SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

UNIT-II

COMMUNICATION IN DISTRIBUTED SYSTEM

2 System models

1. Outline

What are the three basic ways to describe Distributed systems? –

- Physical models – consider DS in terms of hardware – computers and devices that constitute a system and their interconnectivity, without details of specific technologies
- Architectural models – describe a system in terms of the computational and communication tasks performed by its computational elements. Client-server and peer-to-peer most commonly used
- Fundamental models – take an abstract perspective in order to describe solutions to individual issues faced by most distributed systems
  - interaction models
  - failure models
  - security models

Difficulties and threats for distributed systems:

- Widely varying modes of use
- Wide range of system environments
- Internal problems
- External threats

2.2 Physical models

- Baseline physical model – minimal physical model of a distributed system as an extensible set of computer nodes interconnected by a computer network for the required passing of messages.

Three generations of distributed systems

- Early distributed systems
  - 10 and 100 nodes interconnected by a local area network
  - limited Internet connectivity
  - supported a small range of services e.g.
    - * shared local printers
    - * file servers
    - * email

* file transfer across the Internet

- Internet-scale distributed systems
  - extensible set of nodes interconnected by a network of networks (the Internet)
- Contemporary DS with hundreds of thousands nodes + emergence of:
  - mobile computing
    * laptops or smart phones may move from location to location – need for added capabilities (service discovery; support for spontaneous interoperation)
  - ubiquitous computing
    * computers are embedded everywhere
  - cloud computing

* pools of nodes that together provide a given service

•Distributed systems of systems (ultra-large-scale (ULS) distributed systems)

SRM UNIVERSITY (Under section 3 of UGC Act 1956)

•significant challenges associated with contemporary DS:

Fig

Distributed systems:	*Early*	*Internet-scale*	*Contemporary*
Scale	Small	Large	Ultra-large
Heterogeneity	Limited (typically relatively homogenous configurations)	Significant in terms of platforms, languages and middleware	Added dimensions introduced including radically different styles of architecture
Openness	Not a priority	Significant priority with range of standards introduced	Major research challenge with existing standards not yet able to embrace complex systems
Quality of service	In its infancy	Significant priority with range of services introduced	Major research challenge with existing services not yet able to embrace complex systems

2.3 Architectural Models

Major concerns: make the system *reliable*, *manageable*, *adaptable* and *cost-effective*

2.3.1 Architectural elements

- What are the entities that are communicating in the distributed system?
- How do they communicate, or, more specifically, what communication paradigm is used?
- What (potentially changing) roles and responsibilities do they have in the overall architecture?
- How are they mapped on to the physical distributed infrastructure (what is their placement)?

Communicating entities

- From system perspective: **processes**
  - in some cases we can say that:
    - * **nodes** (sensors)
    - * **threads** (endpoints of communication)
- From programming perspective
  - ***objects***
    - * computation consists of a number of interacting objects representing natural units of decomposition for the given problem domain
    - * Objects are accessed via interfaces, with an associated interface defi- nition language (or IDL)

– ***components*** – emerged due to some weaknesses with distributed objects

 * offer problem-oriented abstractions for building distributed systems

 * accessed through interfaces

  ·+ assumptions to components/interfaces that must be present (i.e. making all dependencies explicit and providing a more complete contract for system construction.)

– ***web services***

 * closely related to objects and components

 * intrinsically integrated into the World Wide Web

  ·using web standards to represent and discover services

The World Wide Web consortium (W3C):

Web service is a software application identified by a URI, whose interfaces and bindings are capable of being defined, described and discovered as XML artefacts. A Web service supports direct interactions with other software agents using XML-based message exchanges via Internet-based protocols.

- objects and components are often used within an organization to develop tightly coupled applications
- web services are generally viewed as complete services in their own right

Communication paradigms

What is:

- interprocess communication?
- remote invocation?
- indirect communication?

Interprocess communication – low-level support for communication between processes in distributed systems, including *message-passing* primitives, direct access to the API offered by Internet protocols (socket programming) and support for *multicast communication*

Remote invocation – calling of a remote operation, procedure or method

Request-reply protocols – a pattern with message-passing service to support client-server computing

Remote procedure call (RPC)

- procedures in processes on remote computers can be called as if they are proce- dures in the local address space
- supports client-server computing with servers offering a set of operations through a service interface and clients calling these operations directly as if they were available locally
  - RPC systems offer (at a m

Remote method invocation (RMI)

- strongly resemble RPC but in a world of distributed objects
- tighter integration into object-orientation framework

In RPC and RMI –

- senders-receivers of messages
  - coexist at the same time
  - are aware of each other’s identities

Indirect communication

- Senders do not need to know who they are sending to (*space uncoupling*)
- Senders and receivers do not need to exist at the same time (*time uncoupling*)

Key techniques in indirect communication:

- Group communication
- Publish-subscribe systems:

- (sometimes also called distributed event-based systems)
- publishers distribute information items of interest (events) to a similarly large number of consumers (or subscribers)

•Message queues:

- (publish-subscribe systems offer a one-to-many style of communication), message queues offer a point-to-point service
- producer processes can send messages to a specified queue
- consumer processes can
  - * receive messages from the queue or
  - * be notified

•Tuple spaces (also known as generative communication):

- processes can place arbitrary items of structured data, called tuples, in a

- other processes can either read or remove such tuples from the tuple space by specifying patterns of interest
- readers and writers do not need to exist at the same time (Since the tuple space is persistent)

•Distributed shared memory (DSM):

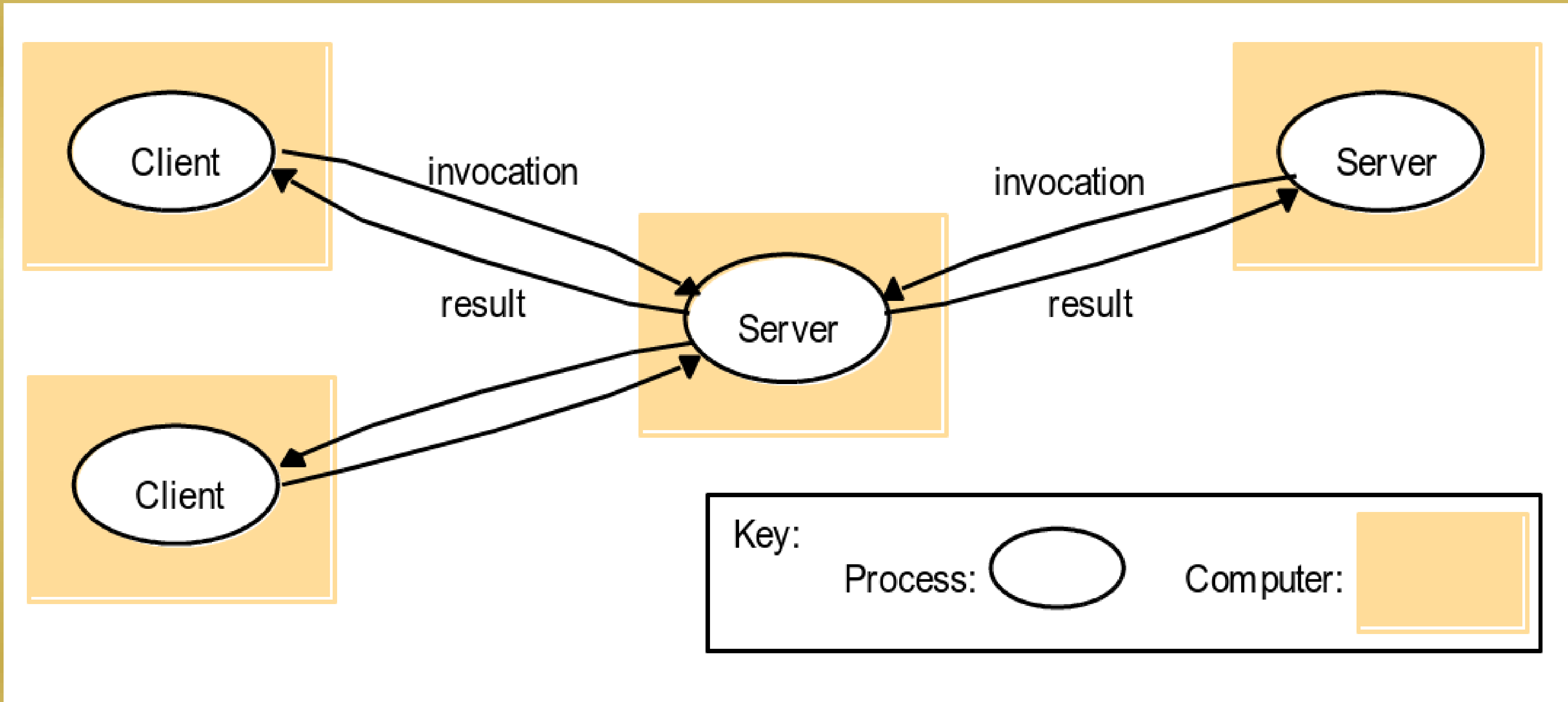
- abstraction for sharing data between processes that do not share physical memory

Communicating entities (what is communicating)		*Communication paradigms (how they communicate)*		
System-oriented entities	*Problem-oriented entities*	*Interprocess communication*	*Remote invocation*	*Indirect communication*
Nodes	Objects	Message passing	Request-reply	Group communication
Processes	Components	Sockets	RPC	Publish-subscribe
	Web services	Multicast	RMI	Message queues
				Tuple spaces
				DSM

