Session 7

Inter-Process Communication - 1

Course Leader: Jishmi Jos Choondal

Objectives

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

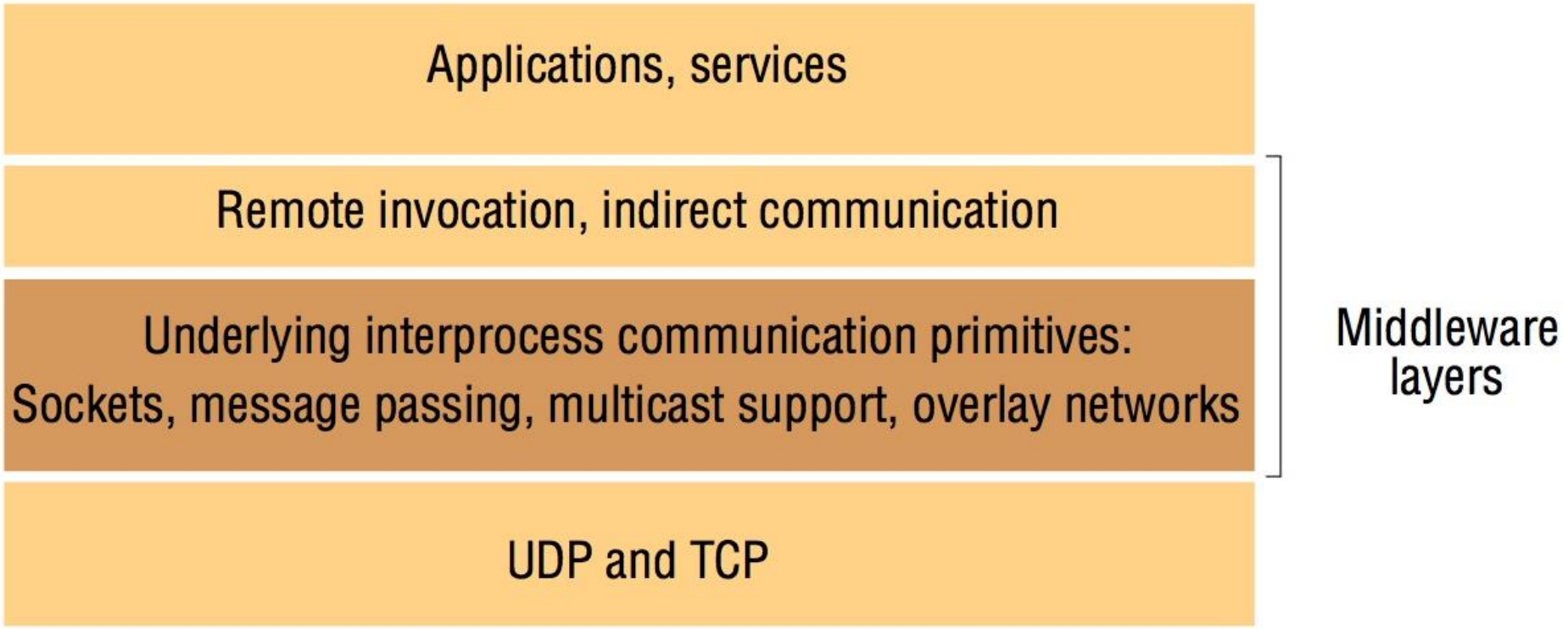
- Explain external data representation and marshalling
- Discuss protocol support for Client-Server and Group Communications

Contents

- External data representation and marshalling
- Client-server communication
- Group communication

Inter-Process Communication

Middleware Layers



Characteristics of interprocess communication

- The message passing between a pair of processes can be supported by send and receive
- Communication of data from the sending process to the receiving process may involve the synchronization of the two processes
- Synchronous Communication:In synchronous communication send and receive process synchronize at each message
  - Whenever a send is issued the sending process or thread is blocked until the corresponding receive is isssued
  - Whenever a receive is issued the process blocks until a message arrives
- Asynchronous communication: the use of send operation is non-blocking and the receive operation can have blocking and non blocking variants

External Data Representation and Marshalling

External Data Representation and Marshalling

- External Data Representation
  - It is an agreed format to support RMI or RPC for the representation of data structures and primitive values
- Remote Method Invocation
  - Allows an object to invoke a method in an object in a remote process
  - Examples of systems for remote invocation are CORBA and Java RMI
- Remote Procedure Call
  - Allows a client to call a procedure in a remote server

External Data Representation and Marshalling

- Marshalling
  - Process of taking a collection of data items and assembling them into a form suitable for transmission in a message
- Unmarshalling
  - Process of diassembling them on arrival to produce an equivalent collection of data items in the destination
- Alternative approaches to external data representation and marshalling include
  - CORBA's Common Data Representation (CDR)
  - Java's object serialization
  - XML or Extensible Markup Language

CORBA

Common Object Request Broker Architecture: CORBA defines a range of standards permitting interoperation between complex object oriented systems potentially built by diverse vendors

CORBA enables communication between software written in different languages and running on different computers.

