DISTRIBUTED OBJECTS AND REMOTE INVOCATION

27/4/2026

From Chapters (4+5+6+8) of Distributed Systems Concepts and Design,

By G. Coulouris, J. Dollimore, T. Kindberg and G. Blair

Published by Addison Wesley/Pearson Education

Topics

- Introduction to Remote Invocation and Distributed Objects
- Request-reply protocols
- Remote Procedure Call (RPC)
- Remote method invocation (RMI)
- Event-based Distributed Programming

Introduction

- Distributed applications: applications that are composed of cooperating programs running in several different processes.
- Programs need to be able to invoke operations in other processes (running in different computers).
- Objects that can receive remote method invocations are called remote objects and they implement a remote interface.

Introduction

- This lecture is concerned with how processes (objects) communicate in distributed systems, examining:

External Data Representation is concerned with how the objects and data structures used in application programs can be translated into a form suitable for sending messages over the network, taking into account the fact that different computers may use different representations for simple data items.

Request-reply protocols represent a pattern on top of message passing and support the two way exchange of messages as encountered in client-server computing.

Introduction

Remote Procedure Call (RPC)

- Client calls a procedure implemented and executing on a remote computer
- Call as if it was a local procedure

Remote Method Invocation (RMI)

- Local object invokes methods of an object residing on a remote computer
- Invocation as if it was a local method call

Event-based Distributed Programming

- Objects receive asynchronous notifications of events happening on remote computers/processes

Introduction

- Middleware
  - Software that provides a programming model above the basic building blocks of processes and message passing is called middleware.
  - The middleware layer uses protocols based on messages between processes to provide its higher-level abstractions such as remote invocation and events.
  - Important aspect of middleware is the provision of location transparency and independence from the details of communication protocols, OS and computer HW and Programming languages.

  (Figure 1)

Introduction

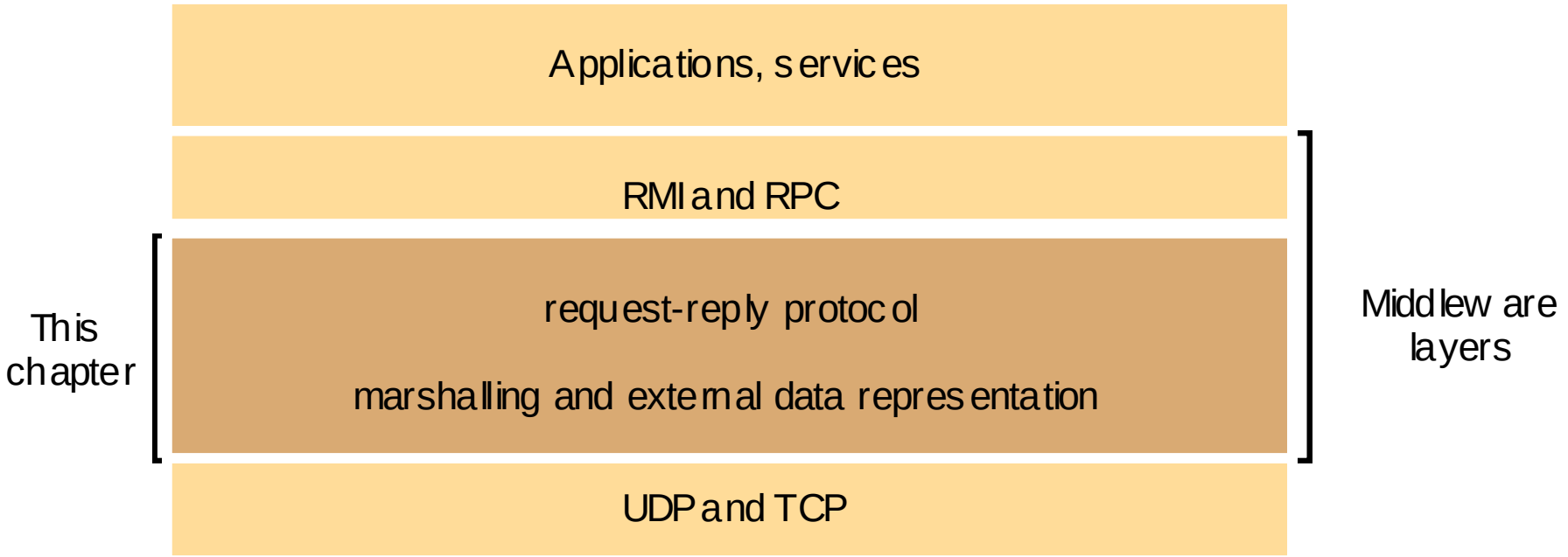


Figure 1. Middleware layers

Introduction

- Transparency Features of Middleware
  - Location transparency:
    - In RPCs and RMI, the client calls a procedure/method without knowledge of the location of invoked procedure/method.
  - Transport protocol transparency:
    - E.g., request/reply protocol used to implement RPC can use either UDP or TCP.

