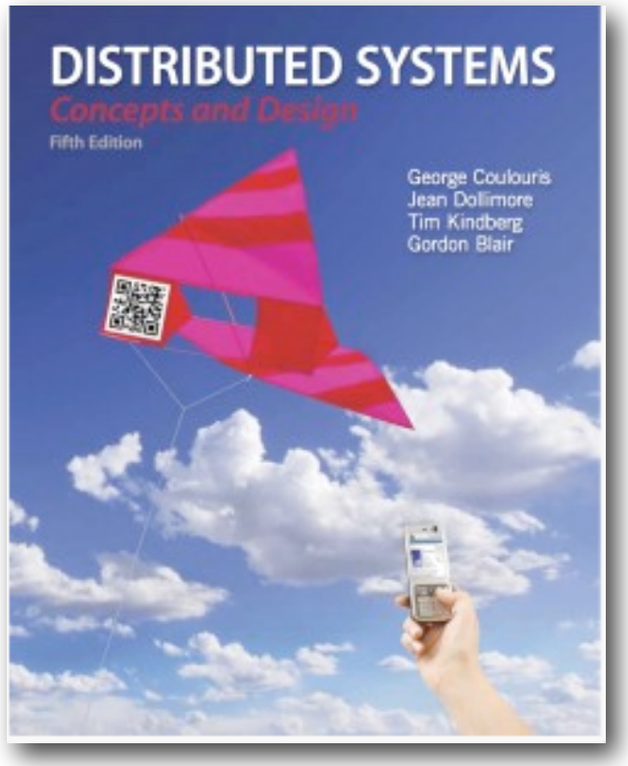
DTU Compute
Department of Applied Mathematics and Computer Science

Communication Paradigms

Nicola Dragoni
Embedded Systems Engineering
DTU Compute

1. Point-to-point Communication
   - Characteristics of Interprocess Communication
   - Sockets
   - Client-Server Communication over UDP and TCP
2. Group (Multicast) Communication
3. Point-to-point Communication
   - Remote Invocation



From the First Lecture (Architectural Models)...

- The architecture of a system is its structure in terms of separately specified components and their interrelationships.

- 4 **fundamental building blocks** (and 4 key questions):

  - **Communicating entities**: *what are the entities that are communicating in the distributed system?*

  - **Communication paradigms**: *how do these entities communicate, or, more specifically, what communication paradigm is used?*

  - **Roles and responsibilities**: *what (potentially changing) roles and responsibilities do these entities have in the overall architecture?*

  - **Placement**: *how are these entities mapped on to the physical distributed infrastructure (i.e., what is their placement)?*

Communication Paradigms

- 3 types:

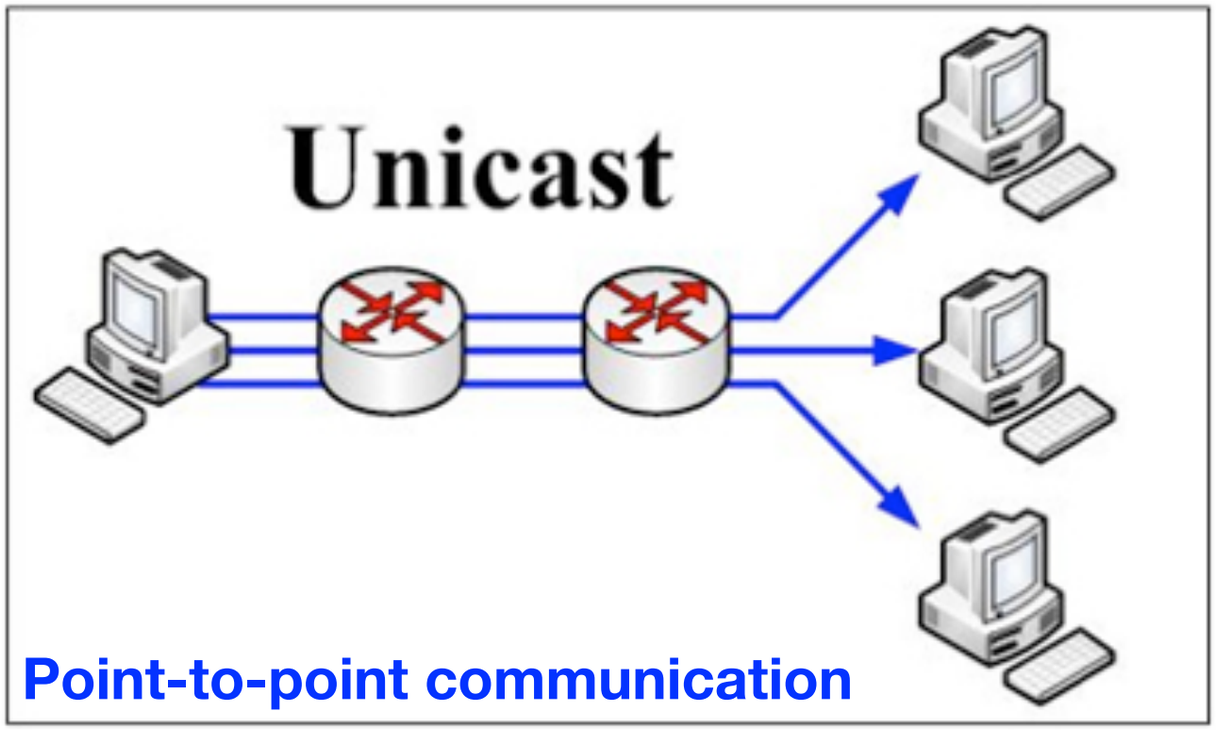
direct communication

- interprocess communication
  low level support for communication between processes in the distributed system, including message-passing primitives, socket programming, multicast communication