Roles and responibilities

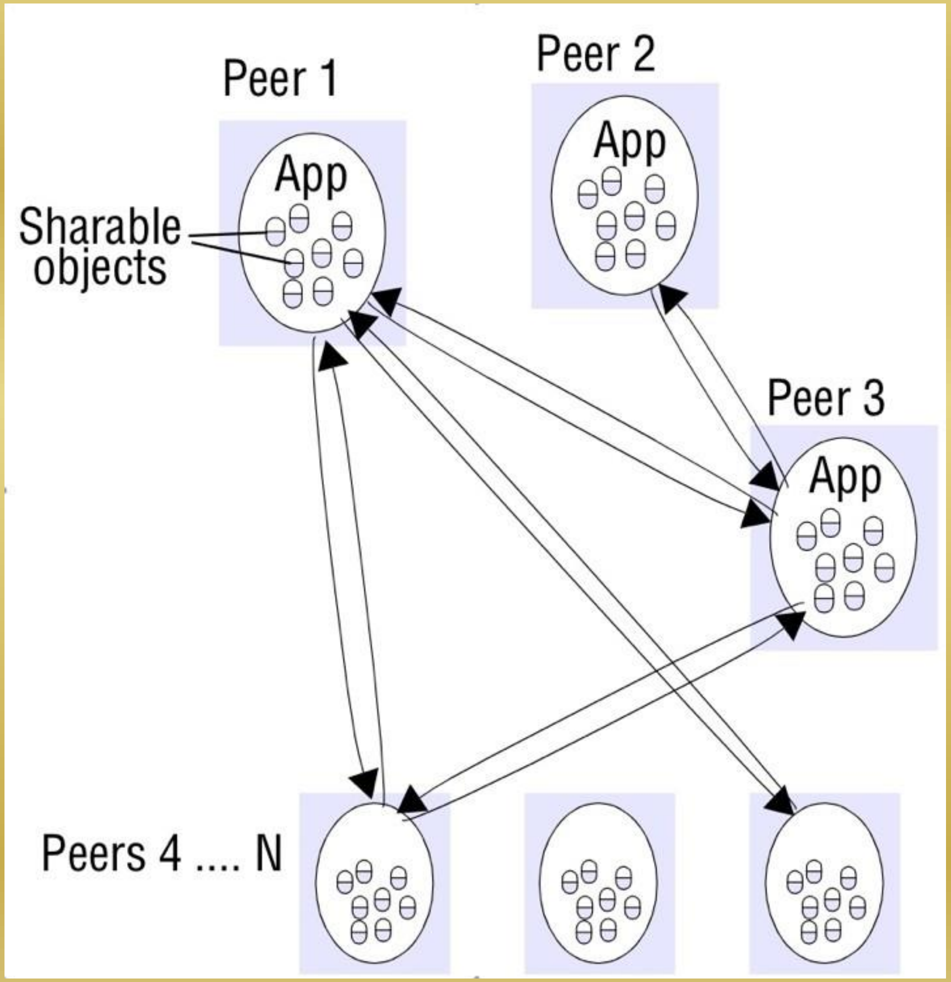
- Client-server

Figure 2.3 Clients invoke individual



•Peer-to-peer

Figure 2.4a Peer-to-peer



– same set of interfaces to each other

2.3 Placement

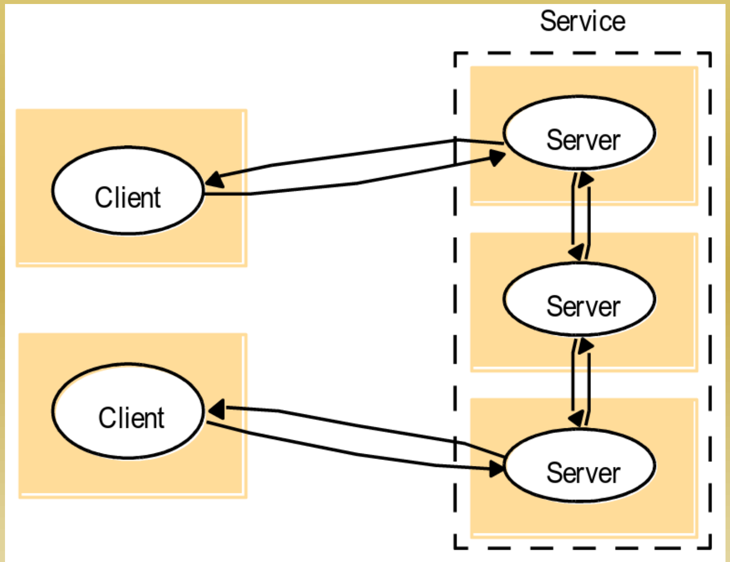
•crucial in terms of determining the DS properties:

- performance
- reliability
- security

Possible placement strategies:

- *mapping of services to multiple servers*
  - mapping distributed objects between servers, or
  - replicating copies on several hosts
  - more closely coupled multiple-servers –

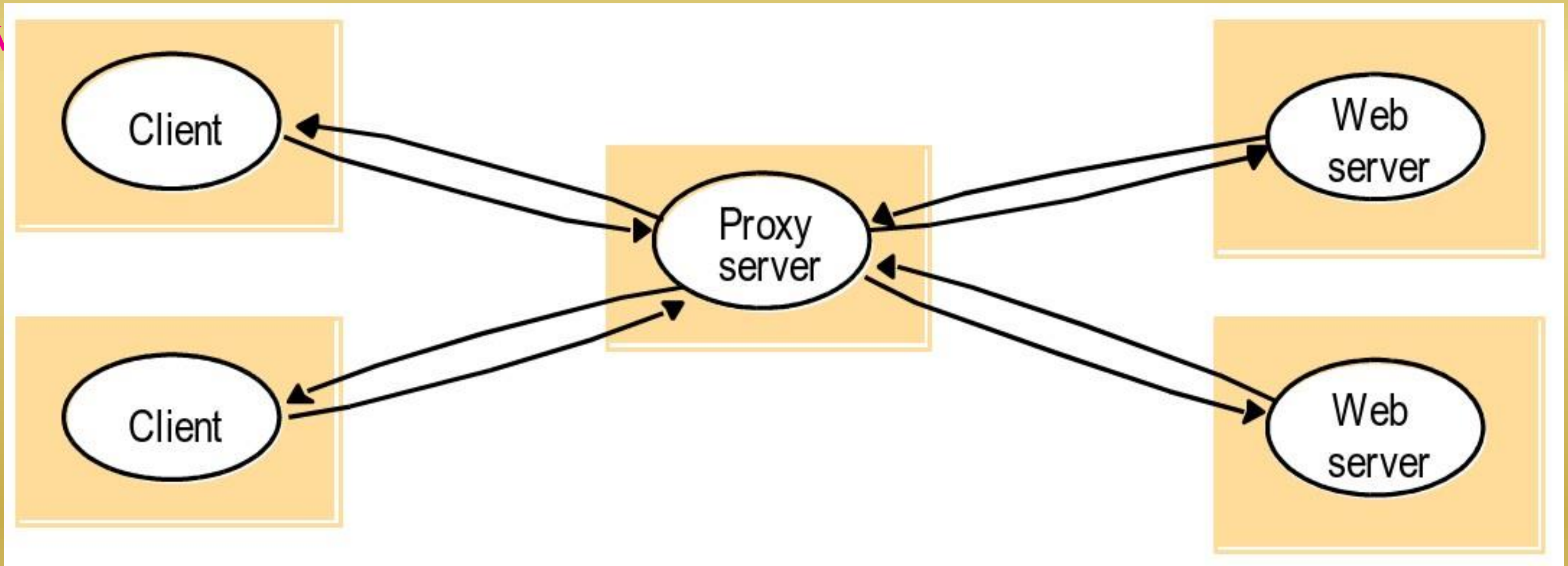
Figure 2.4b A service provided by multi- ple servers



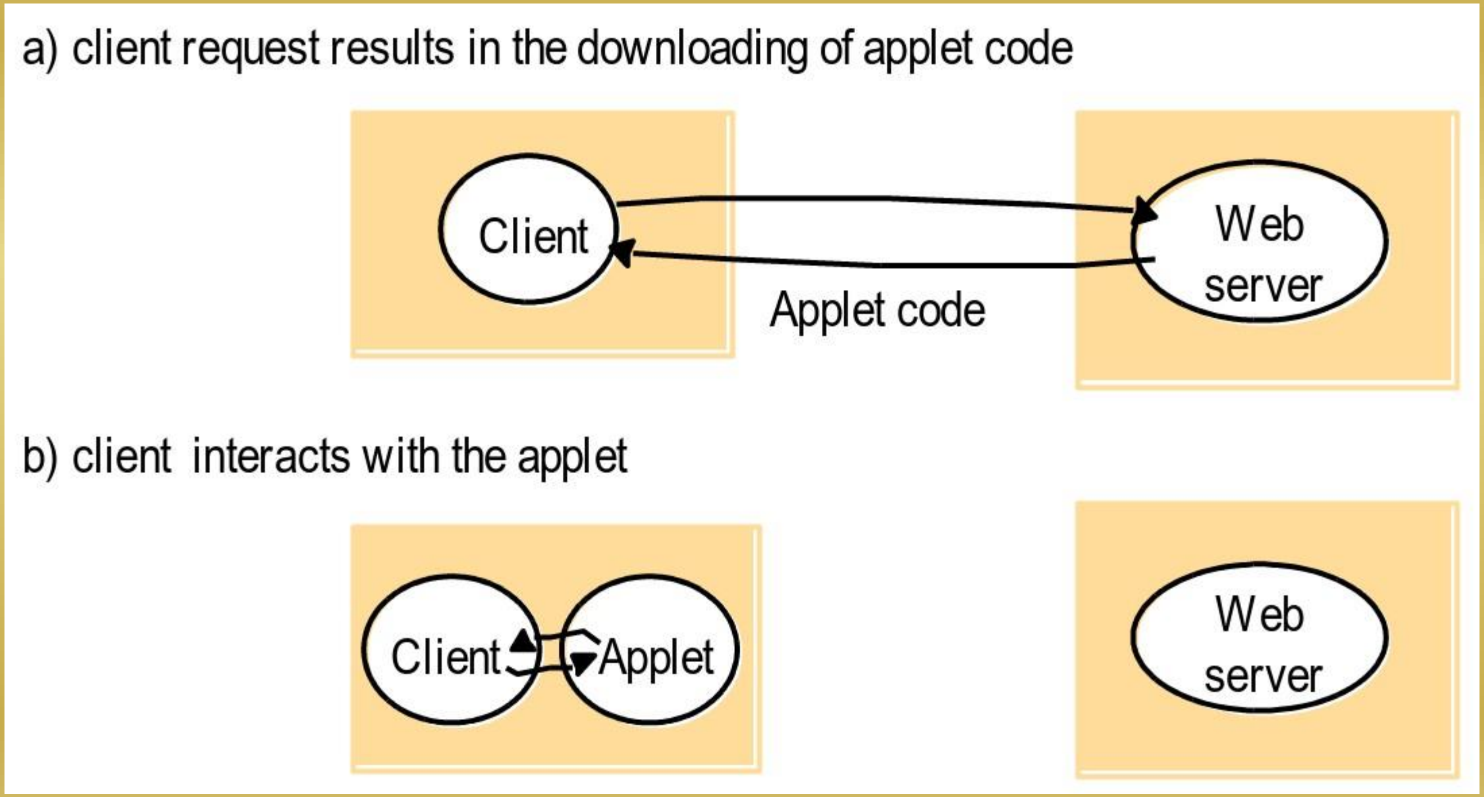
- *cachin g*

– A cache is a store of recently used data objects that is closer to one client or a particular set of clients than the objects themselves

Figure 2.5 Web proxy serv



- *mobile code*
  - Applets are an example of mobile code Figure 2.6 Web Applets



  - yet another possibility – *push* model: server initiates interaction (e.g. on information updates on it)

- *mobile agents*
  - Mobile agent – running program (including both code and data) that travels from one computer to another in a network carrying out a task on someone’s behalf (e.g. collecting information), and eventually re- turning with the results.
  - could be used for
    - * software maintenance
    - * collecting information from different vendors’ databases of prices

Possible security threats with mobile code and mobile agents...

