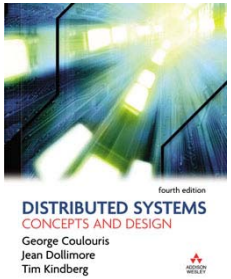
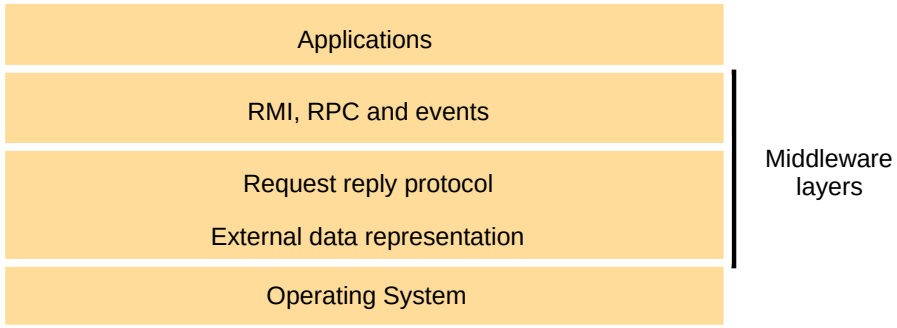
Slides for Chapter 5:
Distributed objects and remote invocation



From Coulouris, Dollimore and Kindberg
Distributed Systems:
Concepts and Design

Edition 4, © Addison-Wesley 2005

Figure 5.1
Middleware layers



5.1 Introduction

- Middleware
  - Software that provides a programming model above the basic building blocks of processes and message passing.
  - An important aspect of middleware is the provision of location transparency and independence from the details of communication protocols, operating systems, and computer hardware.
  - Location transparency:
    - In RPC, the client that calls a procedure cannot tell whether the procedure runs in the same process or in a different process, nor does the client need to know the location of the server.
    - In RMI, the object making the invocation cannot tell whether the object it invokes is local or remote.
    - In distributed event-based programs, the objects generating events and the objects that receive notifications of those events need not be aware of one another's location.

- Middleware (Cont'd)
  - Communication protocols: The protocols that support the middleware abstractions are independent of the underlying transport protocols.
  - Computer hardware: External data representations are described in last chapter.
  - Operating systems: The higher-level abstractions provided by the middleware layer are independent of the underlying operating systems.
  - Use of several programming language: some middleware is designed to allow distributed applications to use more than one programming language.

5.1.1 Interfaces

- An explicit interface is defined for each programming module in order to control the possible interactions between modules.
- The interface of a module specifies the procedures and the variables that can be accessed from other modules.
- Modules are implemented so as to hide all information about them except that which is available through its interface.

- Interfaces in distributed systems:
  - In a distributed program, the modules can run in separate processes.
  - It is not possible for a module running in one process to access the variables in a module in another process – thus, the interface of a module that is intended for RPC or RMI cannot specify direct access to variables.
  - The specification of a procedure or method in the interface of a module in a distributed program describes the parameters as input or output.
  - Input parameters are passed to the remote module by sending the values of the arguments in the request message and then supplying them as arguments to the operation to be executed in the server.
  - Output parameters are returned in the reply message.
  - When a parameter is used for both input and output, the value must be transmitted in both the request and reply messages.
  - Another different between local and remote modules is that pointers in one process are not valid in another remote one.

- Service interfaces:
  - In the client-server model, each server provides a set of procedures that are available for use by clients.
  - The service interface is used to refer to the specification of the procedures offered by a server, defining the types of the input and output arguments of each of the procedures.
- Remote interfaces:
  - In the distributed object model, a remote interface specifies the methods of an object that are available for invocation by objects in other processes, defining the types of the input and output arguments of each of them.
  - The difference is that the methods in remote interfaces can pass object as arguments and results of methods.
  - Reference to remote objects may also be passed.

- Interface definition languages:
  - An RMI mechanism can be integrated with a particular programming language if it includes an adequate notation for defining interfaces – allowing input and output parameters to be mapped onto the language's normal use of parameters.

