Session 8
Inter Process Communication-2

Objectives

At the end of the session, the student will be able to

- Explain the RMI technique of communication between objects
- Discuss the semantics and implementation aspects of RMI
- Describe RPC mechanism

Contents

- Communication between distributed objects: RMI
- Remote Procedure call: RPC

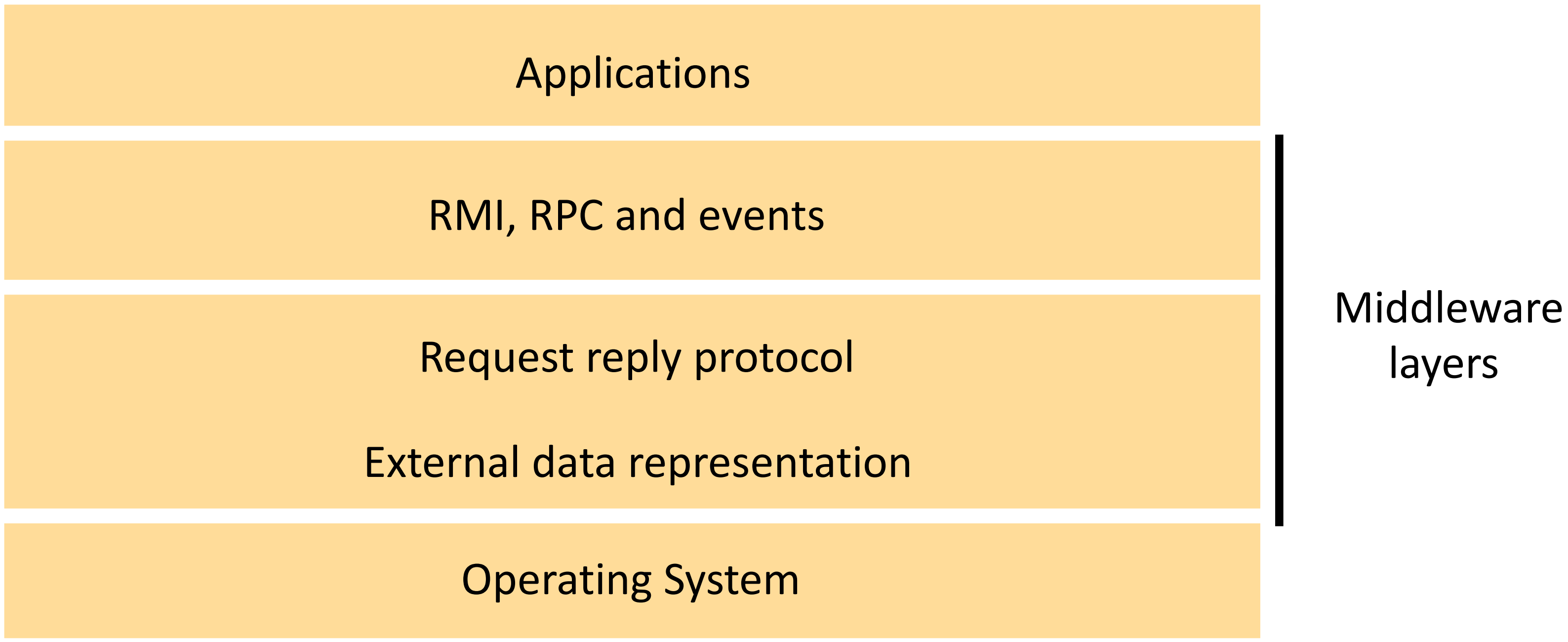
Distributed Objects and Remote Invocation

Introduction

- The communication between distributed objects is by means of ***Remote Method Invocation.***
- ***Remote Procedure Calls (RPC)*** extend the capabilities of conventional procedure calls across a network and are essential in the development of distributed systems.
- ***Events and notifications*** provide a way for heterogeneous objects to communicate with one another asynchronously.

Middleware layers

- An important aspect of middleware is the provision of location transparency and independence from the details of communication protocol, OS and computer hardware.