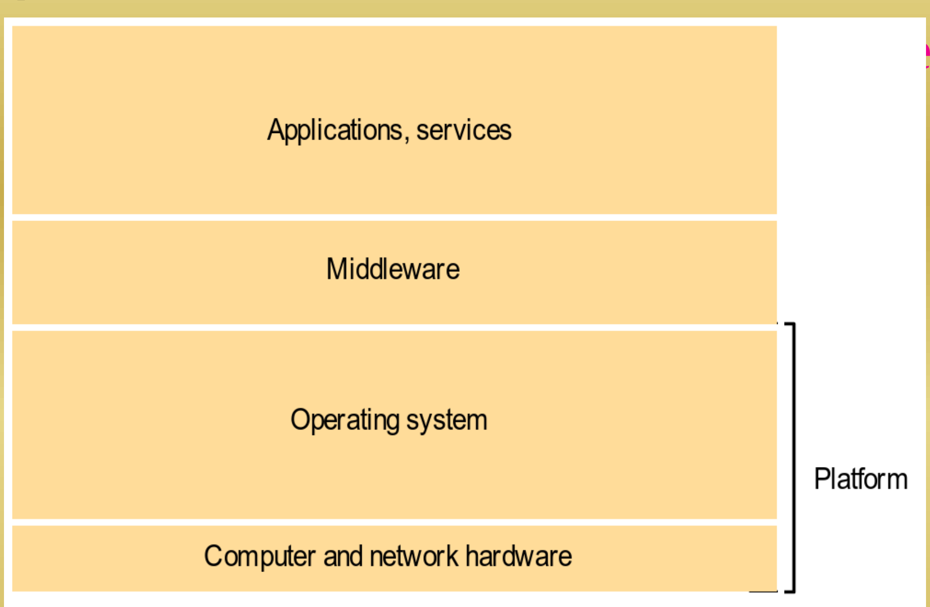
2.3.2 Architectural patterns

Layering

Layered approach – complex system partitioned into a number of layers:

- vertical organisation of services
- given layer making use of the services offered by the layer below
- software abstraction
- higher layers unaware of implementation details, or any other layers beneath them

Platform and Middleware



layers in distributed systems

- A platform for distributed systems and applications consists of the lowest-level hardware and software layers.

- Middleware – a layer of software whose purpose is to mask hetero-geneity and to provide a convenient programming model to application programmers.

Tiered architecture

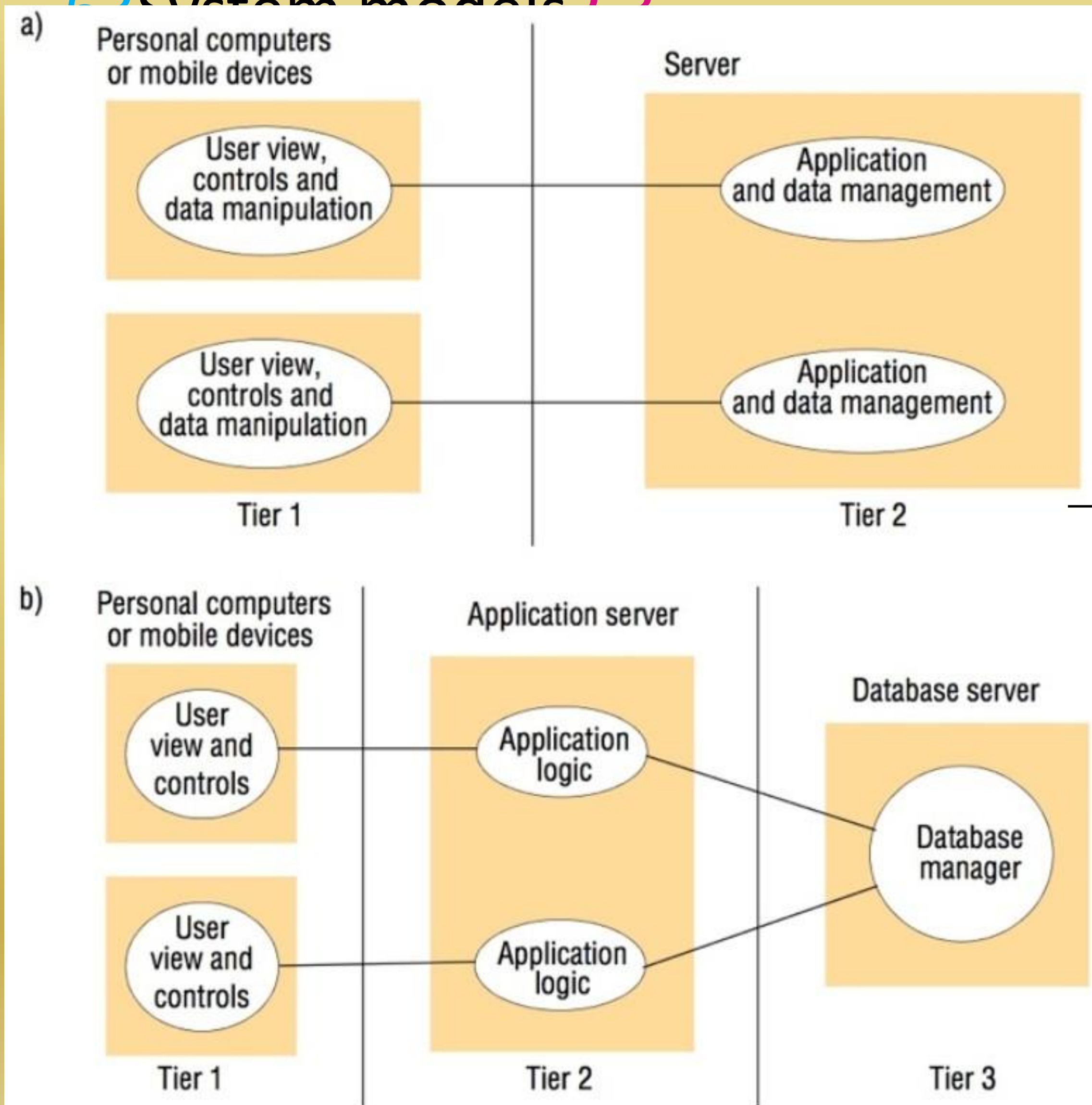
Tiering is a technique to organize functionality of a given layer and place this functionality into appropriate servers and, as a secondary consideration, on to physical nodes

Example: two-tier and three-tier architecture

functional decomposition of a given application, as follows:

- presentation logic
- application logic
- data logic

Figure 2.8 Two-tier and three-tier



- three aspects partitioned into two processes
- (+) low latency
- (-) splitting application logic

- (+) one-to-one mapping from logical elements to physical servers
- (-) added complexity, network traffic and latency

• extension to the standard client-server style of interaction in WWW

AJAX (Asynchronous Javascript And XML) – a way to create interactive, partially/selectively-updatable webpages

– Javascript forntend and server-based backend

Figure 2.9 AJAX example: soccer score updates

```
new Ajax.Request('scores.php?game=Arsenal:Liverpool',
    {onSuccess: updateScore});
function updateScore(request) {
    ..(..request contains the state of the Ajax request including the returned result.
      The result is parsed to obtain some text giving the score, which is used to update the relevant portion of the current page.)
    .....
}
```

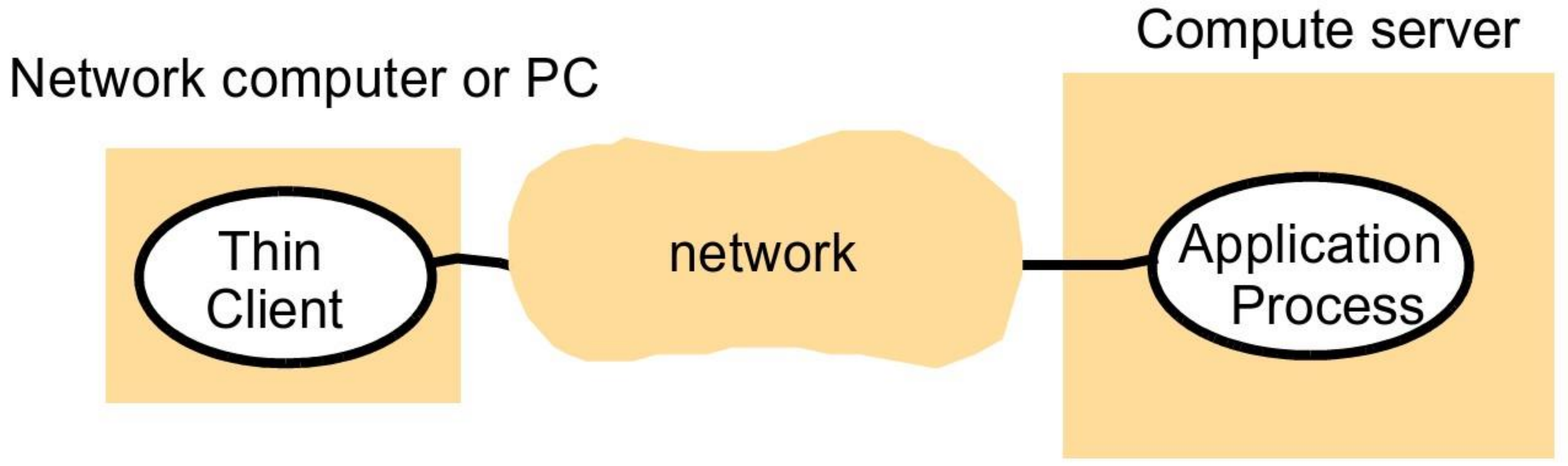
(two-tier architecture)