Figure 5.2
CORBA IDL example

```
// In file Person.idl
struct Person {
        string name;
        string place;
        long year;
} ;
interface PersonList {
        readonly attribute string listname;
        void addPerson(in Person p) ;
        void getPerson(in string name, out Person p);
        long number();
};
```


5.2 Communication between distributed objects

- Address communication between distributed objects by means of RMI. The material is presented under the following headings
  - The object model
  - Distributed objects
  - The distributed object models
  - Design issues
  - Implementation
  - Distributed garbage collection

5.2.1 The object model

- An object's data should be accessible only via its methods.
- Object references:
  - Objects can be accessible via object reference.
  - To invoke a method in an object, the object reference and method name are given, together with any necessary arguments.
  - The object whose method is invoked is called the *target* or *receiver*.
- Interfaces:
  - Provides a definition of the signatures of a set of methods without specifying their implementation.

- Actions:
  - Action in an object-oriented program is initiated by an object invoking a method in another object.
  - An invocation can include additional information (arguments) needed to carry out the method.
  - The receiver executes the appropriate method and then returns control to the invoking object.
  - An invocation of a method can have three effects:
    - The state of the receiver may be changed.
    - A new object may be instantiated.
    - Further invocations on methods in other objects may take places.

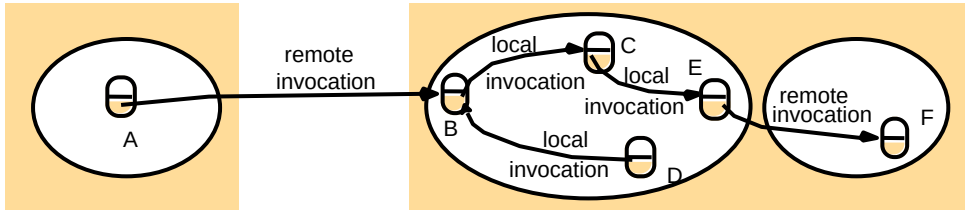
- Exceptions:
  - Programs can encounter many sorts of errors and unexpected conditions of varying seriousness.
  - Exceptions provide a clean way to deal with error conditions without complicating the code.
  - A block of code may be defined to *throw* an exception whenever particular unexpected conditions or error arise.
  - This means that control passes to another block of code that *catches* the exception.
  - Control does not return to the place where the exception was thrown.

- Garbage collection
  - Necessary to provide a means of freeing the space occupied by objects when they are no longer needed.
  - This process is called *garbage collection*.
  - When a language does not support garbage collection, the programmer has to cope with the freeing of space allocated to objects.

5.2.2 Distributed objects

- May adopt the client-server architecture.
  - Objects are managed by servers and their clients invoke their methods using remote method invocation.
  - In RMI, the client's request to invoke a method of an object is sent in a message to the server managing the object.
  - The invocation is carried out by executing a method of the object at the server and the result is returned to the client in another message.
- Can assume other architectural models.
  - E.g., objects can be replicated in order to obtain the usual benefits of fault tolerance and enhance performance.
- The fact that objects are accessed only via their methods gives another advantage for heterogeneous systems in that different formats may be used at different sites.

Figure 5.3
Remote and local method invocations

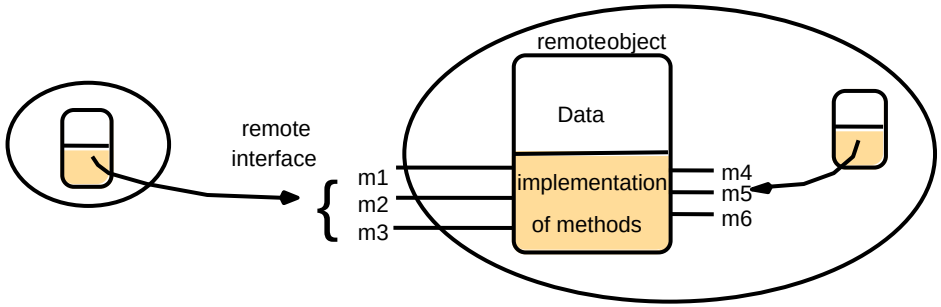


5.2.3 The distributed object model

- Each process contains a collection of objects
  - Some of which can receive both local and remote invocations.
  - The other objects can receive only local invocations.
- Method invocations between objects in different processes are known as *remote method invocations*.
- Method invocations between objects in the same process are local method invocations.
- Two fundamental concepts
  - Remote object reference: Other objects can invoke the methods of a remote object if they have access to its *remote object reference*.
  - Remote interface: Every remote objects has a *remote interface* that specifies which of its methods can be invoked remotely.