Introduction

- Transparency of computer hardware
  - They hide differences due to hardware architectures, such as byte ordering.
- Transparency of operating system
  - It provides independency of the underlying operating system.
- Transparency of programming language used
  - some middleware is designed to allow distributed application to use more one programming language (CORBA using an Interface Definition Languages (IDL) to define interfaces)

External Data Representation

- The information stored in running programs is represented as data structures, whereas the information in messages consists of sequences of bytes.

- Irrespective of the form of communication used, the data structure must be converted to a sequence of bytes before transmission and rebuilt on arrival.

External Data Representation

- External Data Representation is an agreed standard for the representation of data structures and primitive values.

- Data representation are:
  - Using agreed external representation, two conversions necessary
  - Using sender's format, with an indication of the format used, and convert at the recipient.

External Data Representation

- Marshalling
  - Marshalling is the process of taking a collection of data items and assembling them into a form suitable for transmission in a message.
- Unmarshalling
  - Unmarshalling is the process of disassembling a collection of data on arrival to produce an equivalent collection of data items at the destination.

External Data Representation

- Three approaches to external data representation and marshalling are:
  - CORBA's common data representation
  - Java's object serialization
  - XML

- Marshalling and unmarshalling activities is usually performed automatically by middleware layer.

External Data Representation

- **CORBA's common data representation**

which is concerned with an external representation for the structured and primitive types that can be passed as the arguments and results of remote method invocations in CORBA.

- **Java's object serialization**

which is concerned with the flattening and external data representation of any single object or tree of objects that may need to be transmitted in a message or stored on a disk. It is for use only by Java.

- **XML**

which defines a textual format for representing structured data

▪ *CORBA's representation includes just the values of the objects transmitted, and nothing about their types. On the other hand, both Java serialization and XML do include type information*

Request-reply protocols

- This form of communication is designed to support the roles and message exchanges in typical client-server interactions.
- In the normal case, request-reply communication is synchronous because the client process blocks until the reply arrives from the server.
- It is reliable because the reply from the server is effectively an acknowledgement to the client.
- Asynchronous request-reply communication is an alternative that is useful where clients can afford to retrieve replies later.

Request-reply protocols

- Often built over UDP datagrams
- Client-server protocol consists of request/response pairs, hence no acknowledgements at transport layer are necessary
- Avoidance of connection establishment overhead
- No need for flow control due to small amounts of data are transferred

Request-reply protocols

- The request-reply protocol was based on a trio of communication primitives: doOperation, getRequest, and sendReply shown in Figure 2.

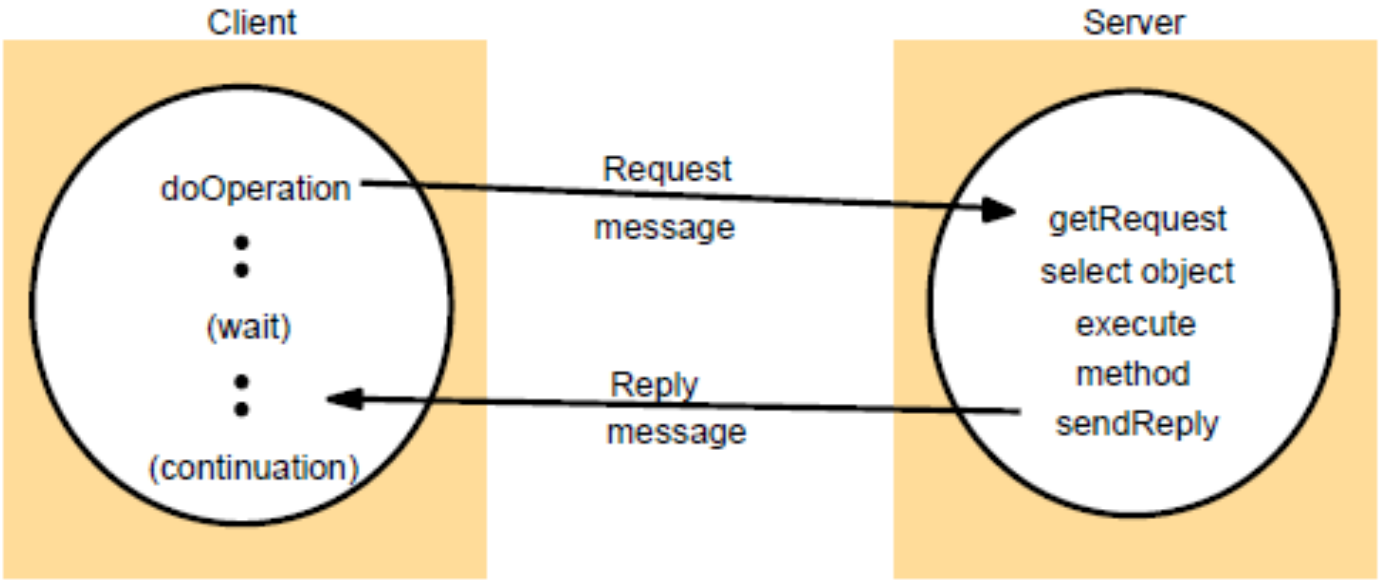


Figure 2. Request-reply communication

Request-reply protocols

- The designed request-reply protocol matches requests to replies.
- If UDP datagrams are used, the delivery guarantees must be provided by the request-reply protocol, which may use the server reply message as an acknowledgement of the client request message.
- Figure 3 outlines the three communication primitives.