Thin clients

•enabling access to sophisticated networked services (e.g. cloud services) with few assumptions to client device

•software layer that supports a window-based user interface (local) for executing remote application programs or accessing services on remote computer

Fig

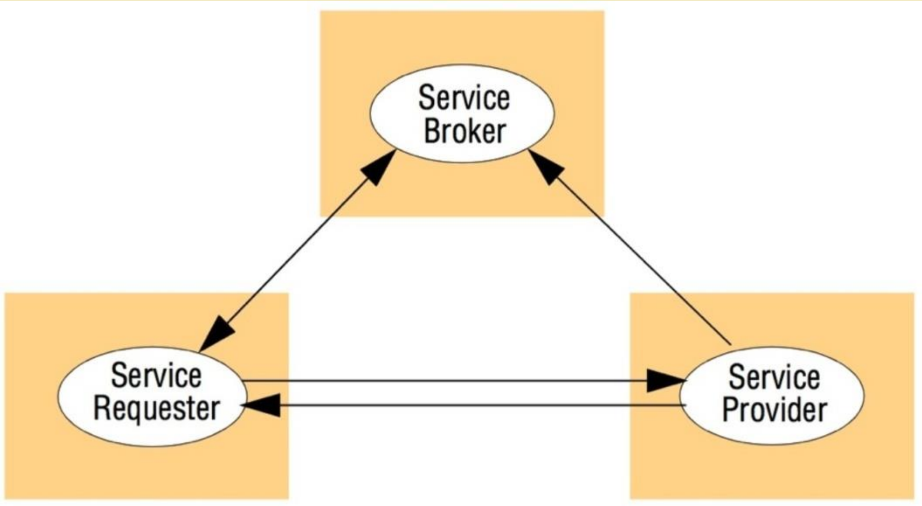


Concept led to Virtual Network Computing (VNC) – VNC clients accessing VNC servers using VNC protocol

Other commonly occurring patterns

- ***proxy* pattern**
  - designed to support location transparency in RPC or RMI
  - proxy created in local address space, with same interface as the remote object
- ***brokerage* in web services**
  - supporting interoperability in potentially complex distributed infrastructures
  - service provider, service requestor and service broker
  - brokerage reflected e.g. in registry in Java RMI and naming service in CORBA

Figure 2.11 The web service architectural patt



- ***Reflection*** **pattern**
  - a means of supporting both:
    - * introspection (the dynamic discovery of properties of the system)
    - * intercession (the ability to dynamically modify structure or behaviour)
  - used e.g. in Java RMI for generic dispatching
  - ability to intercept incoming messages or invocations

– dynamically discover interface offered by a given object

– discover and adapt the underlying architecture of the system

2.3.3 Associated middleware solutions

The task of middleware is to provide a higher-l development of distributed systems and, through layering, to abstract over heterogene- ity in the underlying infrastructure to promote interoperability and portability.

Categories of middleware

Figure 2.12 Categories of mid

Major categories:	*Subcategory*	*Example systems*
Distributed objects (Chapters 5, 8)	Standard	RM-ODP
	Platform	CORBA
	Platform	Java RMI
Distributed components (Chapter 8)	Lightweight components	Fractal
	Lightweight components	OpenCOM
	Application servers	SUN EJB
	Application servers	CORBA Component Model
	Application servers	JBoss
Publish-subscribe systems (Chapter 6)	-	CORBA Event Service
	-	Scribe
	-	JMS
Message queues (Chapter 6)	-	Websphere MQ
	-	JMS
Web services (Chapter 9)	Web services	Apache Axis
	Grid services	The Globus Toolkit
Peer-to-peer (Chapter 10)	Routing overlays	Pastry
	Routing overlays	Tapestry
	Application-specific	Squirrel
	Application-specific	OceanStore
	Application-specific	Ivy
	Application-specific	Gnutella

Limitations of middleware

Some communication-related functions can be completely and reliably imple- mented only with the knowledge and help of the application standing at the end points of the communication system.

Example: e-mail transfer need another layer of fault-tolerance that even TCP can- not offer

4. Fundamental models

What is:

- Interaction model?
- Failure model?
- Security model?

1. Interaction model

- processes interact by passing messages –
  - communication (information flow) and
  - coordination (synchronization and ordering of activities) between processes

- communication takes place with delays of considerable duration
  - accuracy with which independent processes can be coordinated is limited by these delays
  - and by difficulty of maintaining the same notion of time across all the computers in a distributed system

Behaviour and state of DS can be described by a *distributed algorithm:*

- steps to be taken by each interacting process
- + transmission of messages between them

State belonging to each process is completely private

Performance of communication channels

- *latency* – delay between the start of message’s transmission from one process and the beginning of receipt by another
- *bandwidth* of a computer network – the total amount of information that can be transmitted over it in a given time
- *Jitter* – the variation in the time taken to deliver a series of messages

Computer clocks and timing events

- *clock drift rate* – rate at which a computer clock deviates from a perfect refer- ence clock

Two variants of the interaction model

Synchronous distributed systems:

- The time to execute each step of a process has known lower and upper bounds
- Each message transmitted over a channel is received within a known bounded time
- Each process has a local clock whose drift rate from real time has a known bound

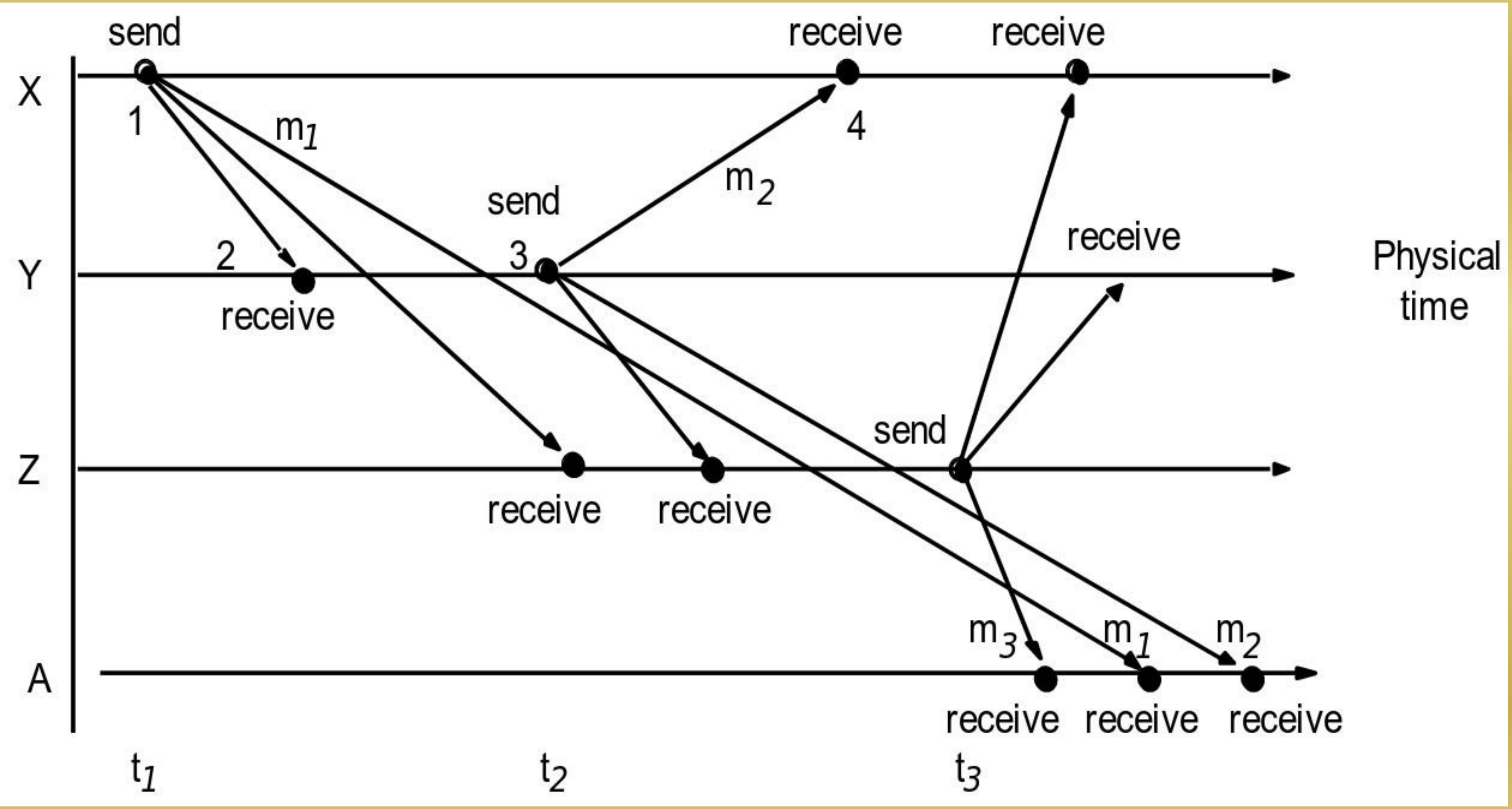
Asynchronous distributed systems:

No bounds on:

- Process execution speeds
- Message transmission delays
- Clock drift rates

Event ordering

Figure 2.13 Real-time ordering of eve



- *Logical time* – based on event ordering

2. Failure model

•faults occur in:

– any of the computers (including software faults)

