Al Sham Private University
Faculty of Informatics Engineering

جامعة الشام الخاصة
كلية الهندسة المعلوماتية

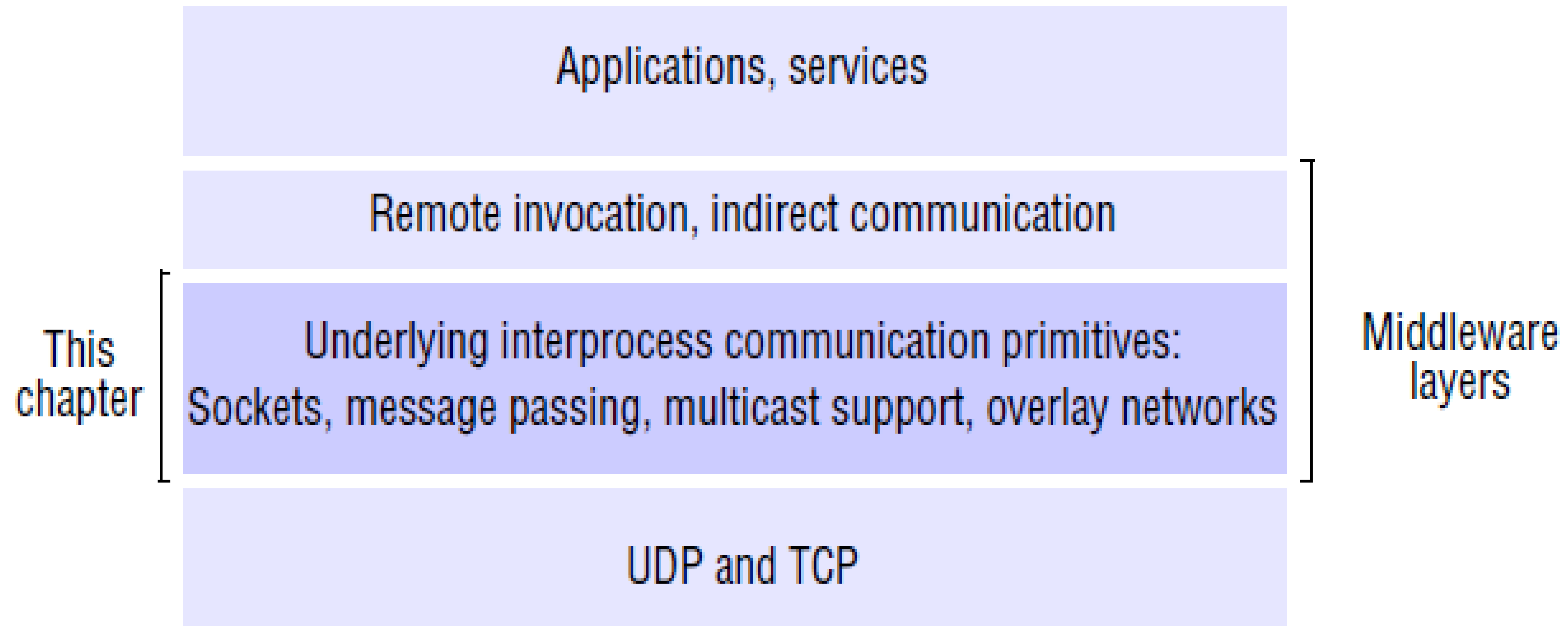
نظم معلومات موزعة
Distributed Information Systems

Lecture 3: Inter-process Communication

اعداد: أ. غاندي هسام

Introduction

- This chapter are concerned with the communication aspects of **middleware**.



The API for the Internet protocols

- we discuss the general characteristics of inter-process communication and then discuss the Internet protocols as an example, explaining how programmers can use them, either by means of UDP messages or through TCP streams.
- Message passing between a pair of processes can be supported by two message communication operations, send and receive, defined in terms of destinations and messages.
- To communicate, one process sends a message (a sequence of bytes) to a destination and another process at the destination receives the message.
- This activity involves the communication of data from the sending process to the receiving process and may involve the synchronization of the two processes.

