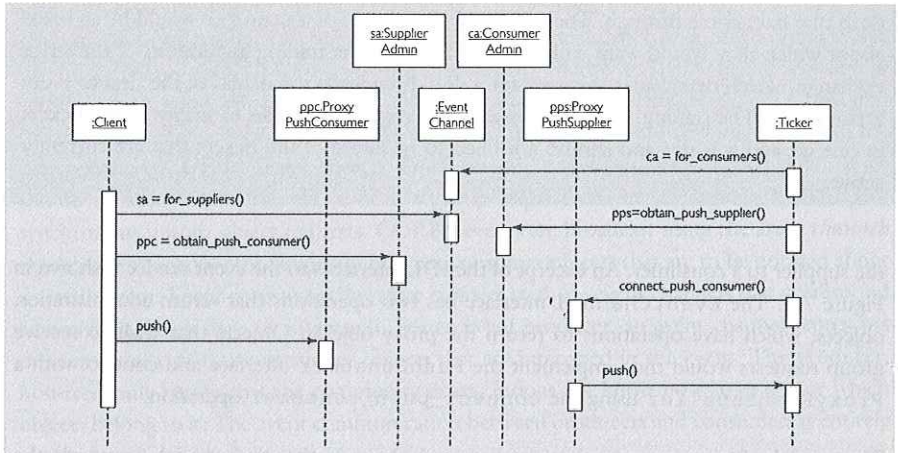
Figure 7.5
Excerpt of CORBA Event Service Interface

```
module CosEventComm {
  exception Disconnected{};
  interface PushConsumer {
    void push (in any data) raises(Disconnected);
  };
};

module CosEventChannelAdmin {
  exception AlreadyConnected {};
  exception TypeError {};
  interface ProxyPushConsumer: CosEventComm::PushConsumer {
  };
  interface ProxyPushSupplier: CosEventComm::PushSupplier {
    void connect_push_consumer(
          in CosEventComm::PushConsumer push_consumer
    ) raises(AlreadyConnected, TypeError);
  };
  interface ConsumerAdmin {
    ProxyPushSupplier obtain_push_supplier();
  };
  interface SupplierAdmin {
    ProxyPushConsumer obtain_push_consumer();
  };
  interface EventChannel {
    ConsumerAdmin for_consumers();
    SupplierAdmin for_suppliers();
  };
};
```

Example 7.6
Group Communication using Event Channels



The above figure shows how the group communication scenario from Example 7.5 is implemented using the CORBA Event service. Each Ticker object first obtains a reference to a `ConsumerAdmin` object and a `ProxyPushSupplier` object. It then registers with the event channel by requesting execution of `connect_push_consumer` and passing itself as an actual parameter. The initialization for an event producer is very similar. It obtains a reference to a `SupplierAdmin` object and a `ProxyPushConsumer` object. The event producer can then request the `push` operation that the proxy has inherited from `PushConsumer`. The proxy finally requests the `push` operation from each of the currently-connected `Ticker` objects.

7.2.2 Multiple Requests

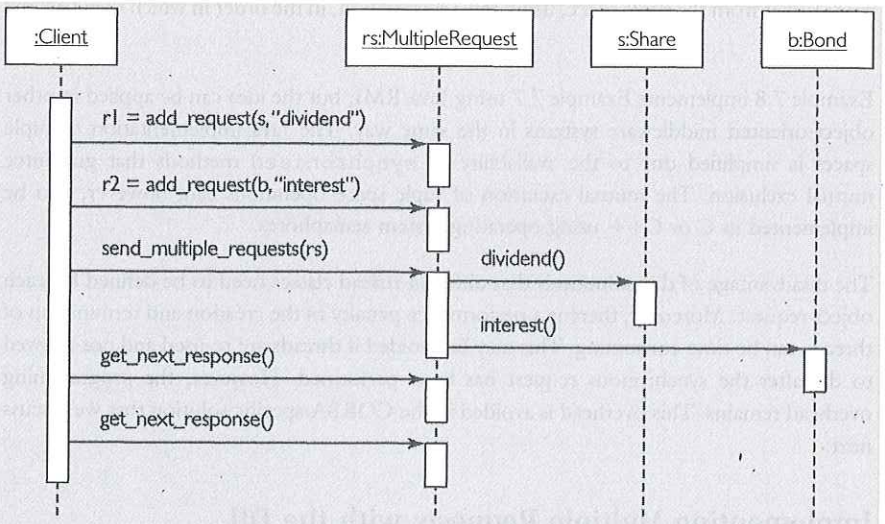
If a client wants to request a number of operation executions at the same time, it can be advantageous to cluster them together in a *multiple request*. The middleware can concurrently transfer multiple requests to the different server objects involved and the client can obtain the results when they become available. Multiple requests, however, are only viable if the requested operations are entirely independent from each other and if there are no precedence dependencies between the requested operations.

Multiple requests invoke operations from multiple server objects that are known to the client.

The main advantage of multiple requests is that the client can collect the results as they become available.