– or in the network

•Failure model defines and classifies the faults

Omission failures

•process or communication channel fails to perform actions it is supposed to do

Process omission failures

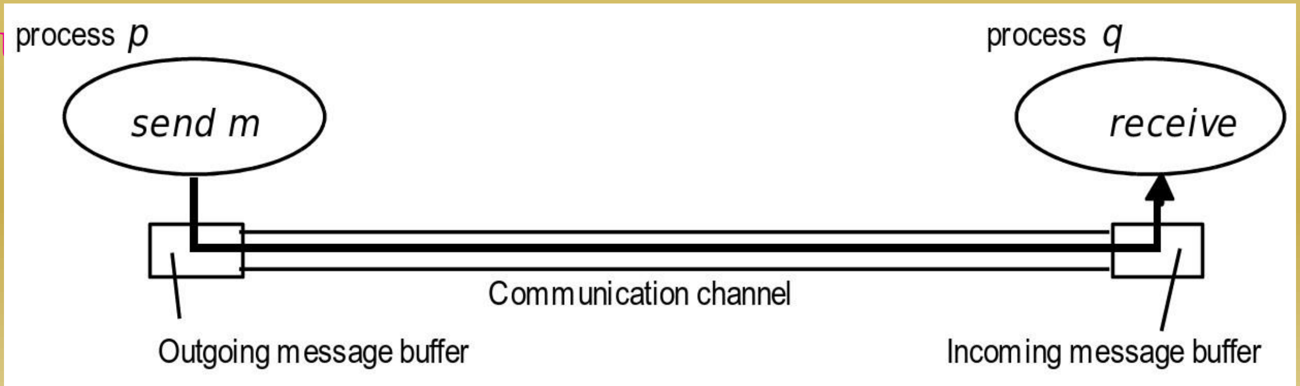
•cheaf omission failure of a process is to crash

– crash is called *fail-stop* if other processes can detect certainly that the

Communication omission failures

- communication channel does not transport a message p's outgoing from message buffer to q's incoming message buffer
  - known as dropping messages
    - * send-omission failures
    - * receive-omission failures
    - * channel-omission failures

Fig



All failures so far: ***benigh failures***

Arbitrary failures

arbitrary or ***Byzantine failure*** is used to describe the worst possible failure se-

mar

Class of failure	Affects	Description
Fail-stop	Process	Process halts and remains halted. Other processes may detect this state.
Crash	Process	Process halts and remains halted. Other processes may not be able to detect this state.
Omission	Channel	A message inserted in an outgoing message buffer never arrives at the other end's incoming message buffer.
Send-omission	Process	A process completes a *send*, but the message is not put in its outgoing message buffer.
Receive-omission	Process	A message is put in a process's incoming message buffer, but that process does not receive it.
Arbitrary (Byzantine)	Process or channel	Process/channel exhibits arbitrary behaviour: it may send/transmit arbitrary messages at arbitrary times, commit omissions; a process may stop or take an incorrect step.

Timing failures

- applicable in synchronous distributed systems

Figure 2.16 Timing failures

Class of Failure	*Affects*	*Description*
Clock	Process	Process's local clock exceeds the bounds on its rate of drift from real time.
Performance	Process	Process exceeds the bounds on the interval between two steps.
Performance	Channel	A message's transmission takes longer than the stated bound.

Masking failures

- knowledge of the failure can enable a new service to be designed to mask the failure of the components on which it depends

Reliability of one-to-one communication

- reliable communication:
  - *Validity:* Any message in the outgoing message buffer is eventually deliv- ered to the incoming message buffer
  - *Integrity:* The message received is identical to one sent, and no messages are delivered twice

3. Security model

- modular nature of distributed systems and their openness exposes them to attack by
  - both external and internal agents
- Security model defines and classifies attack forms,
  - providing a basis for the analysis of threats
  - basis for design of systems that are able to resist them

the security of a distributed system can be achieved by securing the processes and the channels used for their interactions and by protecting the objects that they encapsulate against unauthorized access.

Protecting objects

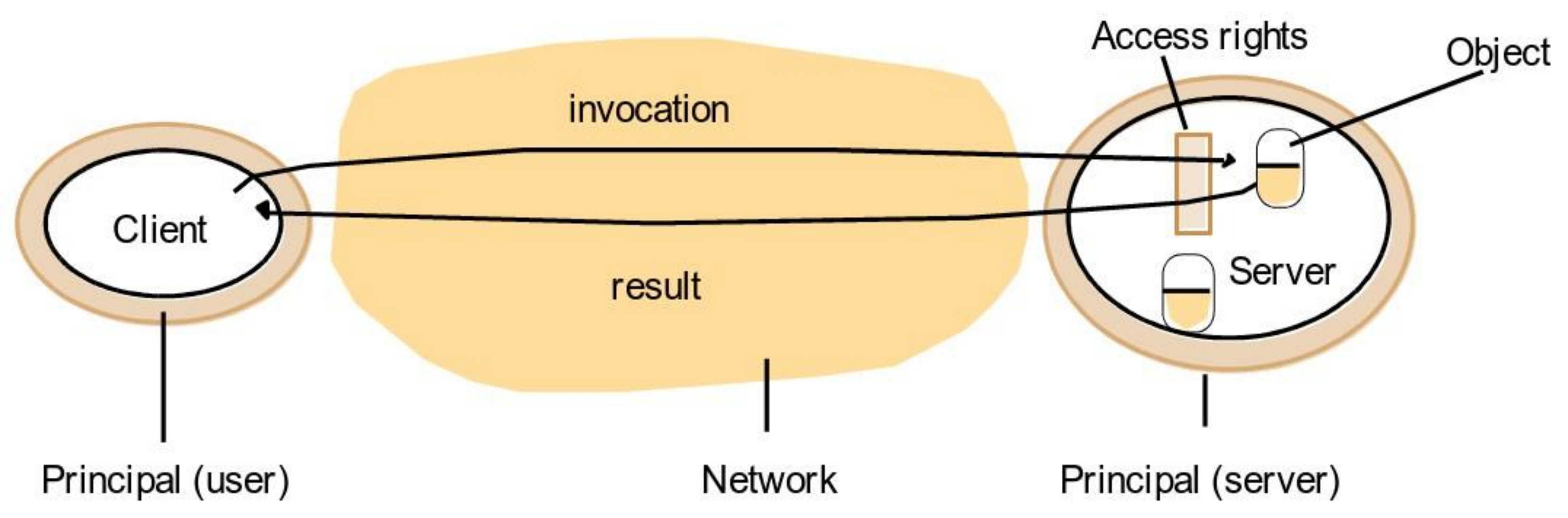
•Users with access rights

•association of each invocation and each result with the authority on which it is issued

– such an authority is called *a principal*

∗ principal may be a user or a process

Figure 2.17 Objects and principals



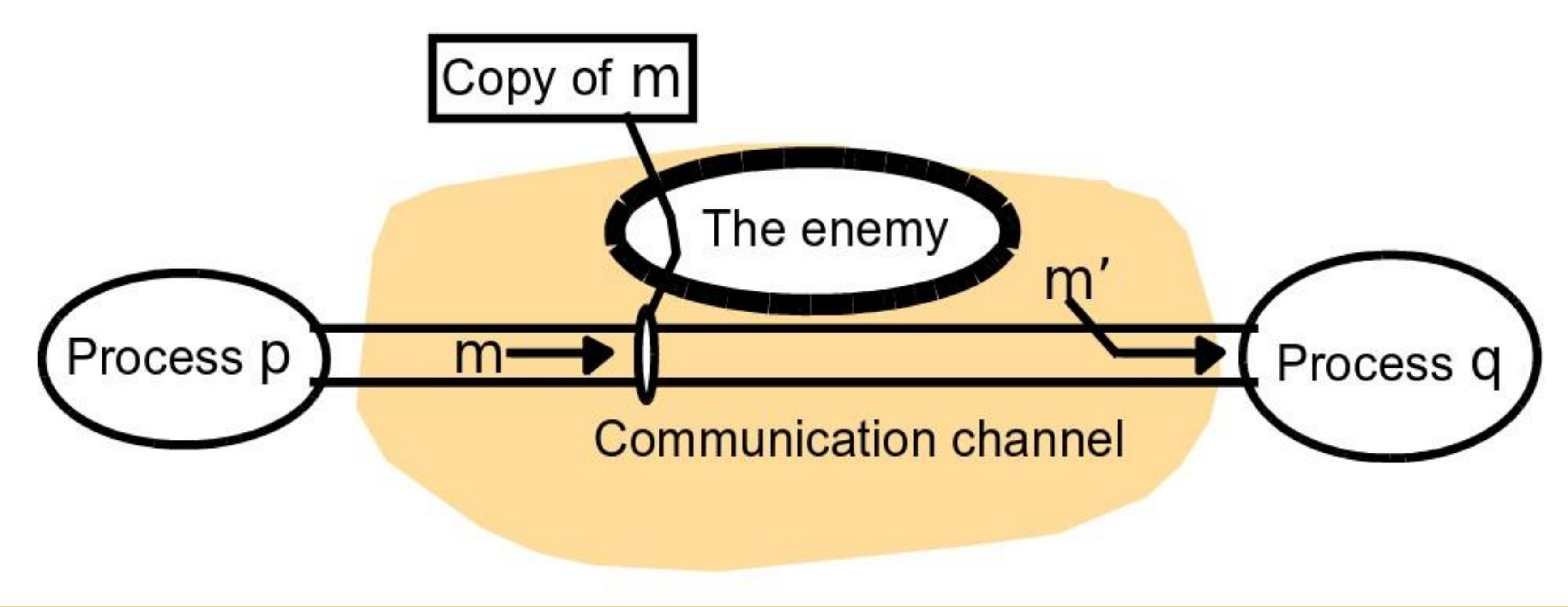
Securing processes and their interactions

- securing communications over open cahnnels

- open service interfaces

The enemy

or also: *adversary*



Threats to processes

- lack of knowledge of true source of a message
  - problem both to server and client side
  - example: spoofing a mail server

Threats to communication channels

- threat to the privacy and integrity of messages
- can be defeated using *secure channels*