Middleware Layers

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, possibly on a different computer. Nor does the client need to know the location of the server
- In RMI, the object making invocation can not tell whether the object it invokes is local or not and does not need to know its location
- 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 locations

Middleware Layers

- **Communication Protocols**: The protocols that support middleware abstractions are independent of the underlying transport protocols. For example the request-reply protocol can be implemented over either UDP or TCP
- **Computer hardware:** Agreed standards for external data representation are used when marshalling and unmarshalling the messages. They hide the differences due to hardware abstractions, such as byte ordering
- **Operating Systems:** The higher level abstractions provided by the middle layer are independent of the underlying operating systems

Middleware Layers

- **Use of several programming languages:** Some middleware is designed to allow distributed applications to use more than one programming language
- In particular, CORBA allows clients written in one language to invoke methods in objects that live in server programs written in another language
- This is achieved by using an *Interface Definition Language* (IDL) to define interfaces

Interfaces

- The interface of a module specifies the procedures and the variables that can be accessed from other modules
- Interface of a module for RPC or RMI cannot specify direct access to variables
- CORBA IDL interfaces can specify attributes by means of some getter and setter procedures added automatically to the interface

Interface Definition Language

- IDLs are designed to allow objects implemented in different languages to invoke one another
- An IDL provides a notation for defining interfaces in which each of the parameters of a method may be described as for input or output in addition to having its type specified
- CORBA IDL is an example of an IDL for RMI and Sun XDR is an example of an IDL for RPC

Communication Between Distributed Objects

- The communication between distributed objects takes place by means of RMI
- Communication is described by the following
  - The Object Model
  - Distributed Objects
  - The Distributed Object Model
- Issues
  - Design for RMI
  - Implementation of RMI
  - Distributed Garbage Collection

The Object Model

- An object communicates with other objects by invoking their methods, generally passing arguments and receiving results
- Objects can encapsulate their data and the code of their methods
- Objects can be accessed via **object references**
- In java a variable that appears to hold an object actually holds a reference to that object
- To invoke a method in an object, the object reference and the method name are given, together with any necessary arguments
- Object references are first class values, that may be assigned to variables, passed as arguments and returned as results of methods

The Object Model, Cont’d.

- An **interface** provides a definition of the signatures of a set of methods without specifying their implementation
- It also defines types that can be used to declare the type of the variables or of the parameters and return values of methods.

- **Action** in OO program is initiated by an object invoking a method in another 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 place.
- An invocation can lead to further invocations of methods in other objects.

The Object Model, Cont'd.

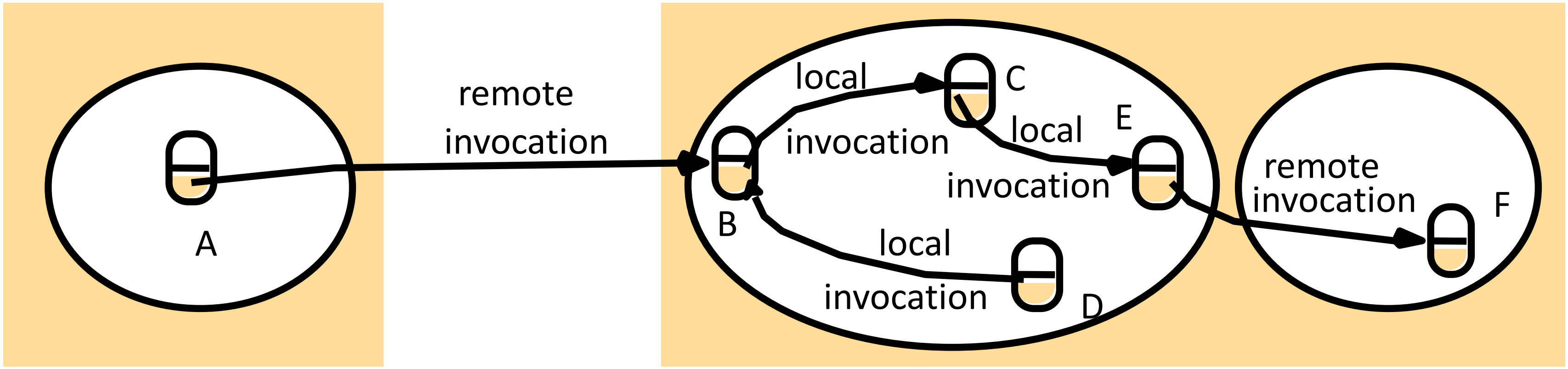
- ***Exceptions*** provide a clean way to deal with error conditions without complicating the code
- Each method heading explicitly lists (throw) as exceptions the error conditions it might encounter, allowing users of the method to deal with them
- This means that control passes to another block of code that *catches* the exception