Implementation details from specific operating systems, programming languages, and hardware platforms are all removed from the responsibility of developers who use CORBA.

CORBA normalizes the method-call semantics between application objects residing either in the same address-space (application) or in remote address-spaces (same host, or remote host on a network)

CORBA

CORBA is a standard (not a product!)

- Allows objects to transparently make requests and receive responses
- Enables interoperability between different applications
  - on different machines
  - in heterogeneously distributed environments
- CORBA is a way of *talking about solutions but* not a *specific set of prebuilt solutions*
  - CORBA worries about syntax but not semantics

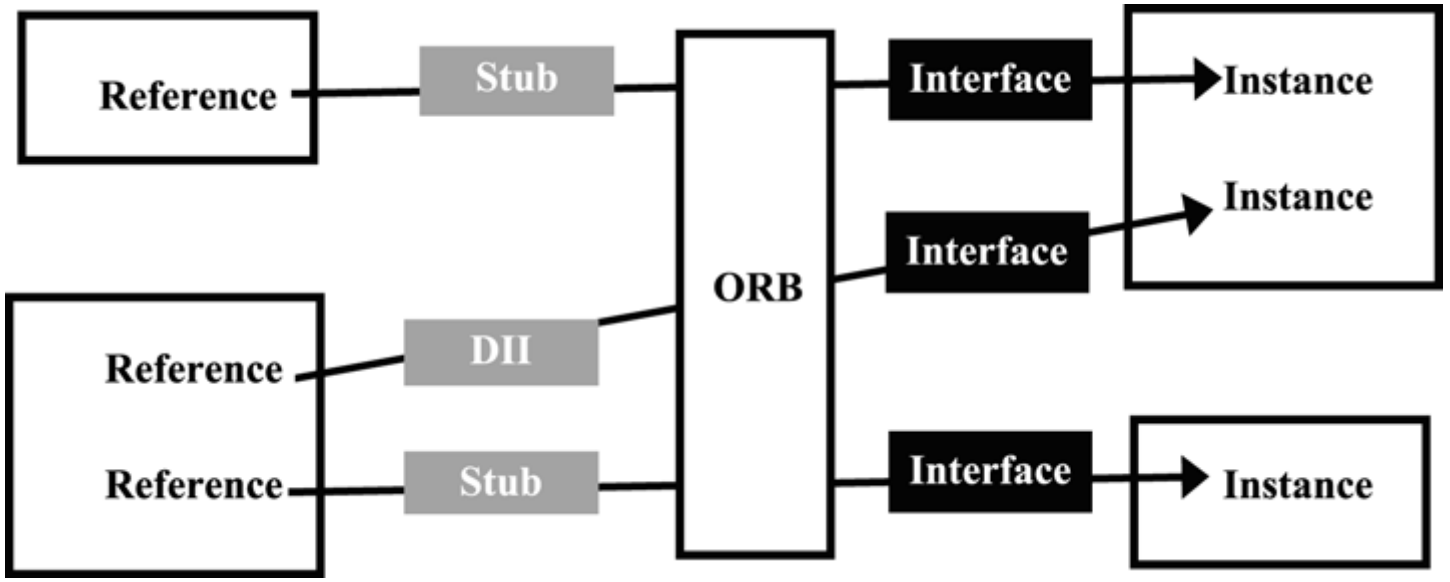
The CORBA Reference Model

- The key to understanding the structure of a CORBA environment is the *Reference Model*
  - which consists of a set of components that a CORBA platform should typically provide.

- A CORBA implementation must supply an *Object Request Broker,* or ORB.

- There are two cases of invocations:
  - the static one (stub) and
  - the Dynamic Invocation Interface (DII), which is more complex to use

The CORBA Reference Model



ORB–Object Request Broker

What is an ORB?