Defeating security threats

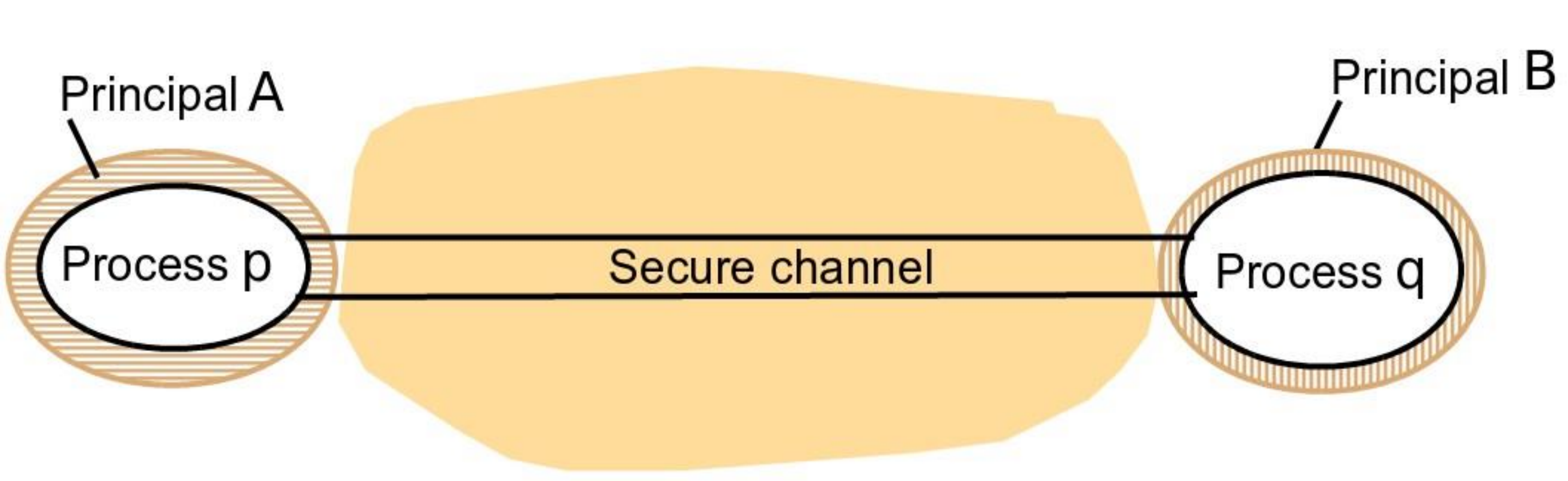
Cryptography and shared secrets

- Cryptography is the science of keeping messages secure
- Encryption is the process of scrambling a message in such a way as to hide its contents

Authentication

- based on shared secrets authentication of messages – proving the identities sup- plied by their senders

Figure 2.19 Secure



Properties of a secure channel:

- Each of the processes knows reliably the identity of the principal on whose behalf the other process is executing
- A secure channel ensures the privacy and integrity (protection against tamper- ing) of the data transmitted across it

- Each message includes a physical or logical timestamp to prevent messages from being replayed or reordered

Other possible threats from an enemy

- Denial of service:
  - the enemy interferes with the activities of authorized users by making ex- cessive and pointless invocations on services or message transmissions in a network, resulting in overloading of physical resources (network band- width, server processing capacity)
- Mobile code:
  - execution of program code from elsewhere, such as the email attachment etc.

The uses of security models

Security analysis involves

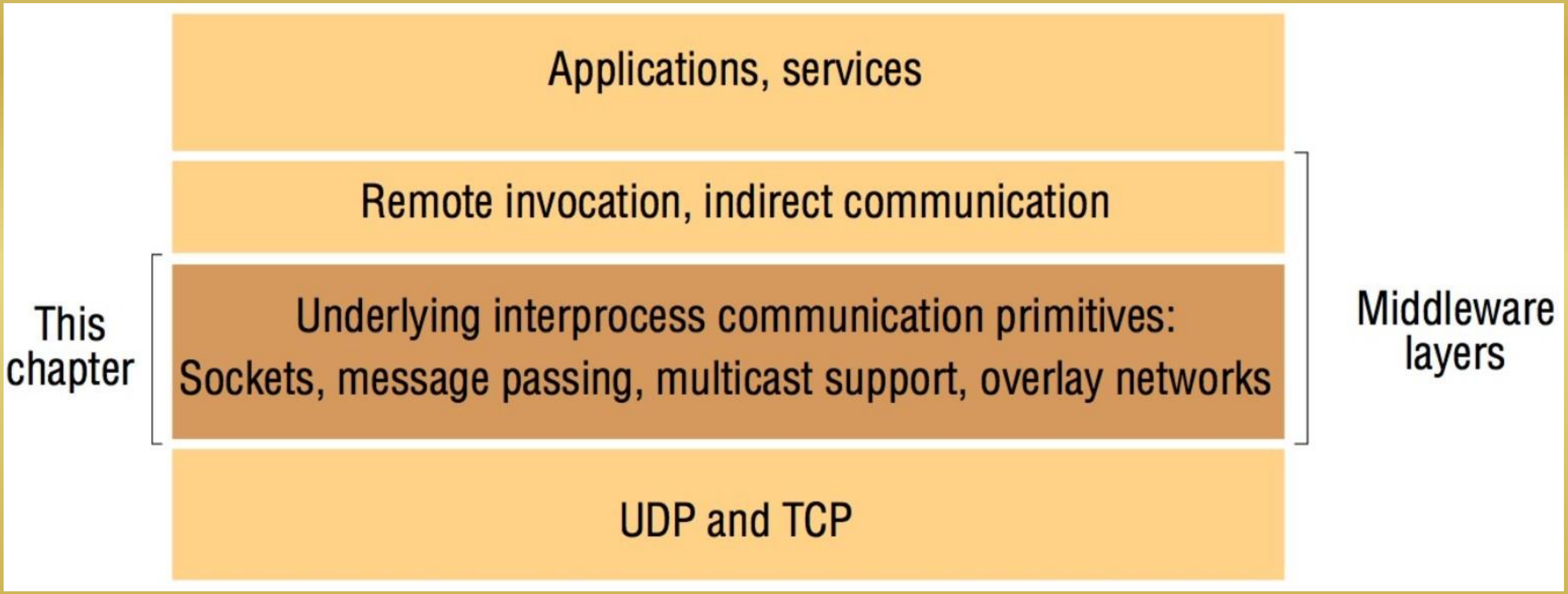
- the construction of a threat model:
  - listing all the forms of attack to which the system is exposed
  - an evaluation of the risks and consequences of each

End of week 2

4 Interprocess communication

4.1 Introduction

Figure 4.1 Middleware layers



How middleware and application programs can use UDP and TCP?

What is specific about IP multicast? Why/how could it be made more reliable? What is an overlay network?

What is MPI?

2. The API for the Internet protocols

1.The characteristics of interprocess communication

Synchronous and asynchronous communication

synchronous – sending and receiving processes synchronize at every message

- both *send* and *receive* – blocking operations
  - whenever *send* is issued – sending process blocked until *receive* is issued
  - whenever *receive* is issued by a process, it is blocked until the message arrives

asynchronous – *send* – nonblocking; *receive* – either blocking or non-

blocking In case threads are supported (Java) blocking receive has no w

disadvantages

thread is handling the communication while other threads can continue their work

Message destinations

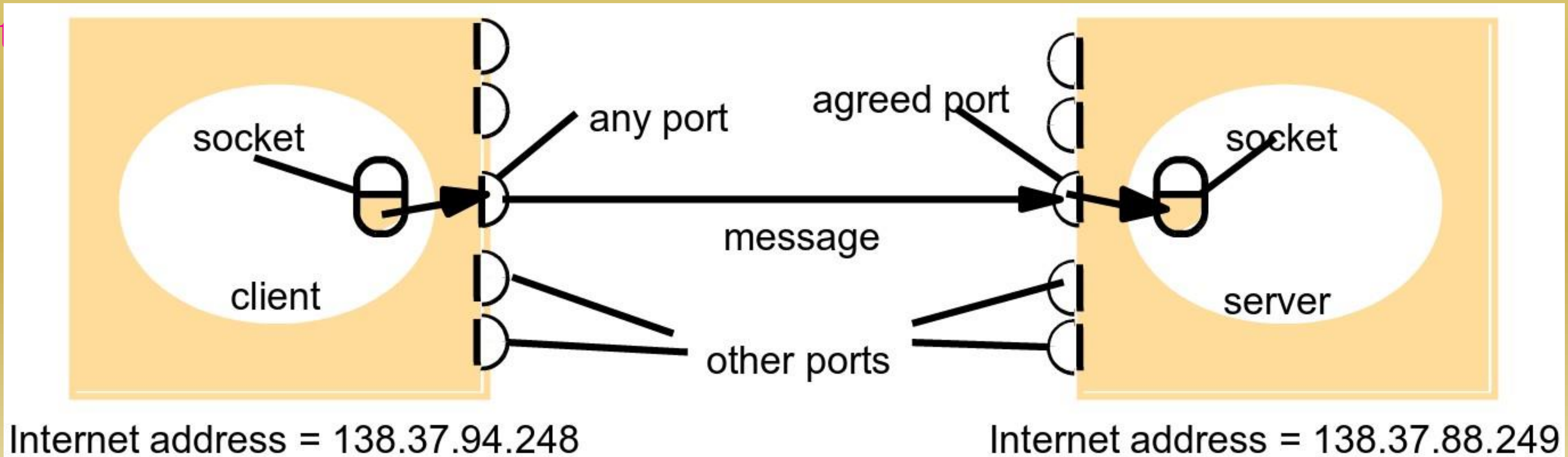
•messages sent to (*Internet address, local port)*

Reliability & ordering – also important factors

4.2.2 Sockets

socket – abstraction providing an endpoint for communication betwee

Figure 4.2 Sockets and port



SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

Java API for Internet addresses

- Java class InetAddress referring to Domain Name System (DNS) hostnames

```
InetAddress aComputer = InetAddress.getByName("bruno.dcs.qmul.ac.uk");
```

4.2.3 UDP datagram communication

– datagram transmission without acknowledgement or retries

- create a socket bound to an Internet address of the local host and a local port

1. A server will bind its socket to a server port
2. A client binds its socket to any free local port

- The receive method returns the Internet address and port of the sender, in addi- tion to the message (allowing the recipient to send a reply)

 Issues related to datagram communication:

Message size:

•in IP protocol – $\leq 2^{16}$(incl. headers), but in most environments ≤ 8 kilobytes

Blocking:

•Sockets normally provide non-blocking *sends* and blocking *receives*

Timeouts:

•if needed, should be fairly large in comparison with the time for message transmission

Receive from any:

•by default every message is placed in a receiving queue

– but it is possible to connect a datagram socket to a particular remote

Failure model for UDP datagrams

(In Chapter 2: failure model for communication channels – reliable communication in terms of 2 properties – *integrity* and *validity*)

UDP datagrams suffer from

- Omission failures
- Ordering

Applications – provide your own checks!