- ***Garbage Collection*** provide a means of freeing the space occupied by objects when they are no longer needed

Distributed Object Model

- Each process contains a collection of objects, some of which can receive both local and remote invocations, whereas the other objects can receive only local invocations
- Method invocations between objects in different processes, whether in the same computer or not, are known as ***remote method invocations***
- Method invocations between objects in the same process are ***local method invocations***

Remote and Local Method Invocations



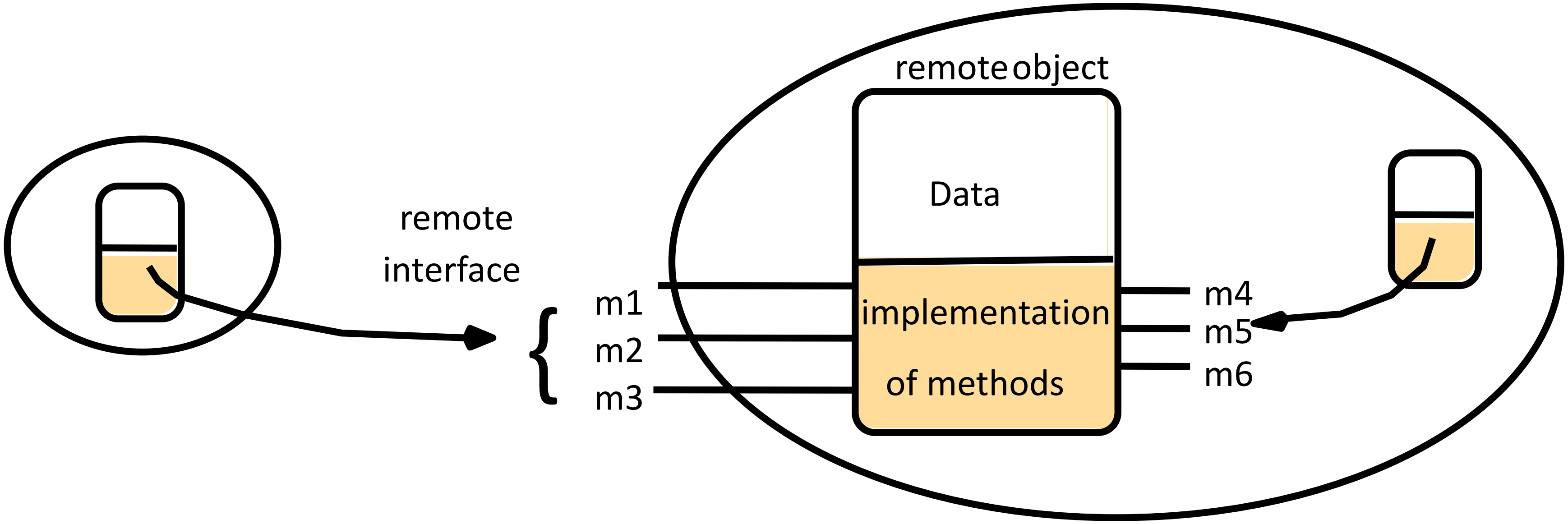
Distributed Object Model

- Other objects can invoke the methods of a remote object if they have access to its ***remote object reference***
    - For example, a remote object reference for B in Fig. must be available to A
    - The remote object to receive a remote method invocation is specified by the invoker as a remote object reference
    - Remote object references may be passed as arguments and results of remote method invocations
- Every remote object has a ***remote interface*** that specifies which of its methods can be invoked remotely.
    - For example, the objects B and F must have remote interfaces

Distributed Object Model

- The class of a remote object implements 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 an object

A Remote Object and its Remote Interface



Distributed Object Model

- As in the non-distributed case, an ***action*** is initiated by a method invocation,
    - 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
- When an invocation crosses the boundary of a process or a computer, RMI is used
    - The remote reference of the object must be available to the invoker
- For example object A might obtain a remote reference to object F from Object B