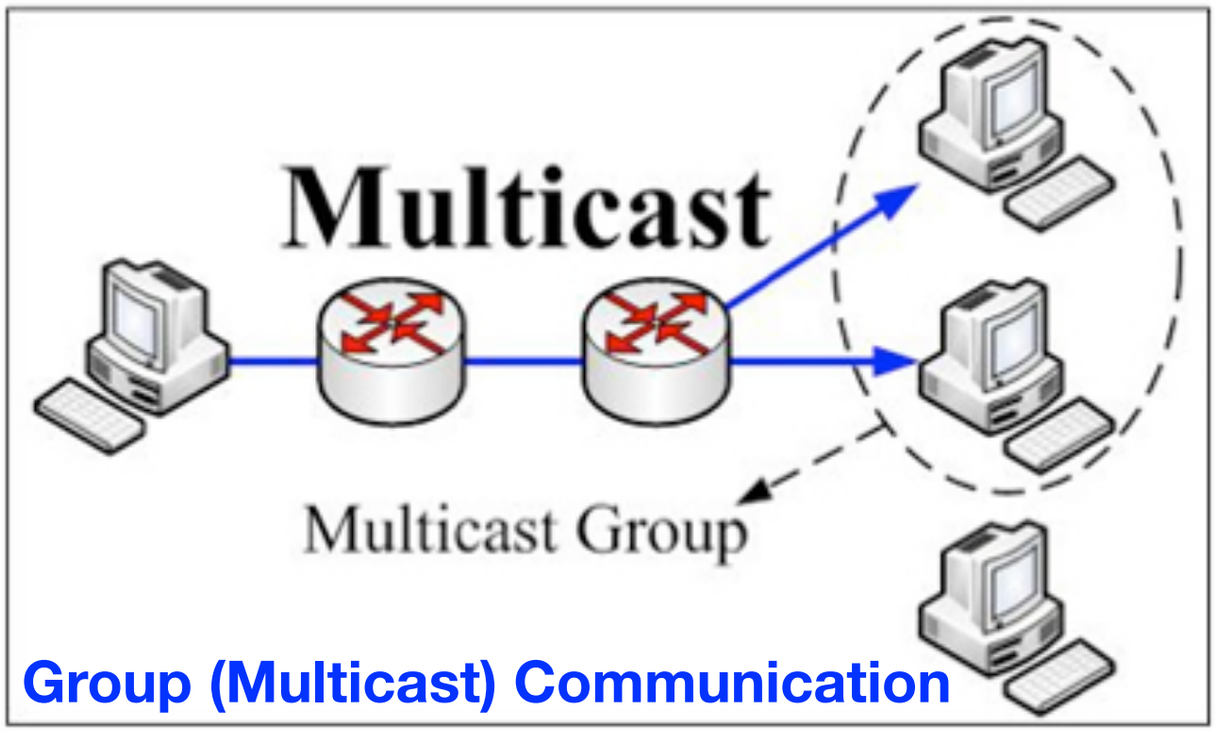
- remote invocation
  most common communication paradigm, based on a two-way exchange between communicating entities and resulting in the calling of a remote operation (procedure or method)

- indirect communication
  communication is indirect, through a third entity, allowing a strong degree of decoupling between senders and receivers.
  Examples: publish subscribe systems, distributed shared memory (DSM).

Unicast VS Multicast



Point-to-point communication



Group (Multicast) Communication

The Characteristics of Interprocess Communication

- Message passing between a pair of processes supported by two communication operations: send and receive

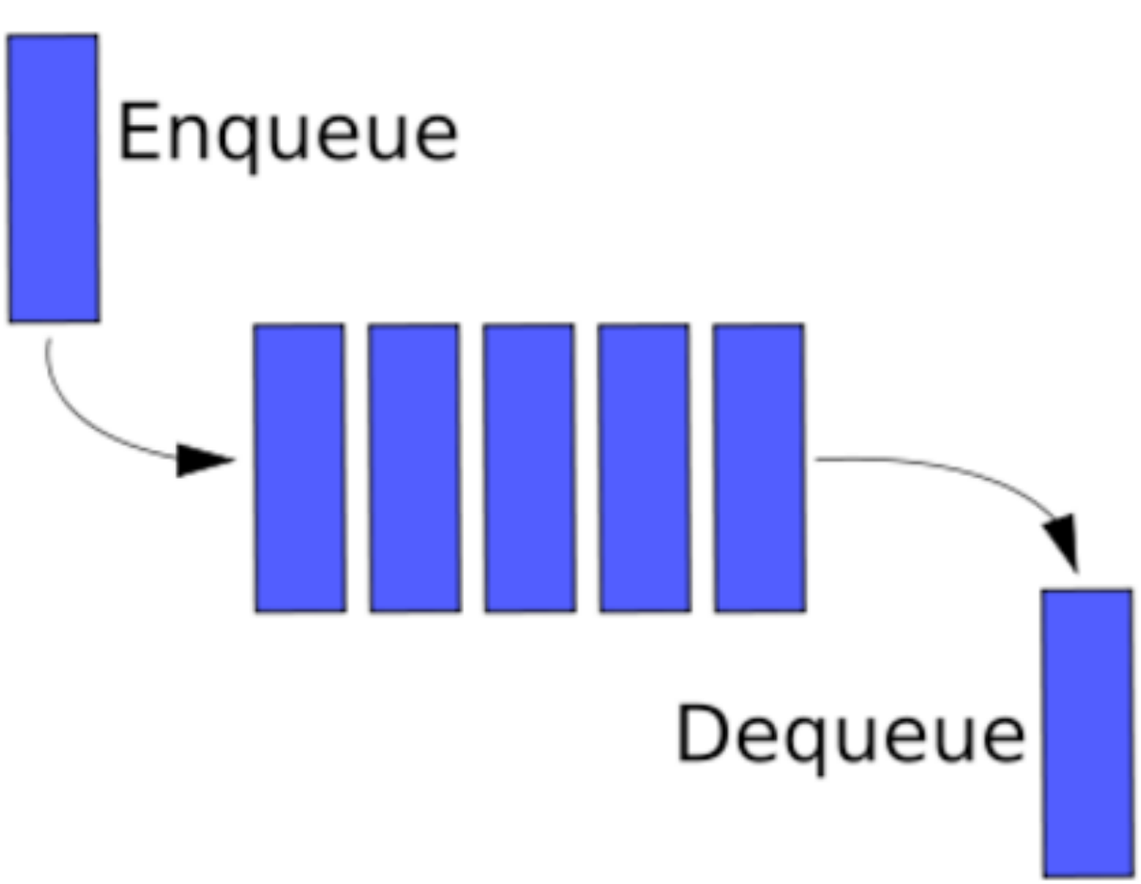
- Defined in terms of destinations and messages