Use of UDP

- Domain Name System (DNS)
- Voice over IP (VOIP)

No overheads associated with guaranteed message delivery. But overheads on:

- the need to store state information at the source and destination
- transmission of extra messages
- latency for the sender

Java API for UDP datagrams

2 classes: DatagramPacket and DatagramSocket

Class **DatagramPacket** – provides constructor for making an instance out of

- an array of bytes comprising a message
- the length of the message
- and the Internet address and
- local port number of the destination socket

DatagramPacket

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

On the receiving side: DatagramPacket has another constructor + methods get-Data, getPort and getAddress

- Class **DatagramSocket** – supports sockets for sending and receiving datagrams
- •constructor with port number
  - – has also no-argument case – system to choose a free port
  - – send and receive
    - * argument – DatagramPacket
- •Methods:
  - – setSoTimout – block receive for specified time before throwing InterruptedIOException
  - – connect – to connect to a particular remote port and internet address for exclusive communication to/from there

Figure 4.3 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();}
} }
```

Figure 4.4 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();}
    }
}
```

4.2.4 TCP stream communication

Network characteristics hidden by stream abstraction:

•Message sizes

•Lost messages

•Flow control

•Message duplication and ordering

•Message destinations

– once connection established – simply read/write to/from stream

– to establish connection

* connect request (from client)

* accept request (from server)

 Pair of socets associated with srtream – read and write

Issues related to stream communication:

- Matching data items – (e.g. int should be followed by float – matching in both side)
- Blocking –
  - while trying to read data before it has arrived in queue
  - writing data to the stream, but the TCP flow-control mechanism still wait- ing for data acknowledgements etc.
- Threads – usually used

Failure model

- integrity

– checksums

– sequence numbers

•validity

– timeouts

– retransmission

Use of TCP

HTTP, FTP, Telnet, SMTP

Java API for TCP streams

Classes ServerSocket and Socket

Class **ServerSocket**:

•to listen connect reqests from

- accept method
  - gets a connect request from the queue or
  - if the queue is empty, blocks until one arrives
  - result of executing accept – an instance of Socket – a socket to use for communicating with the client

Class **Socket**:

- for use by pair of processes
- client constructor – to create a socket specifying DNS hostname and port of a server
  - connects to the specified remote computer and port number
- methods:
  - getInputStream and getOutputStream

Figure 4.5 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 Sn 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){System.
      out.
          println("close:"+e.getMessage());}}
} }
```

Figure 4.6 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 */
        }}
} }
```

4.3 External data representation and marshalling

•messages ←–

- data values of many different types
- different floating-point number representations
- integers – big-endian, little-endian order
- ASCII – 1byte; Unicode – 2bytes

⇒ either:

a) convert data to agreed external format, or

b) transmit data in sender's format + format used – recipient converts the values if needed

external data representation: agreed standard for the representation of data structures and primitive values

marshalling: process of taking a collection of data items and assembling them into a form suitable for transmission in a message

unmarshalling: process of disassembling a collection data items from a message at the destination

- CORBA's (Common Object Request Broker Architecture) common data representation (bin, just values)
- Java's object serialization (bin, data + type info)
- XML (Extensible Markup Languaga) (txt, may refer to externally defined *namespaces*)
- Google – *protocol buffers* (both stored and transmitted data)
- JSON (JavaScript Object Notation)http://www.json.org

4.3.1 CORBA's Common Data Representation (CDR)

primitive types: 1.short (16-bit)
2.long (32-bit)
3.unsigned short

4.unsigned
5.float (32-bit)
6.double (64-bit)

7.char
(which long
8.boolean (TRUE, FALSE)
9.octet (8-bit)

10.any (can represent any basic or constructed type)

Constructed (composite) types: sequence of bytes in a particular order: Figure 4.7 CORBA CDR for constructed types

Type	*Representation*
sequence	length (unsigned long) followed by elements in order
string	length (unsigned long) followed by characters in order (can also can have wide characters)
array	array elements in order (no length specified because it is fixed)
struct	in the order of declaration of the components
enumerated	unsigned long (the values are specified by the order declared)
union	type tag followed by the selected member

4.3 External data representation and marshalling

CORBA CDR that contains the three fields of a struct whose respective types are string, string and unsigned long:

- Person struct with value: {‘Smith’, ‘London’,

198

index in sequence of bytes	← 4 bytes →	*notes on representation*
0–3	5	*length of string*
4–7	"Smit"	*‘Smith’*
8–11	"h___"	
12–15	6	*length of string*
16–19	"Lond"	*‘London’*
20-23	"on__"	
24–27	1984	*unsigned long*

The flattened form represents a *Person* struct with value: {‘Smith’, ‘London’, 1984}

```
struct Person {
    string name;
    string place;
    unsigned long year;
};
```

Marshalling through CORBA IDL

Sun XDR standard

- similar to CORBA in many ways
- sending messages between clients and servers in Sun NFS
- http://www.cdk5.net/ipc

4.3.2 Java object serialization

```
public class Person implements Serializable {
    private String name;
    private String place;
    private int year;
    public Person(String aName, String aPlace, int aYear) {
        name = aName;
        place = aPlace;
        year = aYear;
    }
    // followed by methods for accessing the instance variables
}
```

serialization – flattening an object or a connected set of objects into a serial form suitable for storing on disk or transmitting in a message

4.3 External data representation and marshalling

deserialization – vica versa, assuming no a priori knowledge about of types of objects

– self-containness

•serialization of an object + all objects it references as well to ensure that with the object reconstruction, all of its references can be fulfilled at the destination

•recursive procedure

```
Person p = new Person("Smith", "London", 1984);
```

Figure 4.9 Indication of Java serialized

Serialized values				Explanation
Person	8-byte version number		h0	*class name, version number*
3	int year	java.lang.String name:	java.lang.String place:	*number, type and name of instance variables*
1984	5 Smith	6 London	h1	*values of instance variables*

The true serialized form contains additional type markers; h0 and h1 are handles

- serialize:
  - create an instance of the class ObjectOutputStream and invoke its writeObject method
- deserialize:
  - open an ObjectInputStream on the stream and use its readObject method to reconstruct the original object

(de)serialization carried out automatically in RMI

Reflection -– the ability to enquire about the properties of a class, such as the names and types of its instance variables and methods

- enables classes to be created from their names
- a constructor with given argument types to be created for a given class

- Reflection makes it possible to do serialization and deserialization in a completely generic manner

4.3.3 Extensible Markup Language (XML)

- defined by the World Wide Web Consortium (W3C)
- data items are tagged with ‘markup’ strings
- tags relate to the structure of the text that they enclose
- XML is used to:
  - enable clients to communicate with web services
  - defining the interfaces and other properties of web services
  - many other uses
    - archiving and retrieval systems

* specification of user interfaces
* encoding of configuration files in operating systems

•clients usually use SOAP messages to communicate with web services

SOAP – XML format whose tags are published for use by web services and their clients

XML elements and attributes

Figure 4.10 XML definition of the Person structure

```
<person id ="123456789" >
        <name>Smith </name>
        <place >London</ place
        >
        <year >1984 </ year >
        <!-- acomment -->
</ person >
```

Elements: portion of character data surrounded by matching start and end tags

- An empty tag – no content and is terminated with /> instead of >
  - For example, the empty tag <european/> could be included within the <person> ...</person> tag

Attributes: element – generally a container for data, whereas an attribute – used for labelling that data

- Attributes are for simple values
- if data contains substructures or several lines, it must be defined as an element

Names start with letter _ or :

Binary data – expressed in character data in base64

Parsing and well-formed documents

```
<?XML version = "1.0" encoding = "UTF-8" standalone = "yes" ?>
```

XML namespaces – URL referring to the file containing the namespace definitions.

- For example:

```
xmlns:pers = "http://www.cdk5.net/person"
```

Figure 4.11 Illustration of the use of a namespace in the Person structure

```
<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 schemas [www.w3.org VIII] defines the elements and attributes that can ap- pear in a document, how the elements are nested and the order and number of elements, and whether an element is empty or can include text

- used for encoding and validation

Figure 4.12 An XML schema for the Person structure

```
<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>
```

APIs for accessing XML – in Java, Python etc.

4. Remote object references

Java, CORBA

- *remote object reference* is an identifier for a remote object that is valid through- out a distributed system

Figure 4.13 Representation of a remote object reference

32 bits	*32 bits*	*32 bits*	*32 bits*	
Internet address	port number	time	object number	interface of remote object

4. Multicast communication

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

1.Fault tolerance based on replicated services

2.Discovering services in spontaneous

networking 3.Better performance through

replicated data 4.Propagation of event

notifications

4.4.1 IP multicast – An implementation of multicast communication

Java's API to it via the MulticastSocket class

IP multicast

SRM UNIVERSITY (Under section 3 of UGC Act 1956)

- group specified by a Class D Internet address
  - first 4 bits are 1110 in IPv4
- Being a member of a multicast group allows a computer to receive IP packets sent to the group
- membership dynamic
  - computers allowed to join or leave at any time
  - to join an arbitrary number of groups
  - possible to send datagrams to a multicast group without being a membe
- At the application programming level, IP multicast available only via UDP
- Multicast routers

Multicast address allocation:

- Local Network Control Block (224.0.0.0 to 224.0.0.225)
- Internet Control Block (224.0.1.0 to 224.0.1.225)
- Ad Hoc Control Block (224.0.2.0 to 224.0.255.0)
- Administratively Scoped Block (239.0.0.0 to 239.255.255.255) – constrained propagation

Failure model for multicast datagrams

- failure characteristics as UDP datagrams
- *ureliable* multicast

Java API to IP multicast

Figure 4.14 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);
            byte[] buffer = new byte[1000];
            for(int i=0; i< 3; i++) { // get messages from others in group
                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();}
}
  }
```

End of week 4

4.5 Network virtualization: Overlay networks

Network virtualization – construction of many different virtual networks over an existing network

- each virtual network redefines its own addressing scheme, protocols, routing algorithms – depending on particular application on top

1. Overlay networks

overlay network – virtual network consisting of nodes and virtual links, which sits on top of an underlying network (such as an IP network) and offers something that is not otherwise provided:

- a service for a class of applications or a particular higher-level service
  - e.g. multimedia content distribution
- more efficient operation in a given networked environment
  - e.g. routing in an ad hoc network
- an additional feature
  - e.g. multicast or secure communication.

