Clients need to check whether servers have executed a request

The fact that requests may fail, of course, implies some obligations for the design of a client. Unlike method calls, clients cannot assume that requests have been executed properly when the request returns control to the client. They have to check for exceptions that occurred while the request was executed. Clients sometimes also have a sequence of requests that should be done either together and completely or not at all. As an example, consider a funds transfer from one account to another account. It is usually composed of a debit and a credit operation. If after the debit operation is completed successfully the credit operation fails then the effect of the debit operation should also be undone in order to avoid an erroneous balance on the accounts. To make sure that such a sequence of requests is performed completely or not at all is supported by the concept of transactions. Clients have the obligation to declare the beginning and the end of a transaction.

Designers of server objects have to build server objects in such a way that they can participate in transactions. They must make provisions to undo the effect of changes. All the server objects that participate in a transaction must have successfully completed their operations before the effect of a transaction can be made permanent. In Chapter 11, we discuss protocols and interfaces that are used for the implementation of distributed transactions.

2.4.8 Security

Distribution of objects makes them vulnerable to security attacks.

Centralized applications that are written using object-oriented programming languages often deal with security at a session level. They request users to identify themselves during an authentication procedure at session start and then they trust that the user will not make the session available to unauthorized users.

Distributed objects may communicate over insecure networks, where attackers can openly access network traffic. In this setting it is far more important than in centralized systems that both client and server objects can be certain that they are actually interacting with the object with which they think they are interacting. In addition, different users might make requests of a distributed object in parallel. These requests will then be performed in an interleaved way. Hence, it is not sufficient to authenticate at a session level but individual requests might have to be authenticated. Moreover, server objects have to be able to decide whether a user or client that has requested execution of an operation is authorized to execute that request. Hence, they have to control access to their operations. Finally, they might have to generate irrefutable evidence that they have delivered a request to a client in order to be able sustain claims for payment for the service provided during that request.

Security has to be considered during the design of distributed objects.

All these problems affect the design of both the client and the server object. As security impacts the design of both server and client objects, we will discuss encryption techniques as basic primitives for implementing security. We will discuss in Chapter 12 the techniques and interfaces that support authentication, auditing, access control and non-repudiation services that can be used in the design of distributed objects.

Key Points

- A component model defines how services are defined, how the externally visible state of a component is determined, how components are identified and how services are requested. In this book we take an object-oriented perspective on distributed systems. Hence components are considered as objects. Objects communicate using object requests. A client object may request the execution of an operation or access to an attribute from a server object.
- We have very briefly introduced the Unified Modeling Language. It includes a set of graphical notations that are useful for expressing object models from different perspectives. We have discussed use case diagrams, which can be used to identify the need for system interfaces. Sequence diagrams are used to model scenarios of object interactions. Class diagrams can be used to model the static properties of classes, in particular the attributes and operations of a class and the association the class has with other classes. State diagrams express the internal state of distributed objects as well as transitions between these states. These diagrams support the design of, and the communication about, distributed objects.
- We have discussed the ingredients of a meta-object model for distributed objects. This included operations, attributes, exceptions, subtyping, multiple inheritance and polymorphism. Instances of the meta-object model are object types that designers of distributed systems have to determine. These types define properties that their instances have in common.
- We have discussed the differences between designing centralized objects that are implemented with an object-oriented programming language and distributed objects. These differences were many-fold. The life cycle of distributed objects is more complicated as location information has to be considered. References to distributed objects are more complex as they have to be able to locate objects and because security and type information may have to be added. The latency of requests from distributed objects is about 2,000 times greater than the overhead of calling a method locally. Objects might have to be activated prior to serving a request and they might have to be deactivated if they are no longer used and this adds further to the request latency. Distributed objects have a potential for performing tasks faster than centralized objects due to real parallelism. To cope with request latency, there will have to be additional forms of communication between synchronous and peer-to-peer requests. Distributed objects may fail and therefore both client and server objects have to be built in such a way that they can tolerate failures. Finally, distributed objects communicate via possibly insecure networks and therefore they have to be designed in such a way that security and privacy is not affected.

Self Assessment

2.1 What is an object?
2.2 What is an object type?
2.3 What is an object request?

2.4 What is the difference between an object and an object type?
2.5 How are services modelled in an object-oriented meta-model?
2.6 Why does a meta-object model for distributed objects also include non-object types?
2.7 Give examples when you would use polymorphism in the design of distributed objects.
2.8 In which stage of the software process would you utilize UML use case diagrams?
2.9 What are UML sequence diagrams used for?
2.10 What is the difference between private, protected and public declarations in UML class diagrams?
2.11 What consistency relationships have to hold between sequence diagrams and class diagrams?
2.12 Explain reasons why distribution has to be considered during the design of distributed objects.

Further Reading

The OMG object model is described in the Object Management Architecture Guide [Soley, 1992]. It also includes an exhaustive and comprehensive glossary of object-oriented terminology. A detailed description of the object model of Microsoft's COM is provided by [Box, 1998] and Chapter 5 of [Grimes, 1997] explains the extensions that enable distributed access to COM objects.

The Unified Modeling Language was adopted as an OMG Standard in November 1997. The OMG maintains an extensive web site with about ten documents on UML. The most important of those are the UML Semantics Guide [OMG, 1997a] and the UML Notation Guide [OMG, 1997c]. While these are the definite references about UML, they are not the most readable introductions. [Booch et al., 1999] define a user guide to the UML. [Rumbaugh *et al.*, 1999] provide a reference manual to the UML and [Jacobson *et al.*, 1999] discuss a software development process model that uses the UML. A thorough introduction to UML is also provided by [Muller, 1997] and a very concise overview is given by [Fowler and Scott, 1997]. [Quatrani, 1998] discusses how to use a CASE tool in order to develop UML models.

It is important that readers are aware that distributed systems are always parallel systems. A lot of work has been done on building concurrent systems and there is no need for us to elaborate on that. The theoretical foundations for building parallel and concurrent systems have been laid by the work of [Hoare, 1978] on CSP and [Milner, 1989] on CCS. A very practical approach to concurrency using design patterns and Java's thread model is discussed in [Lea, 1997]. [Magee and Kramer, 1999] pursue an engineering approach to the design and implementation of concurrent object-oriented programs that is very similar to the techniques we present for distributed objects in this book. Magee and Kramer introduce FSP, a process algebra for finite state processes, and describe systematic transformations of FSP specifications into concurrent Java implementations.