- In order for one process A to communicate with another process B:
  - A sends a message (sequence of bytes) to a destination
  - another process at the destination (B) 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

Sending VS Receiving

- A queue is associated with each message destination
- Sending processes cause messages to be added to *remote* queues
- Receiving processes remove messages from *local* queues



Communication between the sending and receiving process may be either synchronous or asynchronous

Synchronous Communication

- The sending and receiving processes synchronize at every message

- In this case, both send and receive are *blocking operations*:

  - whenever a send is issued the sending process is *blocked* until the corresponding receive is issued

  - whenever a receive is issued the receiving process *blocks* until a message arrives

Asynchronous Communication

- The send operation is *non-blocking*:
  - the sending process is allowed to proceed as soon as the message has been copied to a local buffer
  - the transmission of the message proceeds in parallel with the sending process
- The receive operation can have *blocking* and *non-blocking* variants:
  - [non-blocking] the receiving process proceeds with its program after issuing a receive operation
  - [blocking] receiving process blocks until a message arrives

Message Destinations?

- Usually take the form (address, local port)
  - For instance, in the Internet protocols messages are sent to (Internet address, local port) pairs
- Local port: **message destination within a computer**, specified as an integer. It is commonly used to identify a specific service (ftp, ssh, ...)
- A port has exactly one receiver but can have many senders
- Processes may use multiple ports from which to receive messages
- Any process that knows the number of a port can send a message to it
- Servers generally publicise their port numbers for use by clients

Socket Abstraction

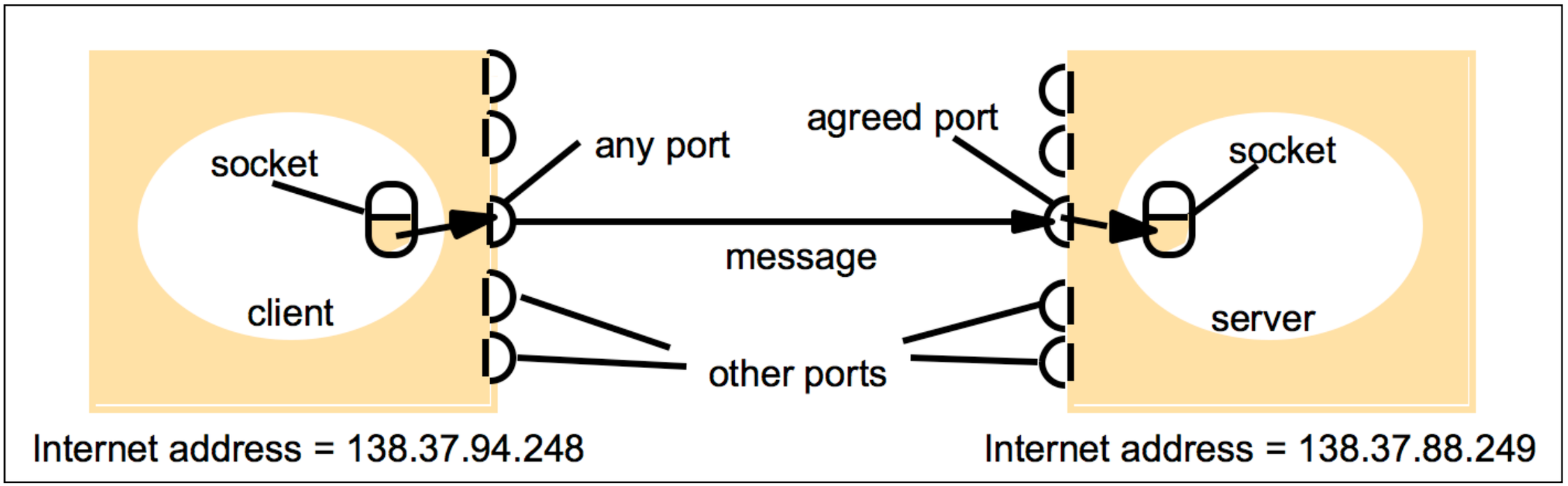
- At the programming level, message destinations can usually be defined by means of the concept of socket

- A socket is an abstraction which provides an endpoint for communication between processes

- A socket address is the combination of an IP address (the location of the computer) and a port (a specific service) into a single identity

- Interprocess communication consists of transmitting a message between a socket in one process and a socket in another process