Request-reply protocols

public byte[] doOperation (RemoteObjectRef o, int methodId, byte[] arguments)

sends a request message to the remote object and returns the reply.
The arguments specify the remote object, the method to be invoked and the arguments of that method.

public byte[] getRequest ();

acquires a client request via the server port.

public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);

sends the reply message *reply* to the client at its Internet address and port.

Figure 3. Operations of the request-reply protocol

Request-reply protocols

- The information to be transmitted in a request message or a reply message is shown in Figure 4.

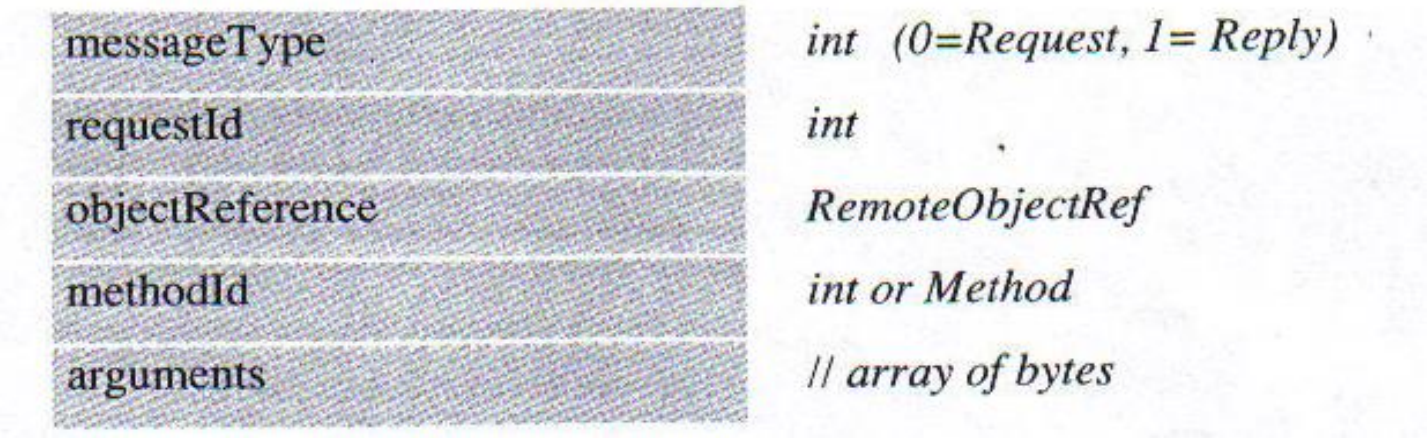


Figure 4. Request-reply message structure

Request-reply protocols

- Message identifier
  - The management of messages to provide additional properties (reliable delivery) requires that each message have a unique identifier.
  - A message identifier consists of two parts:
    - A requestId, which is taken from an increasing sequence of integers by the sending process unique to the sender
    - An identifier for the sender process, for example its port and Internet address.

      unique in the distributed system

Request-reply protocols

- Failure model of the request-reply protocol
  - If the three primitive doOperation, getRequest, and sendReply are implemented over UDP datagram, they have the same communication failures.
    - Omission failure
    - Messages are not guaranteed to be delivered in sender order.
    - Failure of processes
      - ✓ Timeouts
      - ✓ Discarding duplicate request
      - ✓ Lost reply message
      - ✓ History

Request-reply protocols

- Failure model of the request-reply protocol
  - *doOperation uses a timeout when it is waiting to get the server's reply* message.
  - The action taken when a timeout occurs depends upon the delivery guarantees being offered.
  - The simplest option is to return immediately from *doOperation* with an indication to the client that the *doOperation* has failed.
  - *doOperation sends the request message repeatedly until either it gets a reply or it is* reasonably sure that the delay is due to lack of response from the server rather than to lost messages.
  - In cases when the request message is retransmitted, the server may receive it more than once
  - This can lead to the server executing an operation more than once for the same request.
  - the protocol is designed to recognize successive messages (from the same client) with the same request identifier and to filter out duplicates.

Request-reply protocols

Failure model of the request-reply protocol

- If the server has not yet sent the reply, it need take no special action – it will transmit the reply when it has finished executing the operation.
- If the server has already sent the reply when it receives a duplicate request it will need to execute the operation again to obtain the result, unless it has stored the result of the original execution
- A server whose operations are all idempotent need not take special measures to avoid executing its operations more than once. *(an idempotent operation* is an operation that can be performed repeatedly with the same effect as if it had been performed exactly once).
- For servers that require retransmission of replies without re-execution of operations, a history may be used
- An entry in a history contains a request identifier, a message and an identifier of the client to which it was sent
- A problem associated with the use of a history is its memory cost
- the history need contain only the last reply message sent to each client
- messages in the history are normally discarded after a limited period of time.

