.

Exactly-once is the highest degree of reliability for requests. If a request is handled in this way the degree of reliability is the same as with a local procedure call in an imperative programming language. It also means that failures that might occur due to distribution are entirely transparent to the client programmer. Exactly-once reliability is, however, very difficult and expensive to achieve. It cannot be achieved by the middleware alone but also requires the use of fault-tolerant and highly available hardware and operating systems platforms for the execution of server objects.

Atomic requests are either performed completely or not at all.

Atomic requests avoid side-effects when a failure occurs in the middle of the requested operation. Consider a banking application where client objects request the execution of a transfer operation to move money from one account to another. This involves a debit operation on the first account and a credit operation on the second. If the transfer request is not atomic and a failure occurs after the debit has been successfully completed, money will be lost. This is not acceptable for an application to be used by banks. Request atomicity is usually implemented using *transactions*. As transactions are such an important concept for distributed applications, we have dedicated Chapter 11 to their discussion.

At-least-once requests are guaranteed to execute once, but possibly more often.

For requests executed with *at-least-once* reliability, the middleware guarantees that the request is executed. With this degree of reliability, however, the request may be executed more than once. This happens when reply messages indicating successful completion of the request are lost. Then the middleware does not know whether the request has been executed previously and it resends the request, which might lead to an additional execution of the request. Many message-oriented middleware (MOM) systems have at-least-once delivery semantics. Thus, if such a MOM is used to implement asynchronous requests, as suggested above, these requests have at-least-once semantics.

From an application designer's point of view, at-least-once requests are not very attractive in situations where operations modify the server's state. Consider the above debit operation. If it is executed more than once, money is lost. The application designer certainly does not want this to happen. Hence, care has to been taken when requesting operations that modify server states with at-least-once semantics.

At-most-once requests notify the client if a failure occurs.

With *at-most-once* reliability, the requested operation may or may not have been executed. If the operation is not executed the client is informed of the failure. The requests we discussed in Chapter 4 have at-most-once reliability. CORBA and Java/RMI inform the client of the failure by raising an exception and COM returns HRESULT values. If, however, the exceptions are not caught or COM return values are not evaluated, the request is executed with maybe reliability.

The client does not know whether a request with maybe semantics has been executed.

Clients do not know whether requests with *maybe* reliability have been executed. Middleware does not guarantee their execution and the client is not informed when the request fails. An example of requests that have this degree of reliability are the `oneway` requests in CORBA. Oneway operations must not raise any operation-specific exceptions, but the object request broker also does not raise generic exceptions if they fail. Hence, the client does not know whether the requested operation has been completed.

7.3.2 Group and Multiple Requests

For group and multiple requests, a number of additional factors have to be considered when discussing reliability: there may be more than one server and, furthermore, the middleware needs time to deliver the requests. It is, therefore, rather likely that requests are not all delivered at the same time. For the reliability of group and multiple requests we distinguish:

1. k-reliability,
2. totally ordered and
3. best effort.

K-reliability adds a second dimension to the at-least-once reliability that we discussed for unicast requests. As with at-least-once reliability, the requested operations may have been executed more than once due to lost reply messages and care should be taken when using it with operations that modify server states. The CORBA Event service has been specialized into a Notification service, which has k-reliability semantics. It achieves this by persistently storing requests in those cases where the event consumer is unavailable.

K-reliability guarantees that at least k requested operations will be executed.

It is unlikely that requests will be received simultaneously by the multiple servers involved. In most cases, this does not cause further problems. It might cause problems, however, if one request overtakes a request of an earlier cycle. *Totally ordered* group and multiple requests are, therefore, always completed before the next group or multiple request can be started. The CORBA Event service achieves totally ordered requests. This is done, however, at the expense of pushing an event through an event channel in a synchronous way. Hence clients cannot continue before the last member of the group has executed the requested operation.

Totally ordered group and multiple requests disable requests from overtaking previous requests.

Best effort is the lowest degree of reliability. No explicit measures are taken to guarantee a certain quality. Best effort reliability is comparable to the maybe reliability that was introduced for unicast requests. The multiple request primitives provided by CORBA only achieve a best effort as they do not guarantee atomicity nor any order on the request delivery.

Best effort requests do not make any quality of service guarantees.

Having discussed these different degrees of reliability, we note a general trade-off between reliability and performance. Achieving a high degree of reliability is very involved. It requires persistent storage of messages or the use of transactions, which are both rather time-consuming. On the other hand, requests that may or may not have been executed can be dealt with very quickly. Oneway requests in CORBA do not require reply messages to be returned to the client and the client therefore does not need to synchronize with the server. It is the task of the application designer to trade these different characteristics against each other in order to meet the mix of performance requirements that the stakeholders have specified.

Engineers trade high reliability against performance.

Key Points

- Requests in Java/RMI, CORBA and COM are, by default, unicast, synchronous, and executed at-most-once. Clients need to identify servers with an object reference.