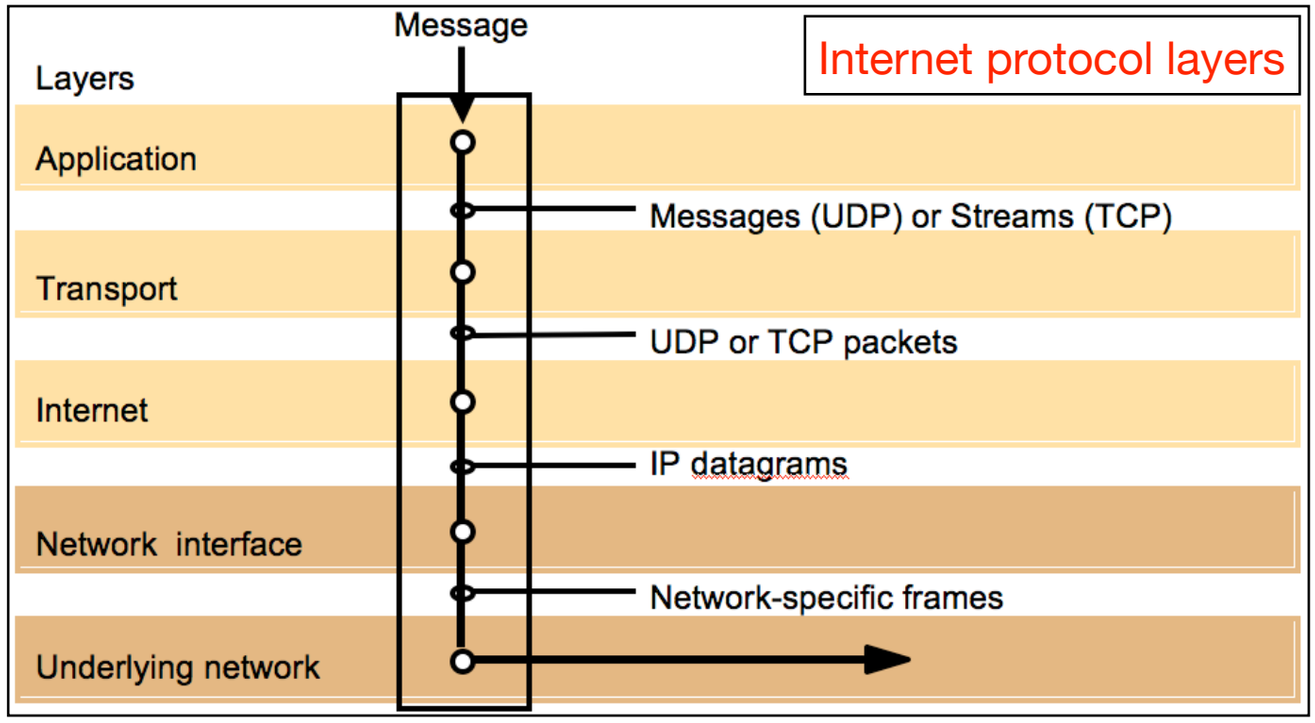
Sockets and Ports

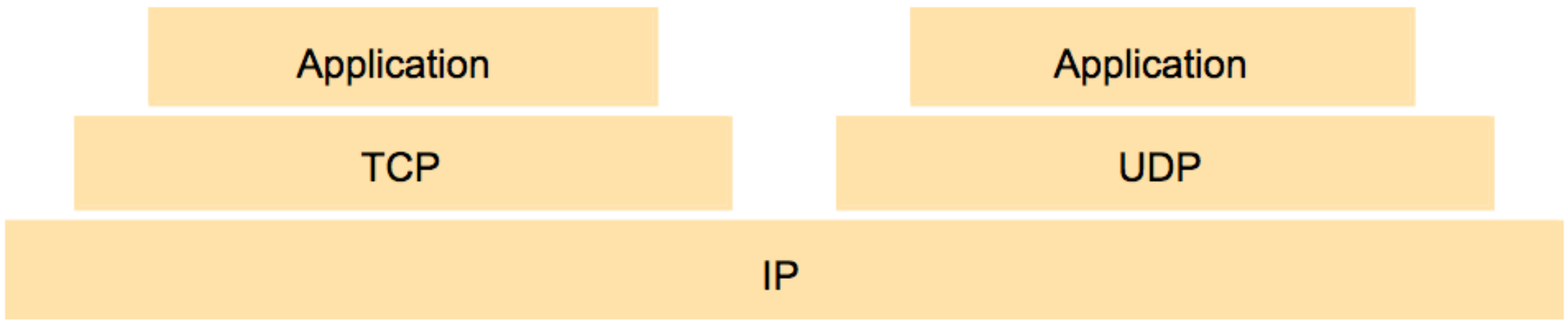


- Messages sent to a particular Internet address and port number can be received only by a process whose socket is associated with that Internet address and port number
- Processes may use the same socket for sending and receiving messages
- Any process may make use of multiple ports to receive messages, BUT a process cannot share ports with other processes on the same computer
- Each socket is associated with a particular protocol, either UDP or TCP

UDP vs TCP in a Nutshell

- TCP (Transport Control Protocol) and UDP (User Datagram Protocol) are two transport protocols
- TCP is a reliable, connection-oriented protocol
- UDP is a connectionless protocol that does not guarantee reliable transmission





UDP Datagram Communication

- A datagram is an independent, self-contained message sent over the network whose arrival, arrival time, and content are **not** guaranteed

- A datagram sent by UDP is transmitted from a sending process to a receiving process without acknowledgement or retries

- If a failure occurs, the message may not arrive

- Use of UDP: for some applications, it is acceptable to use a service that is liable to occasional omission failures
  - DNS (Domain Name Service), which looks up DNS names in the Internet, is implemented over UDP
  - VOIP (Voice Over IP) also runs over UDP

Case Study: JAVA API for UDP Datagrams

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

DatagramPacket Class

- This class provides a constructor that makes 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

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

sender

```
...
byte [] m = args[0].getBytes();
InetAddress aHost = InetAddress.getByName(args[1]);
int serverPort = 6789;
DatagramPacket request = new DatagramPacket(m,  args[0].length(), aHost, serverPort);
...
```

- Instances of DatagramPacket may be transmitted between processes when one process sends it and another receives it