The characteristics of inter-process communication

- Synchronous and asynchronous communication:

- queue is associated with each message destination. Sending processes cause messages to be added to remote queues and receiving processes remove messages from local queues.

- In the *synchronous* form of communication, the sending and receiving processes synchronize at every message. In this case, both *send* and *receive* are *blocking* operations.

- In the *asynchronous*, the use of the send operation is nonblocking in that the sending process is allowed to proceed as soon as the message has been copied to a local buffer, and the transmission of the message proceeds in parallel with the sending process.

- Message destinations:

- messages are sent to (Internet address, local port) pairs. A local port is a message destination within a computer, specified as an integer.

- Processes may use multiple ports to receive messages. Any process that knows the number of a port can send a message to it.
- Servers generally publicize their port numbers for use by clients.

- Reliability

- As far as the validity property is concerned, a point-to-point message service can be described as reliable if messages are guaranteed to be delivered despite a ‘reasonable’ number of packets being dropped or lost.

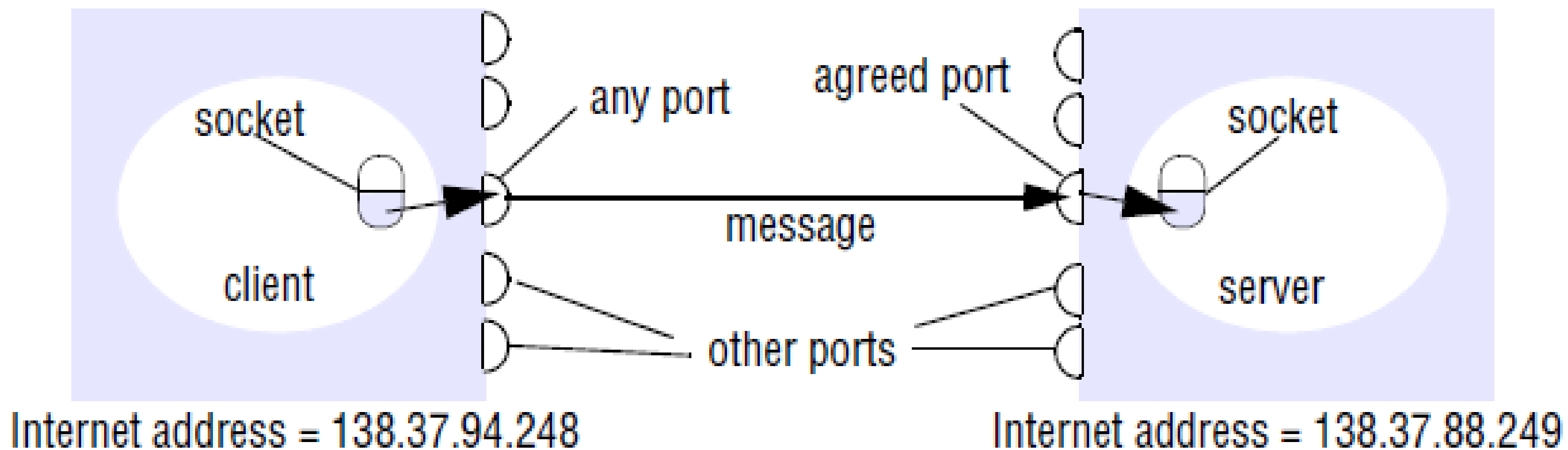
- ❑ Ordering:

- Some applications require that messages be delivered in sender order that is, the order in which they were transmitted by the sender.

- The delivery of messages out of sender order is regarded as a failure by such applications.