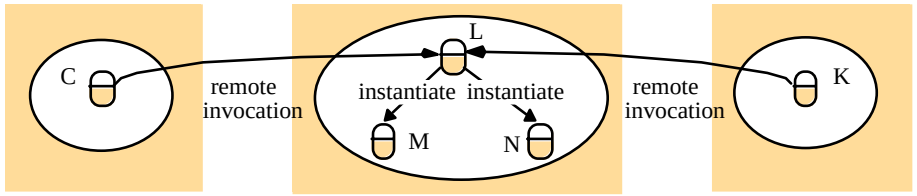
- Remote object references:
  - Remote object reference is an identifier that can be used throughout a distributed system to refer to a particular unique remote object.
  - Remote object references are analogous to local ones in that:
    - The remote object to receive a remote method invocation is specified by the invoker as a remote object references
    - Remote object references may be passed as arguments and results of remote method invocations.
- Remote interfaces:
  - The class of a remote object implements the methods of its remote interface.
  - Objects in other processes can invoke only the methods that belong to its remote interface.
  - Local objects can invoke the methods in the remote interfaces as well as other methods implemented by a remote object.

Figure 5.4
A remote object and its remote interface



- Actions in a distributed object system:
  - An action is initiated by a method invocation – which may result in further invocations on methods in other objects.
  - However, in the distributed case, the objects involved in a chain of related invocations may be located in different processes or different computers.
  - When an invocation crosses the boundary of a process or computers, RMI is used, and the remote reference of the object must be available to the invoker.
  - When an action leads to the instantiation of a new object, that object will normally live within the process where instantiation is requested.
  - Distributed applications may provide remote objects with methods for instantiating objects which can be accessed by RMI.

Figure 5.5 Instantiation of remote objects



- Garbage collection in a distributed-object system:
  - If a language supports garbage collection, then any associated RMI system should allow garbage collection of remote objects.
  - Distributed garbage collection is generally achieved by cooperation between the existing local garbage collector and an added module that carries out a form of distributed garbage collection.
  - If garbage collection is not available, then remote objects that are no longer required should be deleted.
- Exceptions:
  - Any remote invocation may fail for reasons related to the invoked object being in a different process or computer from the invoker.
  - Remote method invocation should be able to raise exceptions.

5.2.4 Design issues for RMI

- RMI is a natural extension of local method invocation.
- RMI invocation semantics:
  - Request-reply protocols where *doOperation* can be implemented in different ways to provide different delivery guarantees.
    - Retry request message: whether to retransmit the request message until either a reply is received or the server is assumed to have failed.
    - Duplicate filtering: when retransmissions are used, whether to filter out duplicate requests at the server.
    - Retransmission of results: whether to keep a history of result messages to enable lost results to be retransmitted without re-executing the operations at the server.
  - Combinations of these choices lead to a variety of possible semantics for the reliability of remote invocations as seen by the invoker (Fig 5.6)

Figure 5.6
Invocation semantics

Fault tolerance measures			*Invocation semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*
Yes	No	Re-execute procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

Maybe invocation semantics:

- the remote method may be executed once or not at all.
- Maybe semantics arises when none of the fault tolerance measures is applied.
- Can suffer from the following types of failure:
  - Omission failure, if the invocation or result message is lost:
  - Crash failure, when the server containing the remote object fails.
- Maybe semantics is useful only for applications in which occasional failed invocations are acceptable.

At-least-once invocation semantics:

- The invoker receives either a result or an exception informing it that no result was received.
- Can suffer from the following types of failure:
  - Crash failures when the server containing the remote object fails.
  - Arbitrary failures. In cases when the invocation message is retransmitted, the remote object may receive it and execute the method more than once, possibly cause wrong values to be stored or returned.