DatagramPacket Class

- The class provides another constructor for use when *receiving a message*
- Its arguments specify an array of bytes in which to receive the message and the length of the array

receiver

```
...
byte[] buffer = new byte[1000];
DatagramPacket request = new DatagramPacket(buffer, buffer.length);
...
```

- A message can be retrieved from DatagramPacket by means of the method getData
- The methods getPort and getAddress access the port and Internet address

receiver

```
...
aSocket.receive(request);
DatagramPacket reply = new DatagramPacket(request.getData(),
        request.getLength(), request.getAddress(), request.getPort());
...
```

DatagramSocket Class

- This class supports sockets for sending and receiving UDP datagrams
- It provides a constructor that takes a port number as argument, for use by processes that need to use a particular local port

```
aSocket = new DatagramSocket(6789);
```

- It also provides a no-argument constructor that allows the system to choose a free local port

```
aSocket = new DatagramSocket();
```

- Main methods of the class:
  - send and receive: for transmitting datagrams between a pair of sockets
  - setSoTimeout: to set a timeout (the receive method will block for the time specified and then trow an InterruptedIOException)

Example: 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,  args[0].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();}
   }
}
```

args[0] is a message
args[1] is a DNS name of the server

how to send a msessage

message converted in array of bytes

IP address of the host

how to receive a message

close the socket

Example: 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();}
    }
}
```

socket bound to the server port 6789

receive the msg

sends back the same message to the client

close the socket

End of the Case Study

TCP Stream Communication

- TCP is a reliable, connection-oriented protocol.
- The API for stream communication assumes that when a pair of processes are establishing a connection, one of them plays the client role and the other plays the server role, but thereafter they could be peers
  - The client role involves creating a **stream socket bound to any port** and then making a connect request asking for a connection to a server at its server port
  - The server role involves creating a **listening socket** bound to a server port and waiting for clients to requests connections
  - When the server accepts a connection, a new stream socket is created for the server to communicate with a client, meanwhile retaining its socket at the server port for listening for connect requests from other clients

TCP Stream Communication

- In other words, the API of the TCP protocol provides the abstraction of a stream of bytes to which data may be written and from which data may be read

- The pair of sockets in client and server are connected by a pair of streams, one in each direction

- Thus each socket has an input stream and an output stream

- A process A can send information to a process B by writing to A's output stream

- A process B obtains the information by reading from B's input stream

Use of TCP

- Many frequently used services run over TCP connections, with reserved port numbers, such as:
  - HTTP (HyperText Transfer Protocol, used for communication between web browsers and web servers)
  - FTP (File Transfer Protocol, it allows directories on a remote computer to be browsed and files to be transferred from one computer to another over a connection)
  - Telnet (it provides access by means of a terminal session to a remote computer)
  - SMTP (Simple Mail Transfer Protocol, used to send email between computers)

Case Study: JAVA API for TCP streams

The Java API provides TCP stream communication by means of two classes: ServerSocket and Socket

ServerSocket Class

- Used 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 of messages, or if the queue is empty, it blocks until one arrives

- The result of executing accept is an instance of the class Socket - a socket for giving access to streams for communicating with the client

receiver

```
int serverPort = 7896;
ServerSocket listenSocket = new ServerSocket(serverPort);
while(true) {
        Socket clientSocket = listenSocket.accept();
        EchoThread c = new EchoThread(clientSocket);
}
```

Socket Class

- The client uses a constructor to create a socket, specifying the hostname and port of a server.

- This constructor not only creates a socket associated with a local port but also *connects* it to the specified remote computer and port number

```
...                                                    sender
int serverPort = 7896;
s = new Socket(args[1], serverPort);
...
```

- It can throw an UnkownHostException if the hostname is wrong or an IOException if an IO error occurs

- The class provides methods getInputStream and getOutputStream for accessing the two streams associated with a socket

Example: 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());}}
        }
}
```

input & output streams

socket bound to hostname and server port 7896

send message & wait for reply (write to output stream & read from input stream)

close the socket

Example: 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();
                    EchoThread c = new EchoThread(clientSocket);
                }
            } catch(IOException e) {System.out.println("Listen :" + e.getMessage());}
    }
}

// Java code continues on the next slide
```

server socket on port 7896

server listens for *connect* requests

When a *connect* request arrives, server makes a new thread in which to communicate with the client.

Example: TCP Server Makes a Connection for Each Client and Then Echoes the Client's Request

```
class EchoThread extends Thread {
        DataInputStream in;
        DataOutputStream out;
        Socket clientSocket;
        public EchoThread (Socket aClientSocket) {
            try {
                    clientSocket = aClientSocket;
                    in = new DataInputStream( clientSocket.getInputStream());
                    out =new DataOutputStream( clientSocket.getOutputStream());
                    this.start();
             } catch(IOException e)  {System.out.println(EchoThread:"+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*/}}
        }
}
```