Request-reply protocols

- Styles of exchange protocols
  - Three protocols are used for implementing various types of request behavior in the presence of communication failure.
    - The request (R) protocol.
    - The request-reply (RR) protocol.
    - The request-reply-acknowledge (RRA) protocol.
    - The messages passed in these protocols are summarized in(Figure 5)

Request-reply protocols

Name	Messages sent by		
	Client	Server	Client
R	Request		
RR	Request	Reply	
RRA	Request	Reply	Acknowledge reply

Figure 5. RPC exchange protocols

Request-reply protocols

- In the R protocol, a single request message is sent by the client to the server.
- The R protocol may be used when there is no value to be returned from the remote method.
- The RR protocol is useful for most client-server exchanges because it is based on request-reply protocol.
- RRA protocol is based on the exchange of three messages: request-reply-acknowledge reply.

Remote Procedure Call (RPC)

- A remote procedure call (RPC) is similar to a remote method invocation (RMI).
- A client program calls a procedure in another program running in a server process.
- The underlying RPC system then hides important aspects of distribution, including the encoding and decoding of parameters and results,
- RPC is generally implemented over a request-reply protocol.

Remote Procedure Call (RPC)

- Before looking at the implementation of RPC systems, we look at tow issues that are important in understanding this concept:
  - the style of programming promoted by RPC programming with interfaces;
  - the call semantics associated with RPC.

Remote Procedure Call (RPC)

- Interfaces
  - Programming languages organize a program as a set of modules that can communicate with each other.
  - Communication by means of procedure calls or direct access to the variables in another module.
  - To control interactions an explicit interface is defined for each module.
  - An Interface specifies the procedures and the variables that can be accessed and hides all implementation details.
  - Accesses the variables in a module can only occur through methods specified in interface.

Remote Procedure Call (RPC)

- Interface in distributed system
  - In a distributed program, the modules can run in separate processes.
  - In the client-server model, in particular, each server provides a set of procedures that are available for use by clients (file server).
  - The term service interface is used to refer to the specification of the procedures offered by a server
  - The definition of service interfaces is influenced by the distributed nature.
    - No direct access to remote variables is possible
    - Using message passing mechanism to transmit data and variables
      - » Request-reply protocols

Remote Procedure Call (RPC)

- Interface definition languages
  - Interface definition languages (IDLs) are designed to allow procedures implemented in different languages to invoke one another
  - An IDL provides a notation for defining interfaces in which each of the parameters of an operation may be described as for input or output in addition to having its type specified
    - ✓ Sun XDR as an example of an IDL for RPC

RPC call semantics

- In Request-reply protocols, we showed that *doOperation can be implemented in different ways to provide different* delivery guarantees. The main choices are:

  - *Retry request message: Controls whether to retransmit the request message until* either a reply is received or the server is assumed to have failed.
  - *Duplicate filtering: Controls when retransmissions are used and whether to filter out* duplicate requests at the server.
  - *Retransmission of results: Controls whether to keep a history of result messages to* enable lost results to be retransmitted without re-executing .

- Combinations of these choices lead to a variety of possible semantics for the reliability of remote invocations as seen by the invoker Figure 6.

RPC call 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*

RPC call semantics

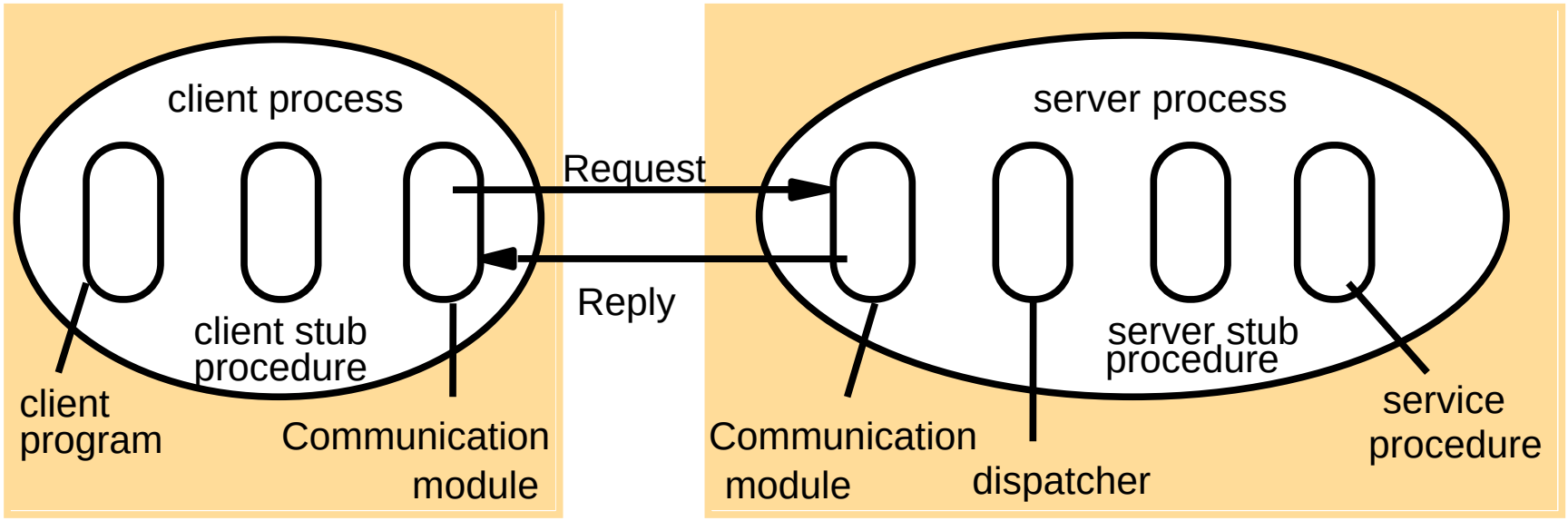
- Maybe
  - For invoker: RPC executed once, or not at all
  - arises when no fault-tolerance measures are applied
  - Suffer from: (1) message lost; (2) server crash
  - Useful for app. in which occasional failed invocation are acceptable
- At least once
  - For invoker: execute at least once, or an exception
  - Suffer from: (1) server crash; (2) arbitrary failures for non-idempotent method
- At most once
  - For invoker: receives result, or an exception
  - Prevent: omission failures by retrying, arbitrary failures