- Highway over which all CORBA-communication occurs

The ORB is responsible for

- Data marshaling
- Object location management
- Delivering request to objects
  - Returning output values back to client

CORBA is Location Transparent

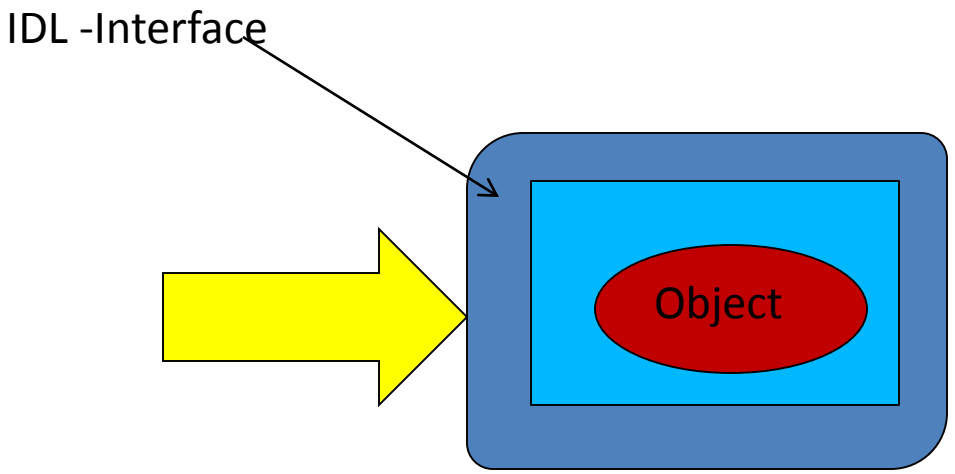
For the client it doesn't matter if the object it is operation on is running

- in different process on another processor
- on the same processor, in a different process
- on the same processor and even in the same process

CORBA works with IDL

- CORBA uses an interface definition language (IDL) to specify the interfaces that objects present to the outer world.
- CORBA then specifies a *mapping* from IDL to a specific implementation language like C++ or Java
- Standard mappings exist for Ada, C, C++, C++11, COBOL, Java, Lisp, PL/I, Object Pascal, Python, Ruby and Smalltalk.
- Non-standard mappings exist for C#, Erlang, Perl, Tcl and Visual Basic implemented by object request brokers (ORBs) written for those languages
- The CORBA specification dictates there shall be an ORB through which an application would interact with other objects

CORBA works with interfaces IDL



- All CORBA Objects are encapsulated
- Objects are accessible through interface only
- Separation of interfaces and implementation enables multiple implementations for one interface

IDL-Interface Definition Language

IDL

- Specifies the Interfaces
- Is programming-language independent and only contain data descriptions

IDL-Compiler translates IDL-specification in to

- *IDLstubs (client-side)*
- *IDL-skeletons (server-side)* in the desired programming language

CORBA ORB Interfaces

- ORB interface: contains functionality that might be required by clients or servers

- Dynamic Invocation Interface (DII): used for dynamically invoking CORBA objects that were not known at implementation time to servers

- Interface Repository: Registry full of qualified interface definitions, provides type information necessary to issue requests using DII

CORBA's Common Data Representation (CDR)

- CORBA CDR is the external data representation defined with CORBA 2.0
- CDR can represent all of the data types that can be used as arguments and return values in remote invocations in CORBA
- These consistes of 15 primitive types and a range of composite types
- Each argument or result in a remote invocation is represented by a sequence of bytes in the invocation or result message

Java Object Serialization

- Serialization:
  - Flattens objects into an ordered, or serialized stream of bytes
  - The ordered stream of bytes can then be read at a later time, or in another environment, to recreate the original objects
- Java object serialization is used to persist Java objects to a file, database, network, process or any other system
- Deserialization: Restoring the state of an object or a set of objects from their serialized form

Java Object Serialization

•Java provides classes to support writing objects to streams and restoring objects from streams.

•Only objects that support the java.io.**Serializable** interface or the java.io.**Externalizable** interface can be written to streams.