socket's input and output streams

thread waits to read a msg and writes it back

close the socket

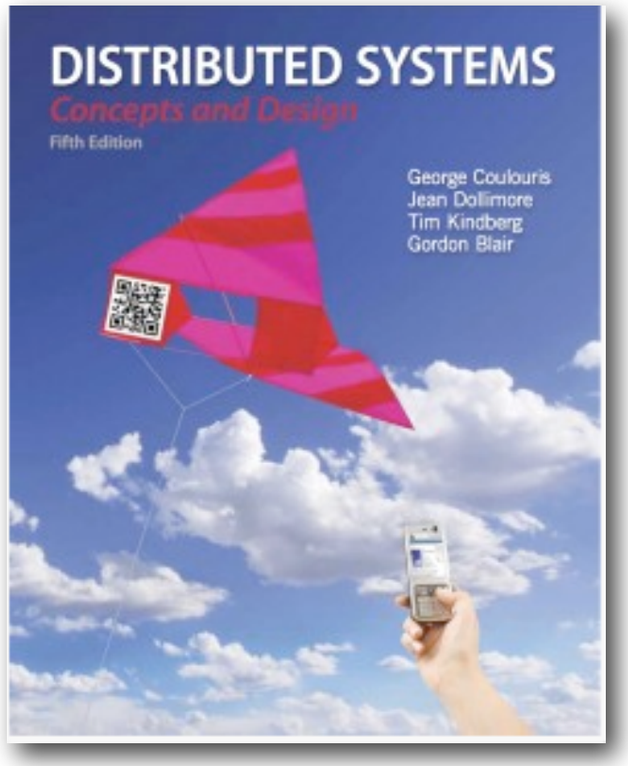
End of the Case Study

Closing a Socket

- When an application closes a socket: it will not write anymore data to its output stream
  - Any data in the output buffer is sent to the other end of the stream and out in the queue at the destination socket with an indication that the stream is broken
- When a process has closed its socket, it will no longer able to use its input and output streams
- The process at the destination can read the data in its queue, but any further reads after the queue is empty will result in an error/exception (for instance, EOFException in Java)
- When a process exits or fails, all of its sockets are eventually closed
- Attempts to use a closed socket or to write to a broken stream results in an error/exception (for instance, IOException in Java)

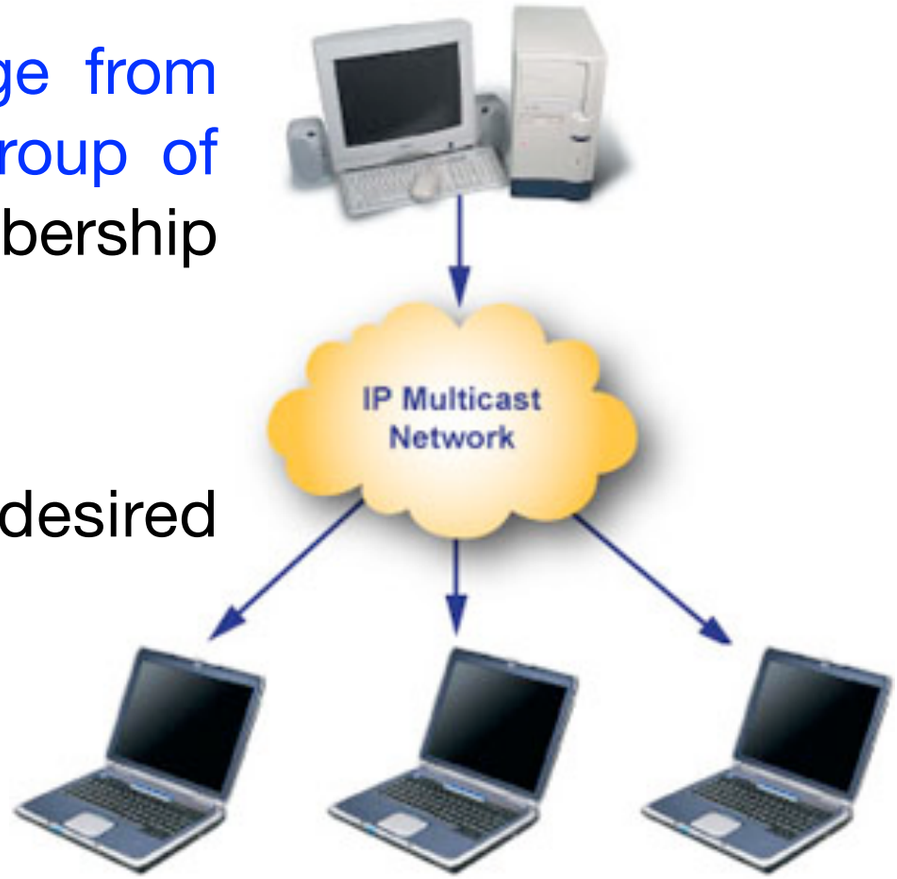
Communication Paradigms

1. Point-to-point Communication
   - Characteristics of Interprocess Communication
   - Sockets
   - Client-Server Communication over UDP and TCP
2. Group (Multicast) Communication
3. Point-to-point Communication
   - Remote Invocation



Multicast

- A multicast operation 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



- There is a range of possibilities in the desired behaviour of a multicast

- The simplest provides no guarantees about message delivery or ordering (see lecture in week 12 on “Multicast Communication”)

What Can Multicast Be Useful for?

- Multicast messages provides a useful infrastructure for constructing distributed systems with the following characteristics:

1. Fault tolerance based on replicated services

   - A replicated service consists of a group of members
   - Client requests are multicast to all the members of the group, each of which performs an identical operation
   - Even when some of the members fail, clients can still be served

What Can Multicast Be Useful for?

- Multicast messages provides a useful infrastructure for constructing distributed systems with the following characteristics:

2. Better performance through replicated data

  - Data are replicated to increase the performance of a service - in some cases replicas of the data are placed in users' computers

  - Each time the data changes, the new value is multicast to the processes managing the replicas

What Can Multicast Be Useful for?