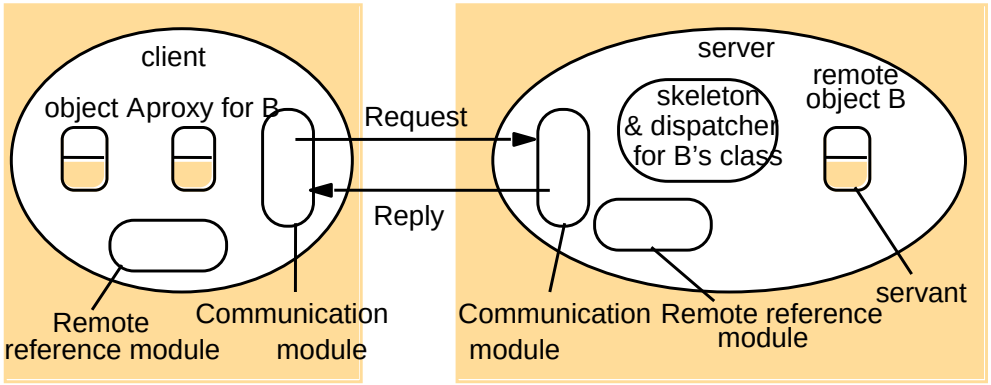
At-most-once invocation semantics:

- The invoker receives either a result or an exception informing it that no result was received, in which case the method will have been executed either once or not at all.

Transparency

- Remote invocations should be made transparent in the sense that the syntax of a remote invocation is the same as that of a local invocation, but that the difference between local and remote objects should be expressed in their interfaces.

Figure 5.7
The role of proxy and skeleton in remote method invocation



5.2.5 Implementation of RMI

- Figure 5.7
- Communication module
  - The two cooperating communication modules carry out the request-reply protocol.
  - The communication module in the server selects the dispatcher for the class of the object to be invoked, passing on its local reference, which it gets from the remote reference module in return for the remote object identifiers in the *request* message.

- Remote reference module
  - Responsible for translating between local and remote object references and for creating remote object references.
  - The remote reference module in each process has a *remote object table* that records the correspondence between local object references in that process and remote object references.
  - The actions of the remote reference module are as follows:
    - When a remote object is to be passed as argument or result for the first time, the remote reference module is asked to create a remote object reference, which it adds to its table.
    - When a remote object reference arrives in a request or reply message, the remote reference module is asked for the corresponding local object reference, which may refer either to a proxy or to a remote object.
  - This module is called by components of the RMI software when they are marshalling and unmarshalling remote object references.

- Servants
  - A servant is an instance of a class which provide the body of a remote object.
  - Handles the remote requests passed on by the corresponding skeleton.
  - Live within a server process.
  - Created when remote objects are instantiated and remain in use until they are no longer needed.
  - Finally, being garbage collected or deleted.

- The RMI software
  - Consists of a layer of software between the application-level objects and the communication and remote reference modules.
  - The roles of the middleware objects are as follows:
    - Proxy – to make remote method invocation transparent to clients by behaving like a local object to the invoker; instead of executing an invocation, it forwards it in a message to a remote object.
    - Dispatcher – A server has one dispatcher and skeleton for each class representing a remote object.
    - Skeleton – the class of a remote object has a skeleton, which implements the methods in the remote interface. A skeleton method unmarshals the arguments in the request message and invokes the corresponding method in the servant.

- Generation of the classes for proxies, dispatchers, and skeletons --- are generated automatically by an interface compiler.
- Server and client programs
  - The server program contains the classes for the dispatchers and skeletons, together with the implementations of the classes of all of the servants that it supports.
  - In addition, the server program contains an *initialization* section – for creating and initializing at least one of the servants to be hosted by the server.
  - The client program will contain the classes of the proxies for all of the remote objects that it will invoke.

- The binder:
  - A binder in a distributed system is a separate service that maintains a table containing mappings from texture names to remote object references.
  - It is used by servers to register their remote objects by name and by clients to look them up.
- Server threads:
  - Whenever an object executes a remote invocation, that execution may lead to further invocations of methods in other remote objects – which may take some time to return.
  - To avoid the execution of one remote invocation delaying the execution of another, servers generally allocate a separate thread for the execution of each remote invocation.

- Activation of remote objects
  - Some applications require that information survive for long periods of time. However, it's not practical for the objects representing such information to be kept in running processes for unlimited periods.
  - To avoid the potential waste of resources, the servers can be started whenever they are needed by clients.
  - Processes that start server processes to host remote objects are called *activators* for the following reasons.
    - A remote object is described as *active* when it is available for invocation within a running process.
    - It is called *passive* if it is not currently active but can be made active.
  - A passive object consists of two parts
    - The implementation of its methods
    - Its state in the marshalled form.