•public interface Serializable
  - •The Serializable interface has no methods or fields
  - •Only objects of classes that implement java.io.Serializable interface can be serialized or deserialized

Transient Fields and Java Serialization

•The *transient* keyword is a modifier applied to instance variables in a class

•It specifies that the variable is not part of the persistent state of the object and thus never saved during serialization

•We can use the transient keyword to describe temporary variables, or variables that contain local information, such as a process ID or a time lapse

Java Object Serialization

```
import java.io.*;
public class SerializationDemo {
public static void main(String args[]) {
        // Object serialization
        try {
                MyClass object1 = new MyClass("Hello", -7, 2.7e10);
                System.out.println("object1: " + object1);
                FileOutputStream fos = new
FileOutputStream("serial");
                ObjectOutputStream oos =
                    new  ObjectOutputStream(fos);
                oos.writeObject(object1);
                oos.flush();
                oos.close();
        } catch(Exception e) {
                System.out.println("Exception during serialization: "
+ e);
                System.exit(0);
        }
// continues …
```

Java Object Serialization

```
// Object deserialization
        try {
                MyClass object2;
                FileInputStream fis = new FileInputStream("serial");
                ObjectInputStream ois = new ObjectInputStream(fis);
                object2 = (MyClass)ois.readObject();
                ois.close();
                System.out.println("object2: " + object2);
        }
        catch(Exception e) {
                System.out.println("Exception during deserialization: " + e);
                System.exit(0);
        }
}
}
```

Java Object Serialization

```
class MyClass implements Serializable {
String s;
int i;
double d;
public MyClass(String s, int i, double d) {
        this.s = s;
        this.i = i;
        this.d = d;
        }
public String toString() {
        return "s=" + s + "; i=" + i + "; d=" + d;
        }
}
```

Result:

object1: s="Hello"; i=-7; d=2.7E10
object2: s="Hello"; i=-7; d=2.7E10

Extensible Markup Language

- It is a markup language that was defined by the World Wide Web Consortium (W3C) for general use on the web

- Both HTML and XML were derived from SGML (Standardized Generalized Markup Language), a very complex markup language

XML

- HTML was designed for defining the appearence of the webpages
- XML was designed for writing structured documents for the web
- XML data items are tagged with ’markup’ strings.
  - To describe the logical structure of the data and
  - To associate attribute value pairs with logical structures
- XML is used
  - To enable clients to communicate with web services and
  - For defining interfaces and other properties of web services

XML

- XML is extensible: users can define their own tags
  - In contrast to HTML which uses a fixed set of tags
- XML definition of person Structure

```
<person id="123456789">
        <name>Smith</name>
        <place>London</place>
        <year>1934</year>
        <!-- a comment -->
</person >
```

- Use of a *namespace*

```
<person pers:id="123456789" xmlns:pers = "http://www.cdk5.net/person">
        <pers:name> Smith </pers:name>
        <pers:place> London </pers:place >
        <pers:year> 1984 </pers:year>
</person>
```

XML Schema

```
<xsd:schema  xmlns:xsd = URL of XML schema definitions >
        <xsd:element name= "person" type ="personType" />
                <xsd:complexType name="personType">
                        <xsd:sequence>
                                <xsd:element name = "name"  type="xs:string"/>
                                <xsd:element name = "place"  type="xs:string"/>
                                <xsd:element name = "year"  type="xs:positiveInteger"/>
                        </xsd:sequence>
                        <xsd:attribute name= "id"   type = "xs:positiveInteger"/>
                </xsd:complexType>
</xsd:schema>
```

Distributed Communication

Distributed Communication

- The following are the most commonly used communication pattern in distributed programs
  - Client Server Communication
  - Group Communication
- In Client Server Communication request and reply messages provide the basics for RMI and RPC
- In Group Communication the same message is sent to several processes

Client Server Communication

- Designed to support the roles and message exchanges in typical client server interactions
- In normal case request-reply communication is **synchronous** because the client process blocks until the reply arrives from the server
- **Asynchronous** request-reply communication is an alternative that may be useful in situations where clients can afford to retrieve replies later