Distributed Object Model

- Distributed ***garbage collection*** is achieved by cooperation between the existing local garbage collector and an added module that carries out a form of distributed garbage collection, usually based on *reference counting*
- The process containing the remote object may have crashed or may be too busy to reply, or the invocation or result message may be lost
- Therefore, remote method invocation should be able to raise ***exceptions*** such as timeouts that are due to distribution as well as those raised during the execution of the method invoked

Design Issues for RMI

1. The choice of invocation Semantics
   - Although local invocations are executed exactly once, this cannot always be the case for RMI

2. The level of transparency that is desirable for RMI
   - The syntax of a remote invocation is the same as that of a local invocation
   - But the difference between local and remote objects should be expressed in their interfaces

RMI Invocation Semantics

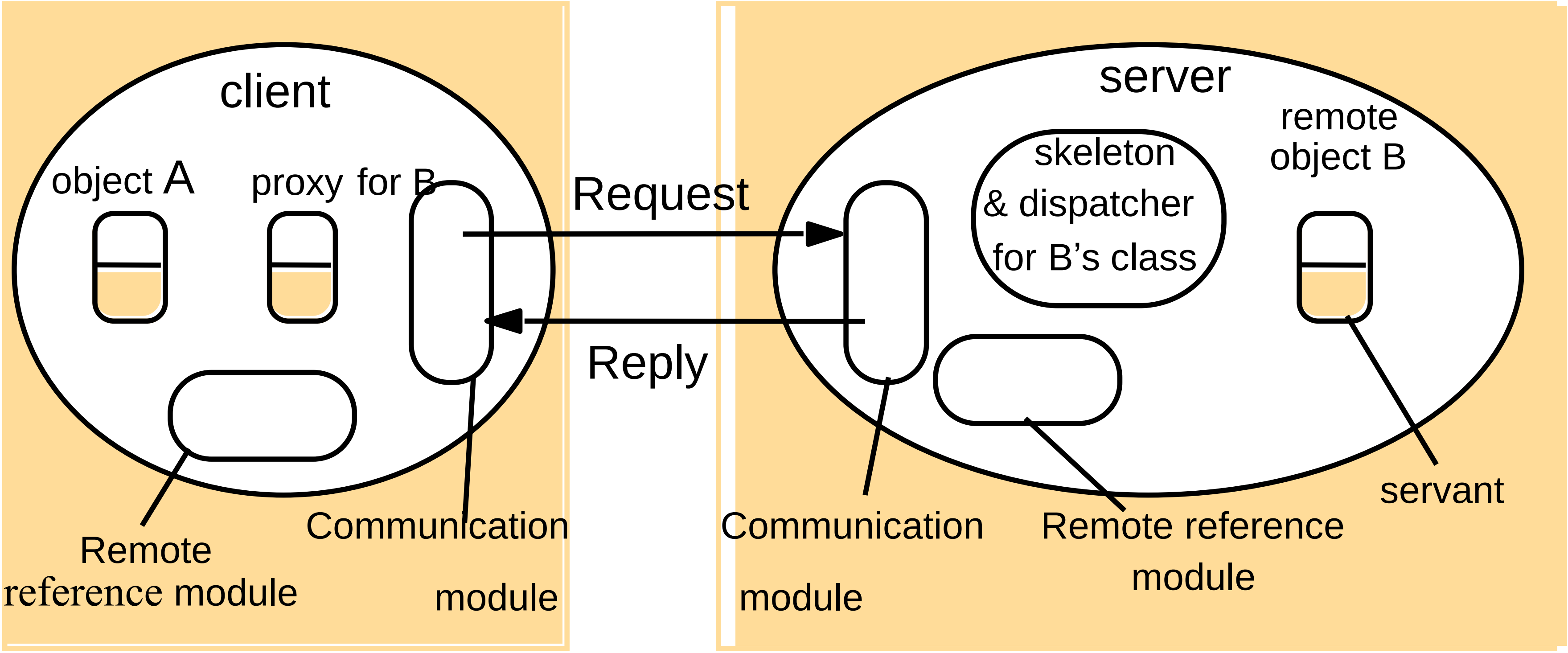
The choice of RMI invocation semantics are defined as