Implementation of RPC

- RPC, like RMI, may be implemented to have one of the choices of invocation semantics - at-least-once, at-most-once are generally chosen.
- The software components required to implement RPC is shown in Figure 7.

Implementation of RPC

- Service interface: the procedures that are available for remote calling
- Building blocks
  - Communication module: implement the desired design choices of invocation semantics in terms of retransmission of requests, dealing with duplicates and retransmission of results.
  - Client stub procedure (as a local procedure to the client): marshalling, sending, unmarshalling
  - Dispatcher: select one of the server stub procedures
  - Server stub procedure (as skeleton in RMI): unmarshalling, calling, marshalling
- The client and server stub procedures and the dispatcher can be generated automatically by an interface compiler from the IDL of service.



Implementation of RPC

- RPC only addresses procedure calls.
- RPC is not concerned with objects and object references.
- A client that accesses a server includes one stub procedure for each procedure in the service interface.
- A client stub procedure is similar to a proxy method of RMI (discussed later).
- A server stub procedure is similar to a skeleton method of RMI (discussed later).

Implementation of RPC

- The client that accesses a service includes one *stub procedure for each procedure in the* service interface
- The stub procedure 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.

Implementation of RPC

- 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.
- The server stub procedure then unmarshals the arguments in the request message, calls the corresponding service procedure and marshals the return values for the reply message.
- The service procedures implement the procedures in the service interface.
- Example Sun RPC

Remote method invocation (RMI)

- Remote method invocation (RMI) is closely related to RPC but extended into the world of distributed objects
- In RMI, a calling object can invoke a method in a potentially remote object.
- Discuss RMI under following headings
  - The object model
  - Distributed objects
  - The distributed object model
  - Implementation

The object model

- An object-oriented program consists of a collection of interacting objects, each of which consists of a set of data and a set of methods.
- An object communicates with other objects by invoking their methods, generally passing arguments and receiving results
- Object references
  - Objects can be accessed via object references
  - To invoke a method in an object, the object reference and method name are given, together with any necessary arguments

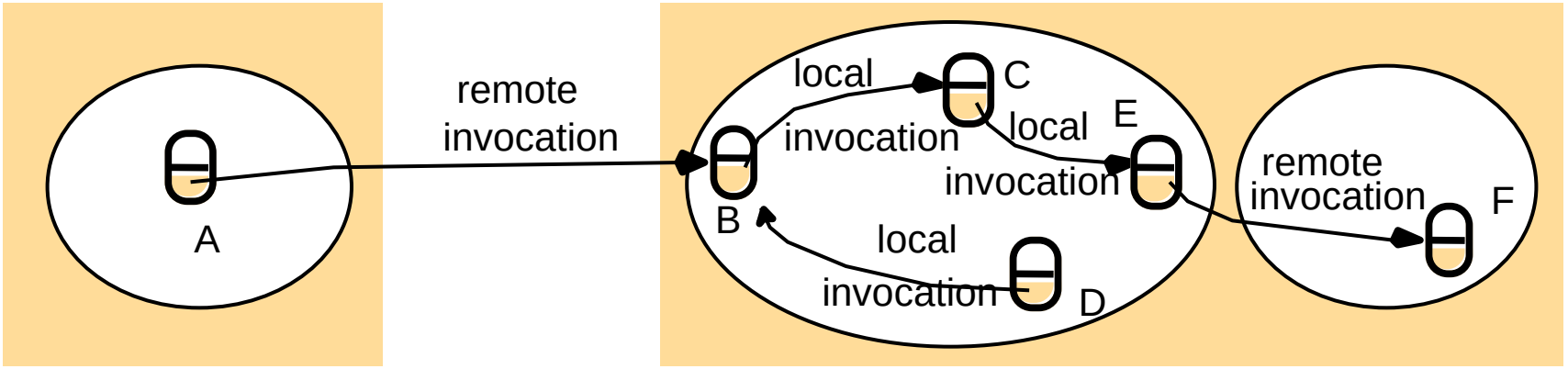
The object model

- Interfaces
  - A definition of the signatures of a set of methods
  - A class can implement several interfaces, e.g. Java
- Actions
  - Initiated by an object invoking a method in another object
  - Two affects
    - Change the state of the receiver
    - Further invocations on methods in other objects
- Garbage collection
  - Freeing the space occupied by cancelled objects
  - C++: collected by programmers
  - Java: collected by JVM