Sockets

- Both forms of communication (UDP and TCP) use the socket abstraction, which provides an endpoint for communication between processes.
- Sockets originate from BSD UNIX but are also present in most other versions of UNIX, including Linux as well as Windows and the Macintosh.
- Inter-process communication consists of transmitting a message between a socket in one process and a socket in another process



- For a process to receive messages, its socket must be bound to a local port and one of the Internet addresses of the computer on which it runs.

❑ Java API for Internet addresses:

- As the IP packets underlying UDP and TCP are sent to Internet addresses, Java provides a class, InetAddress, that represents Internet addresses.

- Users of this class refer to computers by Domain Name System (DNS) hostnames.

```
InetAddress aComputer = InetAddress.getByName(“aspu.edu.sy");
```

UDP datagram communication

- A *datagram* sent by UDP is transmitted from a sending process to a receiving process without acknowledgement or retries.
- A datagram is transmitted between processes when one process sends it and another receives it.
- To send or receive messages a process must first create a socket bound to an Internet address of the local host and a local port.
- A server will bind its socket to a server port – one that it makes known to clients so that they can send messages to it.
- A client binds its socket to any free local port.
- The receive method returns the Internet address and port of the sender, in addition to the message, allowing the recipient to send a reply.

❑ Java API for UDP datagrams:

- The Java API provides datagram communication by means of two classes: *DatagramPacket* and *DatagramSocket*.

- DatagramPacket: This class provides a constructor that makes an instance out of a array of bytes comprising a message, the length of the message and the Interne address and local port number of the destination socket, as follows:

Datagram packet

array of bytes containing message	length of message	Internet address	port number

UDP client sends a message to the server and gets a reply

```
import java.net.*;
import java.io.*;
public class UDPClient{
    public static void main(String args[]){
      // args give message contents and server hostname
      DatagramSocket aSocket = null;
      try {
            aSocket = new DatagramSocket();
            byte [] m = args[0].getBytes();
            InetAddress aHost = InetAddress.getByName(args[1]);
            int serverPort = 6789;
            DatagramPacket request =
                  new DatagramPacket(m,  m.length(), aHost, serverPort);
            aSocket.send(request);
            byte[] buffer = new byte[1000];
            DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
            aSocket.receive(reply);
            System.out.println("Reply: " + new String(reply.getData()));
       } catch (SocketException e){System.out.println("Socket: " + e.getMessage());
       } catch (IOException e){System.out.println("IO: " + e.getMessage());
       } finally { if(aSocket != null) aSocket.close();}
    }
}
```

UDP server repeatedly receives a request and sends it back to the client

```
import java.net.*;
import java.io.*;
public class UDPServer{
    public static void main(String args[]){
        DatagramSocket aSocket = null;
        try{
            aSocket = new DatagramSocket(6789);
            byte[] buffer = new byte[1000];
            while(true){
                DatagramPacket request = new DatagramPacket(buffer, buffer.length);
                aSocket.receive(request);
                DatagramPacket reply = new DatagramPacket(request.getData(),
                    request.getLength(), request.getAddress(), request.getPort());
                aSocket.send(reply);
            }
        } catch (SocketException e){System.out.println("Socket: " + e.getMessage());
        } catch (IOException e) {System.out.println("IO: " + e.getMessage());
        } finally {if (aSocket != null) aSocket.close();}
    }
}
```

- Failure model for UDP datagrams:

- UDP datagrams suffer from the following failures:

Omission failures: Messages may be dropped occasionally, either because of a checksum error or because no buffer space is available at the source or destination.

Ordering: Messages can sometimes be delivered out of sender order.

- Use of UDP:

For some applications, it is acceptable to use a service that is liable to occasional omission failures. For example, the Domain Name System, which looks up DNS names in the Internet, is implemented over UDP. Voice over IP (VOIP) also runs over UDP. UDP datagrams are sometimes an attractive choice because they do not suffer from the overheads associated with guaranteed message delivery.