This leads to a wide variety of types of overlay as captured by Figure 4.15

183Interprocess communication4.5

Figure 4.15 Types of ove

Motivation	*Type*	*Description*
Tailored for application needs	Distributed hash tables	One of the most prominent classes of overlay network, offering a service that manages a mapping from keys to values across a potentially large number of nodes in a completely decentralized manner (similar to a standard hash table but in a networked environment).
	Peer-to-peer file sharing	Overlay structures that focus on constructing tailored addressing and routing mechanisms to support the cooperative discovery and use (for example, download) of files.
	Content distribution networks	Overlays that subsume a range of replication, caching and placement strategies to provide improved performance in terms of content delivery to web users; used for web acceleration and to offer the required real-time performance for video streaming [www.kontiki.com].

Figure 4.15 Types of overlay

Tailored for network style	Wireless ad hoc networks	Network overlays that provide customized routing protocols for wireless ad hoc networks, including proactive schemes that effectively construct a routing topology on top of the underlying nodes and reactive schemes that establish routes on demand typically supported by flooding.
	Disruption-tolerant networks	Overlays designed to operate in hostile environments that suffer significant node or link failure and potentially high delays.
Offering additional features	Multicast	One of the earliest uses of overlay networks in the Internet, providing access to multicast services where multicast routers are not available; builds on the work by Van Jacobsen, Deering and Casner with their implementation of the MBone (or Multicast Backbone) [mbone].
	Resilience	Overlay networks that seek an order of magnitude improvement in robustness and availability of Internet paths [nms.csail.mit.edu].
	Security	Overlay networks that offer enhanced security over the underling IP network, including virtual private networks, for example, as discussed in Section 3.4.8.

- Advantages:
  - new network services changes to the underlying network
  - encourage experimentation with network services and the customization of services to particular classes of application
  - Multiple overlays can coexist
- Disadvantages:
  - extra level of indirection (henceperformance penalty)
  - add to the complexity of network services

SRM UNIVERSITY (Under section 3 of UGC Act 1956)

4.5.2 Skype: An example of an overlay network

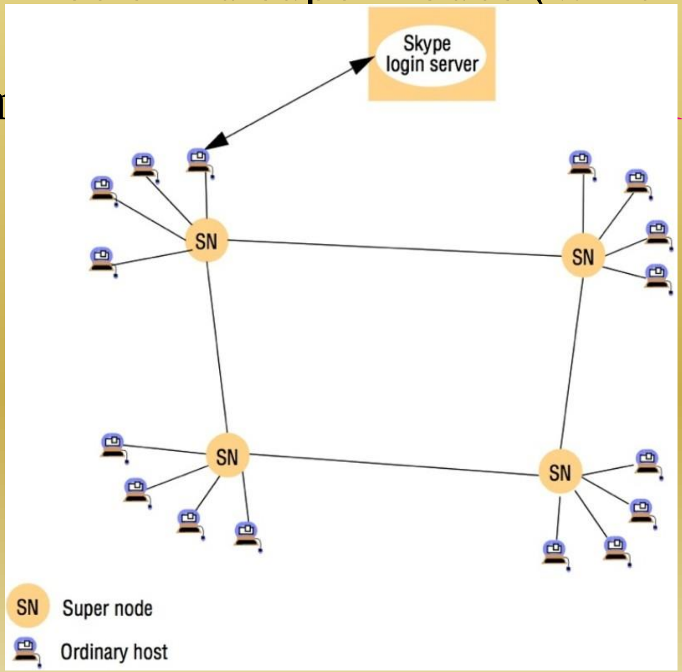
Peer-to-peer application offering VoIP; 370M users (2009); developed by Kazaa

p2p filesharing app

Skype architecture

•hosts and super nodes (which being selected on

dem ...ay architecture



User connection

- users authenticated via login server

Search for users

- super nodes – to perform the efficient search of the global index of users distributed across the super nodes
  - On average, eight super nodes are contacted
  - 3-4 seconds to complete for hosts that have a global IP address (5-6 second, if behind a NAT-enabled router)

Voice connection

- TCP for signalling call requests and terminations and either UDP or TCP for the streaming audio

- UDP is preferred
- TCP can be used in certain circumstances to circumvent firewalls

4.6 Case study: MPI

MPI (The Message Passing Interface)

- A message-passing library specification
  - extended message-passing model
  - not a language or compiler specification
  - not a specific implementation or product
- Full featured; for parallel computers, clusters, and heterogeneous networks
- Designed to provide access to advanced parallel hardware for end users, library writers, and tool developers

MPI as STANDARD

Goals of the MPI standard MPI's prime goals are:

- To provide source-code portability
- To allow efficient

implementations MPI also offers:

- A great deal of functionality
- Support for heterogeneous parallel architectures

4 types of MPI calls

1. Calls used to initialize, manage, and terminate communications
2. Calls used to communicate between pairs of processors (Pair communication)
3. Calls used to communicate among groups of processors (Collective communication)

MPI basic subroutines (functions)

```
MPI_Init: initialise MPI
MPI_Comm_Size: how many PE?
MPI_Comm_Rank: identify the
PE  MPI_Send
MPI_Receive
MPI_Finalise: close
MPI
```

Example (Fortran90) 11.1 Greetings(http://www.ut.ee/~eero/SC/konspekt/Naited/greetings.f90.html)

Figure 4.17 An overview of point-to-point communication in

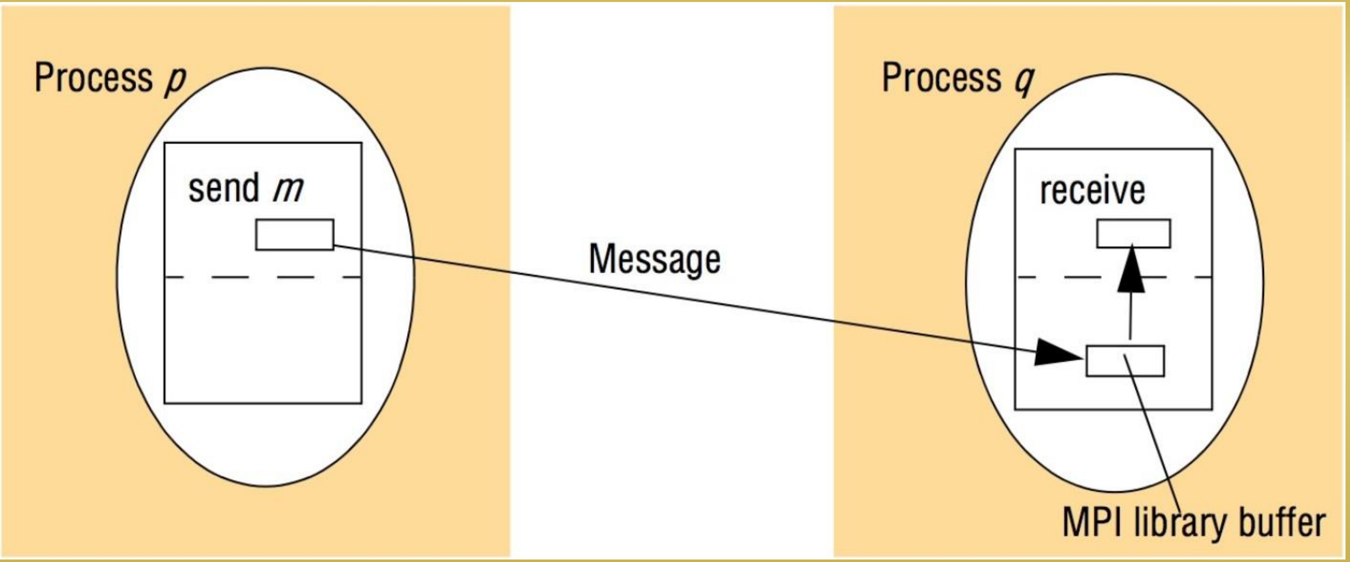


Figure 4.18 Selected send operations in

Send operations	Blocking	Non-blocking
Generic	*MPI_Send*: the sender blocks until it is safe to return – that is, until the message is in transit or delivered and the sender's application buffer can therefore be reused.	*MPI_Isend*: the call returns immediately and the programmer is given a communication request handle, which can then be used to check the progress of the call via *MPI_Wait* or *MPI_Test*.
Synchronous	*MPI_Ssend*: the sender and receiver synchronize and the call only returns when the message has been delivered at the receiving end.	*MPI_Issend*: as with *MPI_Isend*, but with *MPI_Wait* and *MPI_Test* indicating whether the message has been delivered at the receive end.
Buffered	*MPI_Bsend*: the sender explicitly allocates an MPI buffer library (using a separate *MPI_Buffer_attach* call) and the call returns when the data is successfully copied into this buffer.	*MPI_Ibsend*: as with *MPI_Isend* but with *MPI_Wait* and *MPI_Test* indicating whether the message has been copied into the sender's MPI buffer and hence is in transit.
Ready	*MPI_Rsend*: the call returns when the sender's application buffer can be reused (as with *MPI_Send*), but the programmer is also indicating to the library that the receiver is ready to receive the message, resulting in potential optimization of the underlying implementation.	*MPI_Irsend*: the effect is as with *MPI_Isend*, but as with *MPI_Rsend*, the programmer is indicating to the underlying implementation that the receiver is guaranteed to be ready to receive (resulting in the same optimizations),

Case study: MPI