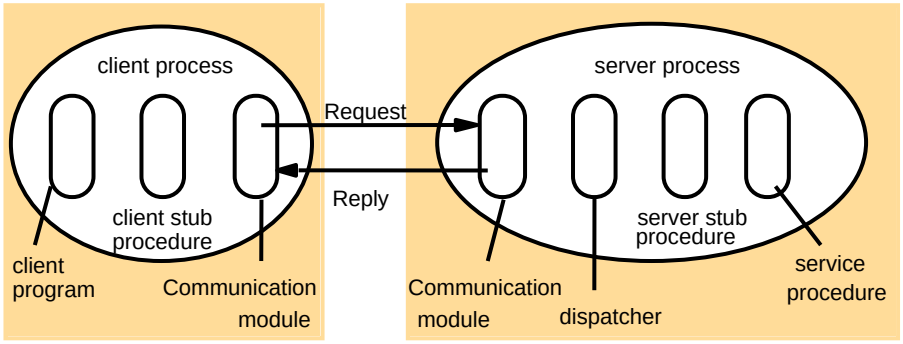
- Activation of remote objects (cont'd)
  - Activation consists of creating an active object from the corresponding passive object by creating a new instance of its class and initializing its instance variables from the stored state.
  - An *activator* is responsible for
    - Registering passive objects that are available for activation.
    - Starting named server processes and activating remote objects in them.
    - Keeping track of the locations of the servers for remote objects that it has already activated.

- Object location
  - A *location service* helps clients to locate remote objects from their remote object references.
  - Use a database that maps remote object references to their probable current locations – the location are probable because an object may have migrated again since it was last heard of.

5.2.6 Distributed garbage collection

- Aim: to ensure that if a local or remote reference to an object is still held anywhere in a set of distributed objects, then the object itself will continue to exist, but as soon as no object any longer holds a reference to it, the object will be collected and the memory it uses recovered.

Figure 5.8 Role of client and server stub procedures in RPC in the context of a procedural language

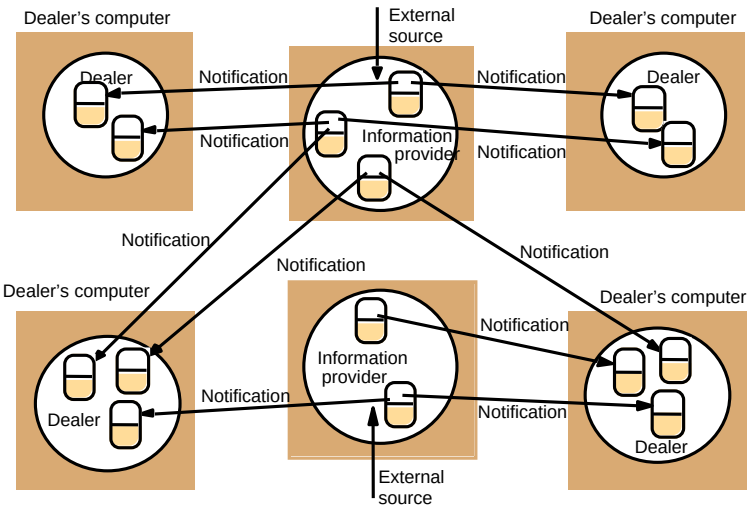


5.3 Remote procedure call

- Very similar to an RMI in that a client program calls a procedure in another program running in a server process.
- Servers may be clients of other servers to allow chains of RPCs.
- However, it lacks the ability to create new instances of objects and therefore does not support remote object references.
- RPC may be implemented to have one of the choices of invocation semantics discussed in Sec 5.2.4 – at-least-once or at-most-once are generally chosen.
- Generally implemented over a request-reply protocol.
- This software is similar to RMI except that no remote reference modules are required, since procedure call is not concerned with objects and object references.

- The client that accesses a service includes one *stub procedure* for each procedure in the service interface. It behaves like a local procedure to the client, but instead of executing the call, it marshals the procedure identifier and the arguments into a request message – which it sends via its communication module to the server.
- When the reply message arrives, it unmarshals the results.
- The server process contains a dispatcher together with one server stub procedure and one service procedure for each procedure in the service interface.
- The dispatcher selects one of the server stub procedures according to the procedure identifier in the request message.
- A *server stub procedure* is like a skeleton method in that it unmarshals the arguments in the request message, calls the corresponding service procedure, and marshals the return values for the reply message.