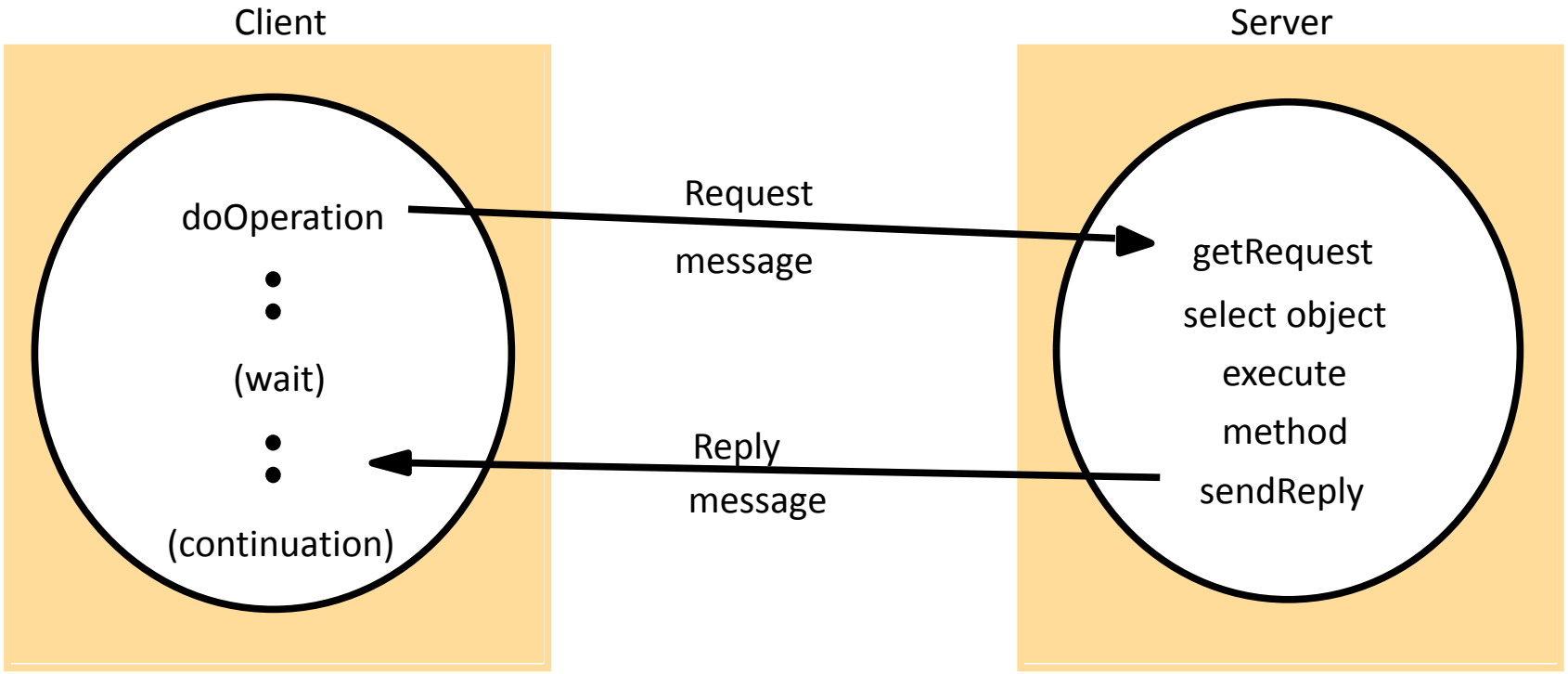
The Request-Reply Protocol

- It is based on a trio of communication primitives:

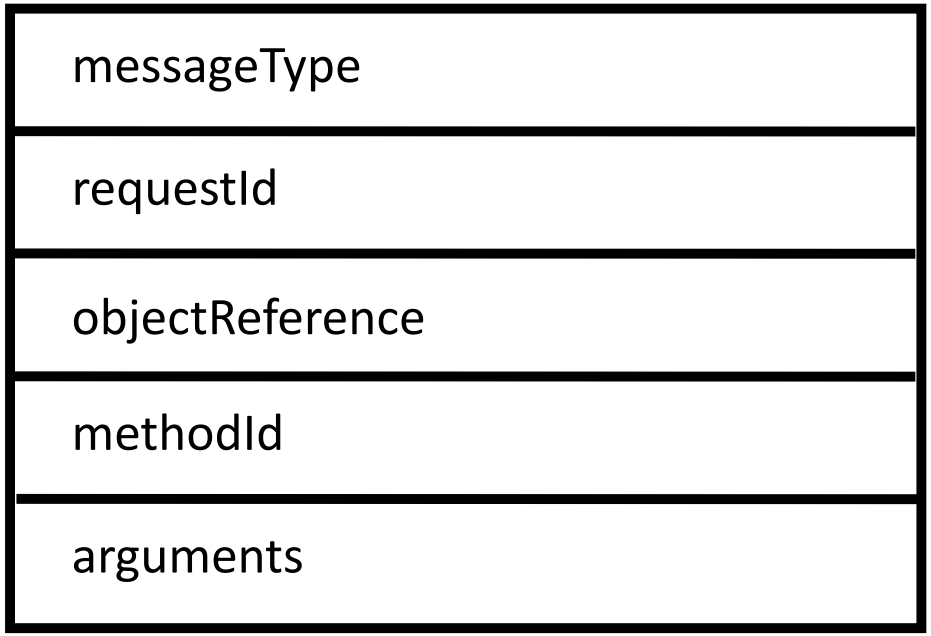
  doOperation, getRequest and sendReply

- The *doOperation* method is used by clients to invoke remote operations
- *GetRequest* is used by a server process to acquire service requests
- When the server has invoked the method in the specified object it uses *sendReply* message to send the result to the client
- When the reply message is received by the client the original *doOperation* is unblocked and execution of the client program continues