Distributed objects

- The state of an object consists of the values of its instance variables.
- In the object-based paradigm the state of a program is partitioned into separate parts, each of which is associated with an object.
- Since object-based programs are logically partitioned, the physical distribution of objects into different processes or computers in a distributed system is a natural extension

Distributed object model



- **Extensions to the object model to make it applicable to distributed objects.**
- **each process contains objects, some of which can receive both local and remote invocations, others only local invocations.**
- **Method invocations between objects in different processes are known as remote method invocations, otherwise local method invocations.**
- **objects that can receive remote invocations are called *remote objects,e.g. B , F***
- **objects need to know the *object reference* of an object in order to invoke its methods. E.g. C must have a reference to E**
- **Remote object references: objects can invoke the methods of a remote object if they have access to its remote object reference. E.g. a remote object reference for B must be available to A.**
- **R*emote interface* specifies the methods of and object that are available for invocation by other objects in other processes. E.g. B and F must have remote interfaces**

Distributed object model

- The following two fundamental concepts are at the heart of the distributed object model:
- Remote object reference
  - An unique identifier in a distributed system refer to a particular unique remote object
  - May be passed as arguments and results of remote method invocation
- Remote interface
  - remote object class 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 interface as well as other methods implemented by a remote object.

Distributed object model

- ## Actions in a distributed system
  - An action is initiated by a method invocation which may result in further invocations on methods in other objects
  - in the distributed case, the objects involved in a chain of related invocations may be located in different processes or different computers
  - RMI is used, and the remote reference of the object must be available to the invoker
  - In Figure 8 , object A needs to hold a remote object reference to object B.
  - Remote object references may be obtained as the results of remote method invocations. For example, object A in Figure 8 might obtain a remote reference to object F from object B.

Distributed object model

- Garbage collection
  - Distributed garbage collection is generally achieved by cooperation between the existing local garbage collector and an added module
  - Usually based on reference counting
- Exception
  - Any remote invocation may fail for reasons related to the invoked object being in a different process or computer from the invoker
  - notify the client and the client handle exceptions