- **Maybe Invocation Semantics**
  - The remote method may be executed once or not at all
  - Can suffer from Omission failures, Crash failures
  - E.g., CORBA
- **At-least-once Invocation semantics**
  - The invoker receives either a result, in which case the invoker knows that the method was executed at least once, or an exception informing it that no result was received
  - Can suffer from Crash failures, Arbitrary failures
  - E.g., Sun RPC
- **At-most-once Invocation Semantics**
  - The invoker receives either a result, in which case the invoker knows that the method was executed exactly once, or an exception informing it that no result was received
  - E.g., Java RMI and CORBA

Implementation of RMI

- Several separate objects and modules are involved in achieving a Remote Method Invocation
  - Communication module
  - Remote reference module
  - Servant

Role of Proxy and Skeleton in RMI



An application-level object A invokes a method in a remote application-level object B for which it holds a remote object reference

Implementation of RMI

- ***Communication module***
  - The two cooperating communication modules carry out the request reply protocol, which transmits request and reply messages between client and the server
  - The communication module uses only the first three items, which specify the message type, its requestId and the remote reference of the object to be invoked
  - The communication modules are together responsible for providing a specified invocation semantics
  - The communication module in the server selects the *dispatcher* for the class of object to be invoked

Implementation of RMI

- ***Remote reference module***
  - ▪Responsible for translating between local and remote object references and for creating remote object references
  - ▪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 which are system wide
  - ▪The table includes
    - •An entry for each of the remote objects held by the process
      - •The remote object B will be recorded in the table at the server
    - •An entry for each local proxy
      - •The proxy for B will be recorded in the table at the client

Implementation of RMI

- The ***actions*** of the Remote reference module are
  - 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 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

Implementation of RMI

- ***Servant***
  - ▪A servant is an instance of a class which provides the body of a remote object
  - ▪It is the servant that eventually handles the remote requests passed on by the corresponding skeleton
  - ▪Servants live within a server process
  - ▪They are created when remote objects are instantiated and remain in use until they are no longer needed, finally being garbage collected or deleted

Implementation of RMI

- ***RMI software***
  - ▪This consists of a layer of software between the application level objects and the communication & remote reference modules
  - ▪The roles of the middleware objects are
    - •Proxy
    - •Dispatcher
    - •Skeleton
  - •The classes for the proxy, dispatcher and skeleton used in the RMI are generated automatically by an interface compiler
  - •The Java RMI compiler generates these from the class of the remote object

The 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 RMI Software

- ***Dispatcher***
  - A server has one dispatcher and skeleton for each class representing a remote object
  - Dispatcher receives the request message from the communication module
  - Dispatcher and proxy use the same allocation of *methodIds* to the methods of the remote interface
- ***Skeleton***
  - The class of a remote object has a skeleton, which implements the methods in remote interface
  - A skeleton method unmarshals the arguments in the *request* message and invokes the corresponding method in the servant

RMI Software

- ***Dynamic invocation***
  - An alternative to proxies
  - The proxy is static, its class is generated from an interface definition and then compiled in to the client code
  - But when a client program receives a remote interface to an object whose remote interface was not available at compile time then it needs ***dynamic invocation***
  - It is not so convenient to use as a proxy, but is useful in applications where not all interfaces of the remote objects to used can be predicted at design time

Distributed Garbage Collection

- The aim is 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
- Each server process maintains a set of the names of the processes that hold remote object references for each of its remote objects
- E.g., *B.holders* is the set of client processes that have proxies for object B