With a multiple request, the client can collect results as they become available.

We now focus on how multiple requests can be implemented. We first discuss a general solution that is applicable to all forms of distributed object middleware and relies on threads and synchronous object requests. We then discuss a more efficient solution that is available only in CORBA and uses the dynamic invocation interface.



Example 7.7
A Multiple Request

Consider a portfolio of securities that includes shares and bonds. If we want to calculate the revenue for the portfolio, we need to request execution of a dividend operation from every share object and the execution of an interest operation from every bond object. If these objects reside on different machines, they can be executed concurrently and the client obtains the results in the order they become available.

Using Threads

When we discussed synchronization primitives in the first section, we saw that all forms of synchronization can be achieved by making a synchronous request in one thread. We can extend this idea for multiple requests. The difference is that we do not just create one thread

to make the synchronous request, but we create a new thread for every operation that we want to request. A further difference is the synchronization of the client with the request threads. The main benefit of multiple requests, which we want to retain in any implementation, is that we obtain the results in the order in which they become available. In order to achieve this synchronization in Java, we use tuple spaces that were first proposed for so-called coordination languages [Carriero and Gelernter, 1992], in particular Linda [Gelernter, 1985]. The basic idea of tuple spaces is to have a bag that contains tagged data, which are read and updated by concurrent processes. A tuple space provides various operations for manipulating the *tuple space* content. To avoid any interference, these operations have to be executed in mutual exclusion. The `out` operation puts a new tuple into the space. The `in` operation removes a tuple from the space and returns it. The `rd` operation reads a tuple from the space without removing it. Both `in` and `rd` block the caller if the desired tuple is not available. Figure 7.6 shows an implementation of tuple spaces in Java that we adopted from [Magee and Kramer, 1999].

To implement multiple requests, the client spawns multiple threads and performs a synchronous object request in each thread. As soon as the request is finished, the client thread puts the result into a tuple space, using the `out` operation, and terminates. The client thread reads tuples from the tuple space, using the `in` operation, in the order in which they become available.

Multiple requests can be implemented using tuple spaces in CORBA, COM and Java/RMI.

Example 7.8 implements Example 7.7 using Java/RMI, but the idea can be applied in other object-oriented middleware systems in the same way. The Java implementation of tuple spaces is simplified due to the availability of `synchronized` methods that guarantee mutual exclusion. The mutual exclusion of tuple space operations can, however, also be implemented in C or C++ using operating system semaphores.

Use of threads implies performance and development overheads.

The disadvantage of this solution is that different thread classes need to be defined for each object request. Moreover, there is a performance penalty as the creation and termination of threads can be time-consuming. This may be avoided if threads are re-used and not allowed to die after the synchronous request has been performed. However, the programming overhead remains. This overhead is avoided in the CORBA-specific solution that we discuss next.

Implementing Multiple Requests with the DII

Multiple requests can be implemented with the Dynamic Invocation Interface.

We have noted above that multiple requests have to be dynamic as they cannot be defined using a static way of invoking a stub from any programming language. Indeed, multiple CORBA requests are supported by CORBA's dynamic invocation interface. Multiple CORBA requests with the DII are referred to as *multiple deferred synchronous requests.* The DII includes an operation to request multiple operation executions from several CORBA server objects. This is achieved by the operation `send_multiple_requests` whose interface is sketched in Figure 7.7.

The operation takes a list of `Request` objects, each of which was created using the `create_request` operation of the ORB interface. After multiple requests have been sent, the ORB can concurrently deal with them and communicate with the run-time of the server objects. Using `send_multiple_requests` rather than invoking the `send`

Figure 7.6
Tuple Space Implementation in Java

```
import java.util.*;
 private Hashtable tuples = new Hashtable();

 // deposits data in tuple space (ts)
 public synchronized void out (String tag, Object data) {
  Vector v = (Vector) tuples.get(tag);
  if (v==null) {
   v=new Vector();
   tuples.put(tag,v);
  }
  v.addElement(data);
  notifyAll();
 }

 private Object get(String tag, boolean remove) {
  Vector v = (Vector) tuples.get(tag);
  if (v==null) return null;
  if (v.size()==0) return null;
  Object o = v.firstElement();
  if (remove) v.removeElementAt(0);
  return o;
 }

 //extracts tagged object from ts, blocks if unavailable
 public synchronized Object in (String tag)
        throws InterruptedException {
  Object o;
  while ((o=get(tag,true))==null) wait();
  return o;
 }

 //reads tagged object from ts, blocks if unavailable
 public synchronized Object rd (String tag)
        throws InterruptedException {
  Object o;
  while ((o=get(tag,false))==null) wait();
  return o;
 }
}
```

operation of the `Request` interface several times reduces communication costs between the client and the ORB run-time environment.

The synchronization of a multiple request is slightly different from that of a single request. Synchronisation uses `get_next_response` to return the next response of a server object that has terminated the requested operation. Note that the order in which responses are returned is not necessarily the order of the requests. The advantage is that no polling is necessary but responses are returned in the order in which they become available.