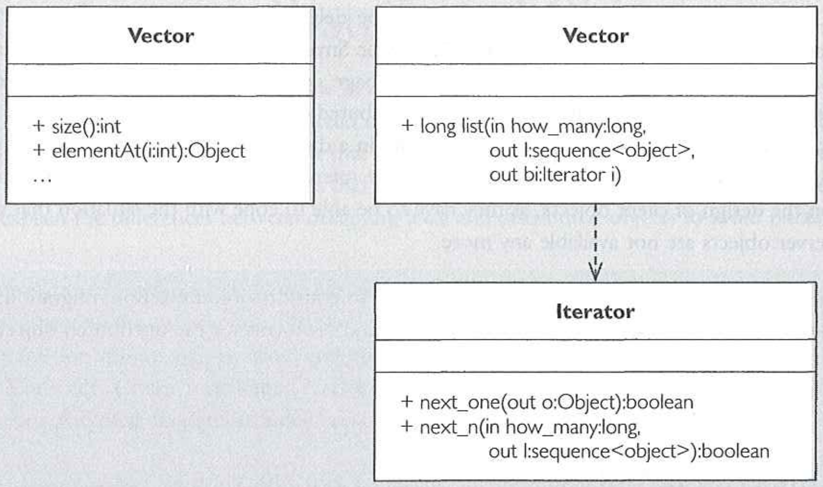
Example 2.15
Interfaces for Local and Remote Iterations


Vector
+ size():int
+ elementAt(i:int):Object
...
Vector
+ long list(in how_many:long, out l:sequence<object>, out bi:Iterator i)
Iterator
+ next_one(out o:Object):boolean
+ next_n(in how_many:long, out l:sequence<object>):boolean

The diagram on the left depicts an excerpt of class `Vector` of the Java `util` package. `Vector` exports an operation `elementAt` that returns the object at a particular position in the vector. Operation `size` returns the number of elements in the vector. An iteration that visits all elements in the vector can then be implemented in a loop ranging from `0` to `size()-1`. Assuming just-in-time compilation of the Java byte code for this loop, the overhead of invoking `elementAt` for a vector with 1,000 elements will be roughly 250 microseconds, which is negligible. On the other hand, if the request for `elementAt` comes a thousand times from a `Vector` object on a remote host, the overhead would be roughly half a second and this is no longer negligible.

The diagram on the right shows a better design. Operation `list` returns a sequence of object references that contains at most the number of elements specified by the parameter `how_many`. If the vector has further elements, `list` also returns a reference to a `BindingIterator`. The iterator can be used to obtain further sequences of elements. This design solves the performance problem because a sequence containing many object references can be transferred at once. Local operations provided for that sequence would then be used to visit the elements of that sequence. If we were to obtain the 1,000 elements in batches of 100 each, the overhead would be 10 remote invocations plus 1,000 local calls to obtain the elements from the sequence. We can assume that the size of an object reference is somewhere below 500 bytes so that the overall size of the returned list will be less than 64KB. Transmitting this amount of data does not considerably increase the overall time of the request. Thus the performance of this approach requires $10 \times 500 + 1000 \times 0.25 = 5250$ microseconds. Hence, the performance of this design will be 100 times better than the naive solution we discussed above.

This is considerably different when requesting an operation from an object that resides on a different host. The overhead of such a request is between 0.1 and 10 milliseconds, depending on the technology that is being used. We have measured an object request brokered by Orbix between two SPARC Ultra servers connected with a 100 Mbit network. Approximately 500 microseconds were needed for that request. Hence, object requests take about 2,000 times as long as local method calls and this overhead is not negligible during the design of distributed objects. Example 2.15 presents the impact on the design.

In order to optimize poor performance, designers need to have quite deep knowledge about the particular approach that is used to solve distribution, as well as about the problem domain. In the above example, we had to make assumptions about the overall cost of a simple object request, the size of the representation of an object reference and the increase of latency depending on the size of the transmitted information. Without this basic knowledge and understanding of how distributed object requests are implemented, designers cannot reasonably build efficient systems. We discuss this in the chapters of Part II

Request latency has to be taken into account when designing distributed objects.

2.4.4 Object Activation

Objects created with an object-oriented programming language reside in virtual memory between the time when they are created and when they are deleted. This is inappropriate for distributed objects for the following reasons:

Distributed objects may not always be available to serve an object request at any point in time.

1. Hosts sometimes have to be shut down and then objects hosted on these machines have to be stopped and re-started when the host resumes operation;
2. The resources required by all the server objects on a host may be greater than the resources the host can provide; and
3. Depending on the nature of the client application, objects may be idle for a long time and it would be a waste of resources if they were kept in virtual memory all the time.

For these reasons, we consider two operations in addition to the principal life cycle operations we discussed above. These are activation and deactivation. *Activation* launches a previously inactive object, brings its implementation into virtual memory and thus enables it to serve object requests. *Deactivation* is the reverse operation: it terminates execution of the object and frees the resources that the object currently occupies.

Activation brings previously inactive objects into main memory so that they can serve an object request.

We note that activation may increase the latency of requests further. If an object has to be activated prior to being able to serve a request, the time needed for that activation increases the request latency. This time depends on the operating system of the server host. Very often this is the time needed to start an operating system process, which is quite significant. It is, therefore, important to devise and implement policies that minimize the need for activation and deactivation.

The overhead of activation adds considerably to the latency of object requests.