- Multicast messages provides a useful infrastructure for constructing distributed systems with the following characteristics:

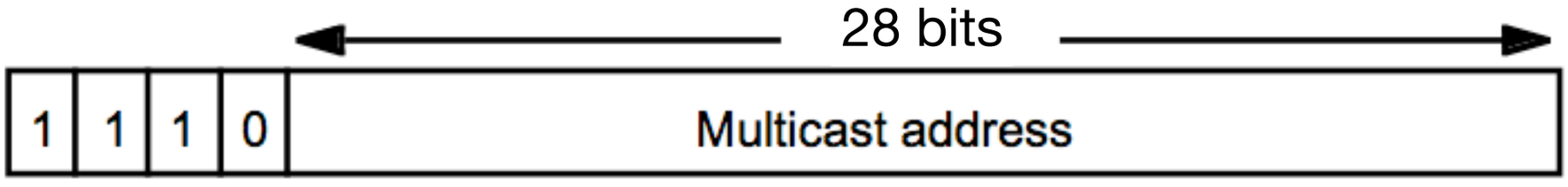
3. Propagation of event notifications

  - Multicast to a group may be used to notify processes when something happens

  - For example, a news system might notify interested users when a new message has been posted on a particular newsgroup

IP Multicast

- IP multicast is built on top of the Internet Protocol, IP
- Note that IP packets are addressed to computers (*ports* belong to the TCP and UDP levels)
- 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 identities of the individual recipients and of the size of the group
- A multicast group is specified by an Internet address whose first 4 bits are 1110 (in IPv4)



IP Multicast - Membership

- Being a member of a multicast group allows a computer to ***receive*** IP packets sent to the group
- It is possible to ***send*** datagrams to a multicast group without being a member
- The membership of multicast groups is dynamic, allowing computers to join or leave at any time and to join an arbitrary number of groups

IP Multicast - IP Level

- At the IP level:
  - A computer belongs to a multicast group when one or more of its processes has sockets that belong to that group
  - When a multicast message arrives at a device:

    copies are forwarded to all of the local sockets that have joined the specified multicast address and are bound to the specified port number

IP Multicast - Programming Level

- At the application programming level, IP multicast is available only via UDP:
  - An application program performs multicasts by sending UDP datagrams with multicast addresses and ordinary port numbers
  - An application program can join a multicast group by making its socket join the group, enabling it to receive messages to the group

Case Study: JAVA API for IP Multicast

The Java API provides a datagram interface to IP multicast through the class MulticastSocket

The Class MulticastSocket

- A subclass of DatagramSocket with the additional capability of being able to join multicast groups

- It provides two alternative constructors, allowing sockets to be created to use either a specified local port or any free local port

```
...
MulticastSocket s =null;
s = new MulticastSocket(6789);
...
```

Joining a Group

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

```
...
MulticastSocket s =null;
s = new MulticastSocket(6789);
InetAddress group = InetAddress.getByName(args[1]);
s.joinGroup(group);
...
```

Leaving a Group

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

```
...
MulticastSocket s =null;
InetAddress group = InetAddress.getByName(args[1]);
s = new MulticastSocket(6789);
...
s.leaveGroup(group);
```

Example:
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);


        // this figure continued on the next slide
```

args[0] = msg contents
args[1] = multicast address

MulticastSocket creation on port 6789

joining a multicast group

sending a DatagramPacket message

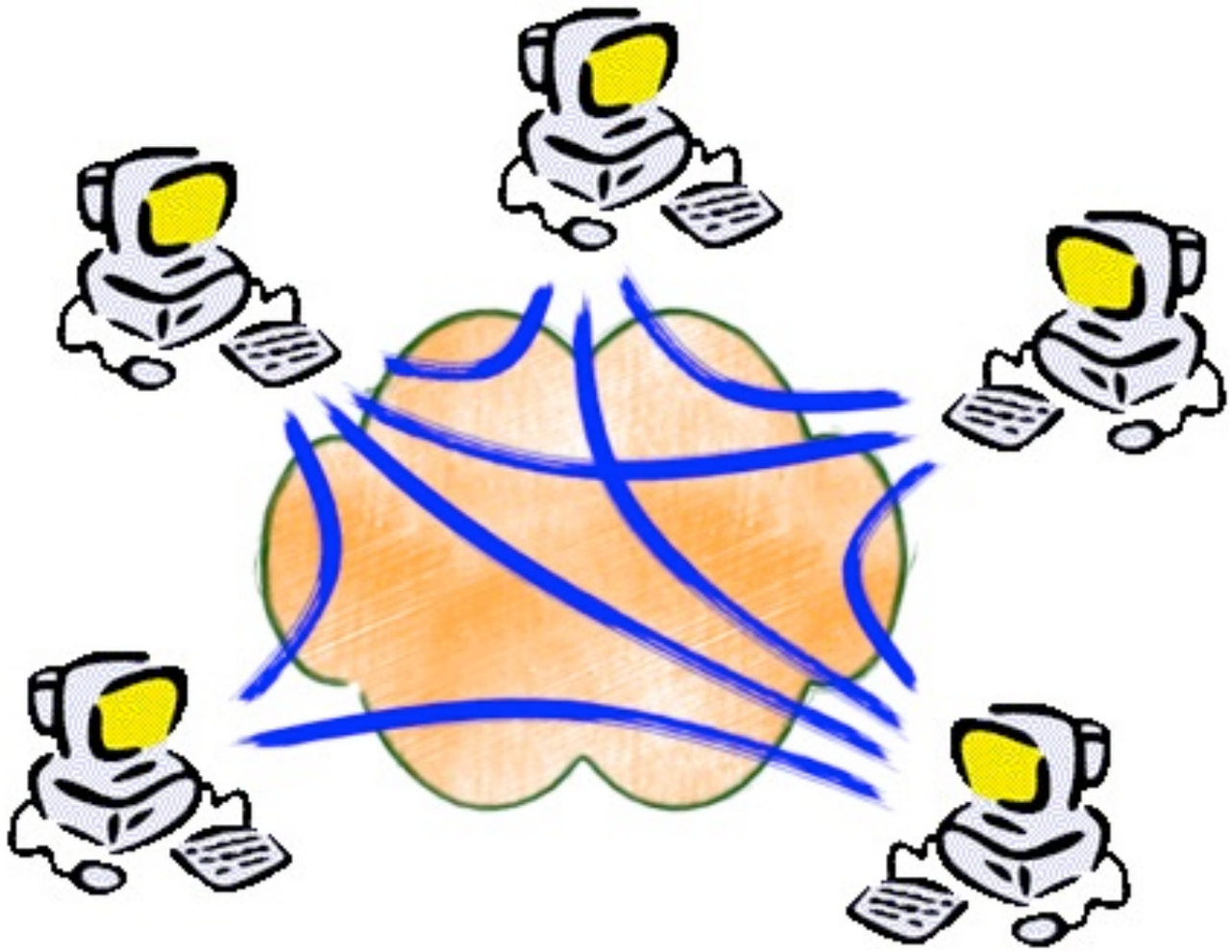
Example:
Multicast Peer Joins a Group and Sends and Receives Datagrams

peer attempts to receive 3 multicast messages from its peers via its socket