Request-Reply Communication



Request-Reply Message Structure



The Request-Reply Protocol

- HTTP is an example of a request-reply protocol
- HTTP used by a web browser clients to make requests to web servers and to receive replies from them
- Webservers manage resources implemented in different ways such as data and as a program
- Client requests specify a URL that includes the DNS host name port number of a webserver as well as the ID of a resource on that server
- HTTP protocol that specifies the messages involved in a request-reply exchange, the methods, arguments and results, and the rules for representing them in the messages

Group Communication

- A multicast operation: An operation that sends a single message from one process to each of the members of a group of processes
  - Usually in such a way that the membership of the group is transparent to the sender
- Multicast messages provide a useful infrastructure for constructing distributed systems
  - Fault tolerance based on replicated services-group of servers
  - Finding the discovery servers in spontaneous networking
  - Better performance through replicated data
  - Propagation of events notification

IP Multicast-An Implemetation of Group Communication

- IP multicast is built on top of the Internet Protocol (IP)
- IP multicast allows the sender to transmit a single IP packet to a set of computers that form a multicast group
- The sender is unaware of the indentities of the individual receipients and of the size of the group
- A multicast group is specified by a Class D internet address-that is an address whose first 4 bytes are 1110 in IPV4
- Being a member of a multicast group allows a computer to receive IP packets sent to the group

IP Multicast, Cont'd.

- The membership of multicast group is dynamic, allowing computers to join or leave at any time and to join an arbitrary number of groups
- It is possible to send datagrams to a multicast group without being a member
- Multicast Address allocation may be permanant or temporary

Java API to IP Multicast

- Java API provides a datagram interface to IP multicast through the class *MulticastSocket,* which is a subclass of *DatagramSocket* with the aditional capabilities of being able to join multicast groups

- The class *MulticastSocket* provides two aternative constructors, allowing sockets to be created to use either a specified local port or any free local port

Java API to IP Multicast

- A process can join a multicast group with a given multicast address by invoking the *joinGroup* method of its multicast socket

- The socket joins a multicast group at a given port and it will receive datagrams sent by processes on other computers to that group at that port

- A process can leave a specified group by invoking the *leaveGroup* method of its multicast socket

Multicast Peer Joins a Group and Sends and Receives Datagrams

```
import java.net.*;
import java.io.*;
public class MulticastPeer{
        public static void main(String args[]){
         // args give message contents & destination multicast group (e.g. "228.5.6.7")
        MulticastSocket s =null;
         try {
                InetAddress group = InetAddress.getByName(args[1]);
                s = new MulticastSocket(6789);
                s.joinGroup(group);
                byte [] m = args[0].getBytes();
                DatagramPacket messageOut =
                        new DatagramPacket(m, m.length, group, 6789);
                s.send(messageOut);

         // continued on the next slide
```

Continued

```
            // get messages from others in group
                    byte[] buffer = new byte[1000];
                    for(int i=0; i< 3; i++) {
                        DatagramPacket messageIn =
                                new DatagramPacket(buffer, buffer.length);
                        s.receive(messageIn);
                        System.out.println("Received:" + new String(messageIn.getData()));
                    }
                    s.leaveGroup(group);
                }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
                }catch (IOException e){System.out.println("IO: " + e.getMessage());}
            }finally {if(s != null) s.close();}
        }
    }
```

Summary

In this session

- We have presented a overview of computer networking with reference to the communication requirements of distributed system
- We have presented the Java interface to the TCP and UDP prtocols
- We dicussed the characteristics of interprocess communication in a distributed system
- We have discussed Middleware layers such as
  - •External data representation and Marshalling (CORBA, Java object serialisation, XML)
  - •Distributed communication in Client-Server and Group modes

Questions

Thank you

- https://www.youtube.com/watch?v=YY0SvestyaQ