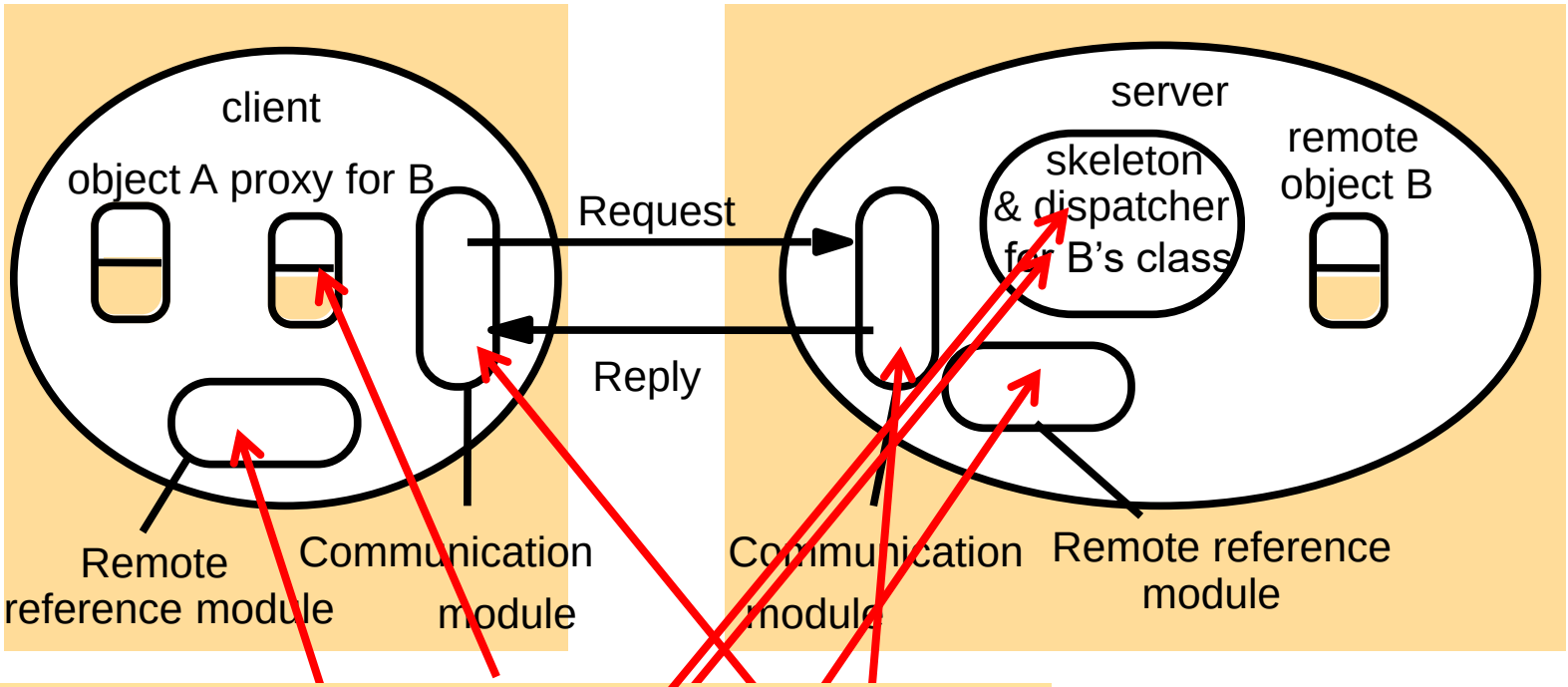
Distributed object model

Objects	*Distributed objects*	*Description of distributed object*
Object references	Remote object references	Globally unique reference for a distributed object; may be passed as a parameter.
Interfaces	Remote interfaces	Provides an abstract specification of the methods that can be invoked on the remote object; specified using an interface definition language (IDL).
Actions	Distributed actions	Initiated by a method invocation, potentially resulting in invocation chains; remote invocations use RMI.
Exceptions	Distributed exceptions	Additional exceptions generated from the distributed nature of the system, including message loss or process failure.
Garbage collection	Distributed garbage collection	Extended scheme to ensure that an object will continue to exist if at least one object reference or remote object reference exists for that object, otherwise, it should be removed. Requires a distributed garbage collection algorithm.

Implementation of RMI

- Several separate objects and modules are involved in achieving a remote method invocation.
- These are shown in Figure 10, in which an application-level object A invokes a method in a remote application-level object B for which it holds a remote object reference.
- The next slides discusses the roles of each of the components shown in that figure, dealing first with the communication and remote reference modules and then with the RMI software that runs over them.

Implementation of RMI



Proxy - makes RMI transparent to client. Class implements remote interface. Marshals requests and unmarshals results. Forwards request.

Skeleton - implements methods in remote interface. Unmarshals requests and marshals results. Invokes method in remote object.

RMI software - between application level objects and communication and remote reference modules

Implementation of RMI

- Communication module
  - Request/reply between client and server
  - The communication modules are together responsible for providing a specified invocation semantics, for example at-most-once.
  - The communication module in the server select dispatcher at server side
- Remote reference module
  - Translate between local and remote object reference
  - Create remote object reference
  - *Remote object table*

Implementation of RMI

- *Remote object table*
  - ❖ entries for remote objects held by the process, in the Figure the remote object B will be recorded in the table at the server.
  - ❖ entries for local proxies, in the Figure the proxy for B will be recorded in the table at the client.
- When a remote object is to be passed as an argument or a 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.
  - ➢ In the case that the remote object reference is not in the table, the RMI software creates a new proxy and asks the remote reference module to add it to the table.
- This module is called by components of the RMI software when they are marshalling and unmarshalling remote object references.

Implementation of RMI – RMI software

- Proxy
  - The role of a proxy is to make remote method invocation transparent to clients by behaving like a local object to the invoker;
  - but instead of executing an invocation, it forwards it in a message to a remote object (It hides the details of the remote object reference, the marshalling of arguments, unmarshalling of results and sending and receiving of messages from the client).
  - There is one proxy for each remote object for which a process holds a remote object reference.
  - The class of a proxy implements the methods in the remote interface of the remote object it represents,
  - The proxy implements them quite differently. Each method of the proxy marshals a reference to the target object, its own *operationId* and its arguments into a *request* message and sends it to the target.
  - It then awaits the *reply* message, unmarshals it and returns the results to the invoker.

Implementation of RMI – RMI software

- Skeleton
  - The class of a remote object has a *skeleton*
  - implement the method in the remote interface
  - They are implemented quite differently,
  - A skeleton method unmarshals the arguments in the *request* message and
  - invokes the corresponding method in the remote object.
  - It waits for the invocation to complete and
  - then marshals the result in the reply message

Implementation of RMI – RMI software

- Dispatcher
  - A server has one dispatcher and one skeleton for each class representing a remote object
  - The dispatcher receives request messages from the communication module.
  - It uses the operationId to select the appropriate method in the skeleton, passing on the request message.
  - The dispatcher and the proxy use the same allocation of operationIds to the methods of the remote interface.