Activation and deactivation of distributed objects should be transparent to users and programmers of client objects. Activation and deactivation may also be transparent to administrators if they are fully implemented by a distribution middleware. This requires, however, that administrators provide sufficient installation and policy information to enable the middleware to implement activation and deactivation. That installation and policy information includes a repository for object implementations as well as activation and deactivation strategies.

Activation and deactivation should be transparent.

Activation and deactivation is, however, often not transparent to designers of server objects. Some server objects expose a state at their interface. Even objects that do not export any attributes might have an internal state. Upon deactivation, that state must be stored on

Stateful objects have to be persistent.

persistent storage so that it can be retrieved upon activation. Hence, designers of such server objects must make provisions for the state to become persistent upon deactivation and to be retrieved upon activation. Persistence of distributed objects can be achieved in the same way as with local objects. Techniques include serialisation and storage in a file system, mapping to relational databases or mapping to object databases. We discuss such techniques in Chapter 10. Persistence of distributed objects, however, has to be integrated with the implementation of activation and deactivation.

2.4.5 Parallelism

Objects that are written in an object-oriented programming language are usually executed sequentially in a process on a host. Some object-oriented programming languages and many operating systems support concurrent programming by offering threads that can be used to execute objects concurrently. Threads implement an interleaved model of concurrency that enables multiple threads to share the same processor. Figure 2.3 shows an example of three processes `A`, `B` and `C` that are executed in an interleaved way on one processor. Only one process is active at a time and a scheduler switches between these processes. Hence, it is not possible to accelerate a job by distributing it across multiple threads. The threads will all be executed on the same processor and that processor restricts the performance of the job.

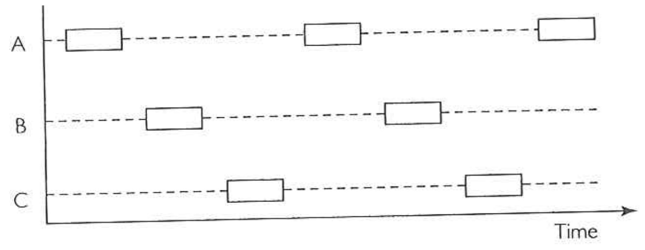


Figure 2.3
Interleaved Model of Concurrency

Distributed objects often execute in parallel.

If a host has more than one processor, threads may actually physically execute in parallel. The client object that requests a service always executes in parallel with the server object. If it requests services in parallel from several server objects that are on different machines, these requests can all be executed in parallel. Hence, distribution has the potential to decrease considerably the elapsed real time needed for a job. This can be achieved if the tasks performed by operations are rather coarse-grained so that the overhead for request latency is small compared to the time needed to perform an operation.

Concurrency may have to be controlled.

Interleaved concurrent execution of two operations may lead to integrity violations, such as inconsistent analysis or lost updates. Given that distributed objects are executed concurrently all the time we may have to control concurrency if the integrity of objects is at risk. Unlike objects stored in databases there is no central control in a distributed system that implements concurrency control. It is therefore necessary for designers of distributed objects to be aware of and implement concurrency control within the server objects they build. We discuss techniques and protocols for concurrency control in Chapter 11.

2.4.6 Communication

There are different communication primitives available for distributed object requests.

In object-oriented programming, requests are implemented as methods call. Such calls block the calling object until a point in time when the result becomes available. Distributed object communication aims at mimicking method calls as closely as possible. Hence, the default form for object requests is synchronous communication and the client is blocked until the server object returns the result.

Due to the request latency inherent to communication between distributed objects it might be inappropriate to force a client to await completion of a request; additional forms of synchronization are needed. They might optimize synchronization for those operations whose signatures indicate that no results have to be transmitted to the client. When designing interfaces it might therefore be advantageous to take such optimizations into account.

Communications between objects in a centralized application occur between two peers. For a method call there is just one caller and one called operation. Again, this is the default for communication between distributed objects and it is referred to as peer-to-peer communication.

Peer-to-peer communication might not always be appropriate for distributed objects, where often multiple distributed objects need to communicate with each other. Hence, distributed object communication primitives need to be designed that facilitate communication between groups of objects. Moreover, it might be advantageous if multiple requests to one or several objects can be processed at once. We focus on advanced communication, that is non-synchronous communication, communication between groups of objects, and request multiplicity in Chapter 7.

2.4.7 Failures

Distributed object requests are more likely to fail than local method invocations.

In the previous chapter, we argued that distributed systems fail more often than centralized systems. This has a number of implications on the design of distributed objects. Unlike objects in centralized applications, distributed objects have to be designed in such a way that they cope with failures. One could take the point of view that the underlying middleware should aim at minimizing failures and implement requests with exactly-once semantics. With exactly-once semantics the middleware would guarantee that every request is executed once and only once.

Different reliabilities are available for distributed objects.

While it is generally possible for a system to implement exactly-once semantics, there is a clear trade-off. An implementation of exactly-once would generally need to rely on persistent storage of requests and thus it would add considerably to the complexity of request implementation. As a result, request latency would be increased further. This might not be appropriate for applications that need a low request latency but can tolerate lost requests. The approach taken by most middleware systems is therefore that they implement at-most-once semantics as a default. This means that they execute every request at most once but they tell clients when a request has failed. Additionally, they provide further mechanisms that designers can use to ensure higher or lower level request reliability. We will revisit request reliability in Chapter 7.