TCP stream communication

- The API to the TCP protocol, which originates from BSD 4.x UNIX, provides the abstraction of a stream of bytes to which data may be written and from which data may be read.
- The following characteristics of stream:
  - Message sizes: The application can choose how much data it writes to a stream or reads from it.
  - Lost messages: The TCP protocol uses an acknowledgement scheme.
  - Flow control: The TCP protocol attempts to match the speeds of the processes that read from and write to a stream. If the writer is too fast for the reader, then it is blocked until the reader has consumed data.
  - Message duplication and ordering: Message identifiers are associated with each IP packet, which enables the recipient to detect and reject duplicates, or to reorder messages that do not arrive in sender order.

- The following are some notable issues related to stream communication:
  - *Matching of data items*: Two communicating processes need to agree as to the contents of the data transmitted over a stream.
  - *Blocking*: The data written to a stream is kept in a queue at the destination socket. When a process attempts to read data from an input channel, it will get data from the queue or it will block until data becomes available.
  - *Threads*: When a server accepts a connection, it generally creates a new thread in which to communicate with the new client. The advantage of using a separate thread for each client is that the server can block when waiting for input without delaying other clients.

❑ Java API for TCP streams:

- The Java interface to TCP streams is provided in the classes *ServerSocket* and *Socket*

- *ServerSocket*: This class is intended for use by a server to create a socket at a server port for listening for connect requests from clients. Its accept method gets a connect request from the queue.

- *Socket*: This class is for use by a pair of processes with a connection. The client uses a constructor to create a socket, specifying the DNS hostname and port of a server. provides the methods *getInputStream* and *getOutputStream* for accessing the two streams associated with a socket.

TCP client makes connection to server, sends request and receives reply

```
import java.net.*;
import java.io.*;
public class TCPClient {
    public static void main (String args[]) {
        // arguments supply message and hostname of destination
        Socket s = null;
        try{
            int serverPort = 7896;
            s = new Socket(args[1], serverPort);
            DataInputStream in = new DataInputStream( s.getInputStream());
            DataOutputStream out =
                new DataOutputStream( s.getOutputStream());
            out.writeUTF(args[0]);        // UTF is a string encoding; see Sec 4.3
            String data = in.readUTF();
            System.out.println("Received: "+ data) ;
        }catch (UnknownHostException e){
            System.out.println("Sock:"+e.getMessage());
        } catch (EOFException e){System.out.println("EOF:"+e.getMessage());
        } catch (IOException e){System.out.println("IO:"+e.getMessage());
        } finally {if(s!=null) try {s.close();}catch (IOException e){/*close failed*/}}
    }
}
```

TCP server makes a connection for each client and then echoes the client's request

```
import java.net.*;
import java.io.*;
public class TCPServer {
    public static void main (String args[]) {
        try{
            int serverPort = 7896;
            ServerSocket listenSocket = new ServerSocket(serverPort);
            while(true) {
                Socket clientSocket = listenSocket.accept();
                Connection c = new Connection(clientSocket);
            }
        } catch(IOException e) {System.out.println("Listen :"+e.getMessage());}
    }
}
```

```
class Connection extends Thread {
	DataInputStream in;
	DataOutputStream out;
	Socket clientSocket;
	public Connection (Socket aClientSocket) {
	   try {
	      clientSocket = aClientSocket;
	      in = new DataInputStream( clientSocket.getInputStream());
	      out =new DataOutputStream( clientSocket.getOutputStream());
	      this.start();
	   } catch(IOException e) {System.out.println("Connection:"+e.getMessage());}
	}
	public void run(){
	   try {			// an echo server
	      String data = in.readUTF();
	      out.writeUTF(data);
	   } catch(EOFException e) {System.out.println("EOF:"+e.getMessage());
	   } catch(IOException e) {System.out.println("IO:"+e.getMessage());
	   } finally { try {clientSocket.close();}catch (IOException e){/*close failed*/}}
	}
}
```

End of Lecture 3