Distributed Garbage Collection

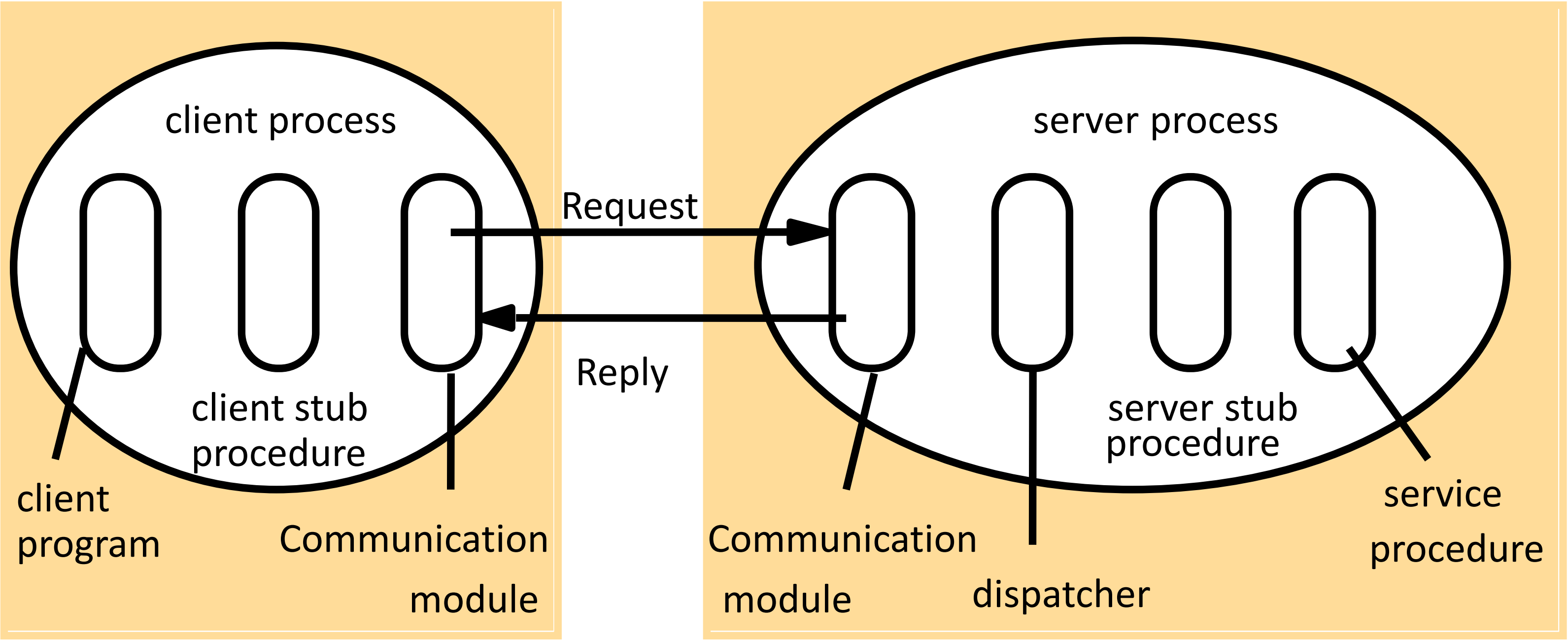
- When a client C first receives a remote reference to a particular remote object B, it makes an *addRef(B)* invocation to the server of that remote object and then creates a proxy, the server adds C to *B.holders.*
- When a client C's garbage collector notices that a proxy for remote object B is no longer reachable, it makes a *removeRef(B)* invocation to the corresponding server and then deletes the proxy, the server removes C from B.holders
- When B.holders is empty, the servers local garbage collector will reclaim the space occupied by B unless there are any local holders

RPC

Remote Procedure Call

- A remote procedure call is very similar to a remote method invocation in that a client program calls a procedure in another program running in server process
  - •Except that no remote reference modules are required, since procedure call is not concerned with objects and object references
- Servers may be clients of other servers to allow chains of RPC’s
- RPC, like RMI, may be implemented to have one of the choices of invocation semantics.
- RPC is generally implemented over a request-reply protocol which is simplified by the omission of object references from request messages

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



Remote Procedure Call

- The client that accesses a service includes one stub procedure for each procedure in the service interface
- The role of a stub procedure is similar to that of a proxy method
- 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

Remote Procedure Call

- A server stub procedure is like a skeleton method in that it marshals the arguments in the request message
- It calls the corresponding service procedure and marshals the return values for the reply message
- The service procedures implement the procedures in the service interface
- The client and server stub procedures and the dispatcher can be generated by an interface compiler from the interface definition of the service

Case Study: Sun RPC

- Sun RPC was designed for client server communication in the Sun NFS network file system
- Implementers have the choice of using remote procedure calls over either UDP or TCP
- It uses at-least-once call semantics
- The Sun RPC system provides an interface language called XDR and an interface compiler called *rpcgen* which is intended for use with the C programming language

Summary

In this session

- We have discussed
  - Distributed Object communication using RMI
  - Remote Procedure Call (RPC) mechanism for distributed communication
- The semantics and implementation of RMI

Questions

Thank you