Figure 5.10
Dealing room system



5.4 Events and notifications

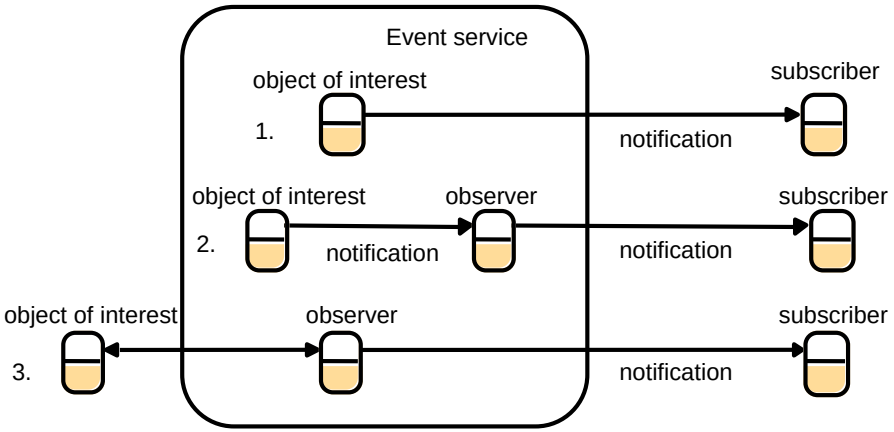
- The idea behind the use of events is that one object can react to a change occurring in another object.
- Notifications of events are essentially asynchronous and determined by their receivers.
- Distributed event-based systems extend the local event model by allowing multiple objects at different locations to be notified of events taking place at an object.

- Use the *publish-subscribe* paradigm
  - An object that generates events *publishes* the types of events that are of interest to them.
  - Different event *types* may refer to the different methods executed by the object of interest.
  - Objects that represent events are called *notifications*.
  - When a publisher experiences an event, subscribes that expressed an interest in that type of event will receive notifications.
  - Subscribing to a particular type of event is called *registering interest* in that type of event.

- Distributed event-based systems have two main characteristics
  - Heterogeneous:
    - When event notifications are used as a means of communication between distributed objects, components in a distributed system that were not designed to interoperate can be made to work together. All that is required is that event-generating objects publish the types of events they offer, and that other objects subscribe to events and provide an interface for receiving notifications.
  - Asynchronous:
    - Notification are sent asynchronously by event-generating objects to all the objects that have subscribed to them to prevent publishers needing to synchronize with subscribers.

- Event types:
  - An event source can generate events of one or more different *types*.
  - Each event has *attributes* that specify information about that event, such as the name or identifier of the object that generated it, the operations, its parameters, and the time.
  - Types and attributes are used both in subscribing to events and in notifications.

Figure 5.11
Architecture for distributed event notification



5.4.1 The participants in distributed event notification

- Figure 5.11 shows an architecture that specifies the roles played by the objects that participate in distributed event-based systems.
- The main component is an event service that maintains a database of published events and of subscribes' interests.

- The roles of the participating objects are as follows:
  - The object of interest: an object that experiences changes of state, as a result of its operations being invoked and is considered as part of the event service if it transmits notifications.
  - Event: an event occurs at an object of interest as the result of the completion of a method execution.
  - Notification: is an object that contains information about an event. Typically, it contains the type of the event and its attributes.
  - Subscriber*:* is an object that has subscribed to some type of events in another object.
  - Observer objects: to decouple an object of interest from its subscribers.
  - Publisher: an object that declares that it will generate notifications of particular types of event. A publisher may be an object of interest or an observer.

- Figure 5.11 shows three cases:
  - An object of interest inside the event service without an observer. It sends notifications directly to the subscribers.
  - An object of interest inside the event service with an observer. The object of interest sends notifications via the observer to the subscribers.
  - An object of interest outside the event service. In this case, an observer queries the object of interest in order to discover when events occur. The observer sends notifications to the subscribers.
- Delivery semantics:
  - A variety of different delivery guarantees can be provided for notifications – the chosen one should depend on the requirements of applications.

- Roles for observers
  - The task of processing notification can be divided among observer processes playing a variety of different roles. E.g.,
    - Forwarding –carry out all the work of sending notifications to subscribers on behalf of one or more objects of interest.
    - Filtering of notifications – to reduce the number of notifications received according to some predicate on the contents of each notification.
    - Patterns of events – A pattern specifies a relationship between several events.
    - Notification mailboxes – in some cases, notifications need to be delayed until a potential subscriber is ready to receive them.