Implementation of RMI - execution

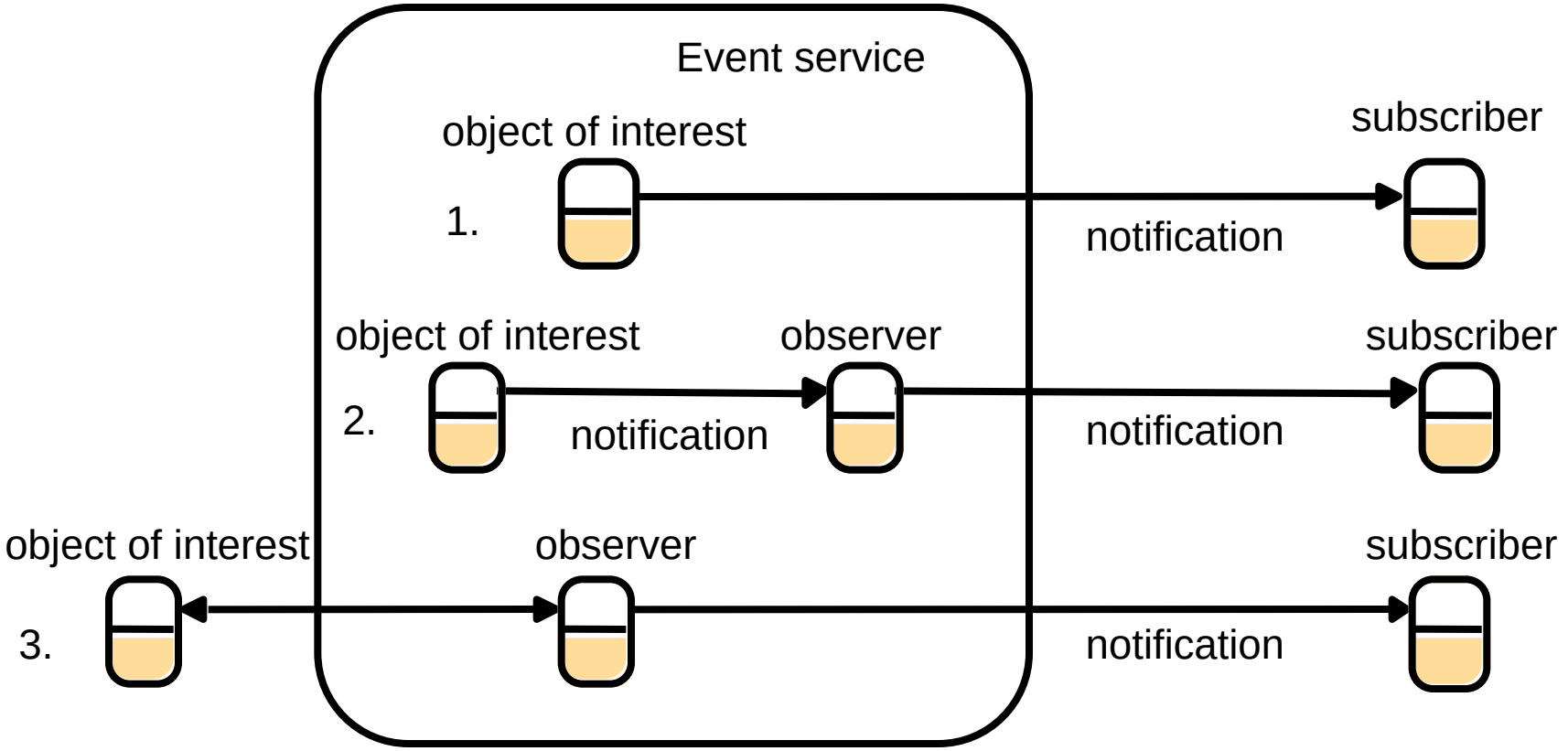
- The classes for proxies, dispatchers and skeletons
  - generated automatically by an interface compiler, e.g. rmic
- Server program
  - create and initialize at least one of the remote objects
  - Register remote objects by name
- Client program
  - look up the remote object references
  - invoke
- The binder
  - A service that maintain a table containing mapping information of textual names to remote object references

Event-based Distributed Programming

- Idea
  - one object react to a change occurring in another object
- Event examples
  - modification of a document
  - an electronically tagged book being at a new location
- Publish/subscribe paradigm
  - event generator publish the type of events
  - event receiver subscribe to the types of events that are interest to them
  - When event occur, notify the receiver
- Distributed event-based system – two characteristics
  - Heterogeneous: components in a DS that were not designed to interoperate can be made to work together
  - Asynchronous: prevent publishers needing to synchronize with subscribers

Architecture for Event-based Distributed Programming

- Event service: maintain a database of published events and of subscribers' interests
- decouple the publishers from the subscribers



The roles of the participating objects

- The object of interest
  - its changes of state might be of interest to other objects
- Event
  - An event occurs at an object of interest as the completion of a method execution
- Notification
  - an object that contains information about an event
- Subscriber
  - an object that has subscribed to some type of events in another object
- Observer objects
  - the main purpose is to decouple an object of interest from its subscribers.
  - Avoid over-complicating the object of interest.
- Publisher
  - an object that declares that it will generate notifications of particular types of event. May be an object of interest or an observer.

Notification delivery

- Delivery semantics
  - Unreliable, e.g. deliver the latest state of a player in a Internet game
  - Reliable, e.g. dealing room
  - real-time, e.g. a nuclear power station or a hospital patient monitor
- Roles for observers
  - Forwarding
    - send notifications to subscribers on behalf of one or more objects of interests
  - Filtering of notifications according to some predicate
  - Notification mailboxes
    - notification be delayed until subscriber being ready to receive