```
            // 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();}
    }
}
```

Example:
Multicast Peer Joins a Group and Sends and Receives Datagrams

- When several instances of this program are run simultaneously on different computers, all of them join the same group and each of them should receive its own message and the messages from that joined after it.



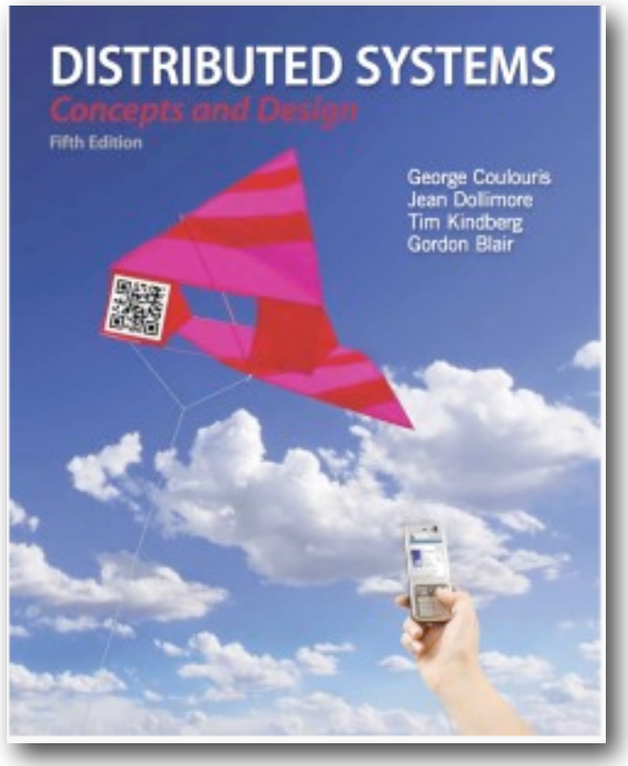
End of the Case Study

DTU Compute
Department of Applied Mathematics and Computer Science

Communication Paradigms

Nicola Dragoni
Embedded Systems Engineering
DTU Compute

1. Point-to-point Communication
   - Characteristics of Interprocess Communication
   - Sockets
   - Client-Server Communication over UDP and TCP
2. Group (Multicast) Communication
3. Point-to-point Communication
   - Remote Invocation
     - Remote Method Invocation (RMI)
     - RMI Invocation Semantics



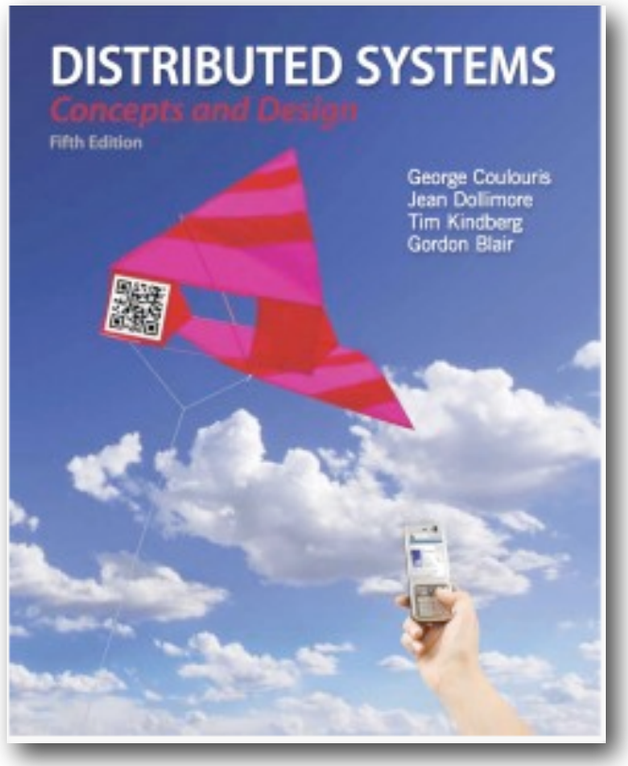
Remote Invocation

- RPC (Remote Procedure Call)
  - the earliest programming model for distributed programming:

    A.D. Birrell and B.J. Nelson. **Implementing remote procedure calls**. *ACM Transactions on Computer Systems*, 2(1), pp. 39-59, 1984

    - *allows client programs to call **procedures** in server programs running in separate processes (and generally in different computers from the client)*
- RMI (Remote Method Invocation)
  - extension of local method invocation of object-oriented programming
  - *allows an object living in one process to invoke the **methods of an object** living in another process*. Most famous example: Java RMI

Remote Method Invocation (RMI)



Let Us Start from Scratch: the Object Model (...in 2 slides...)

- An object-oriented program (Java, C++, ...) consists of a collection of interacting objects, each of which consists of a set of data and a set of methods

- An object can communicate with other objects by invoking their methods, generally passing arguments and receiving results (**request/reply protocol**)

- Objects can encapsulate their data and the code of their methods

- Some languages (JAVA, C++) allow programmers to define objects whose instance variables can be accessed directly

- **BUT in a distributed object system, an object's data should be accessible only via its methods (or interface)**

Actions in the Object Model

- An **action** in an object-oriented program is initiated by an object invoking a method in another object

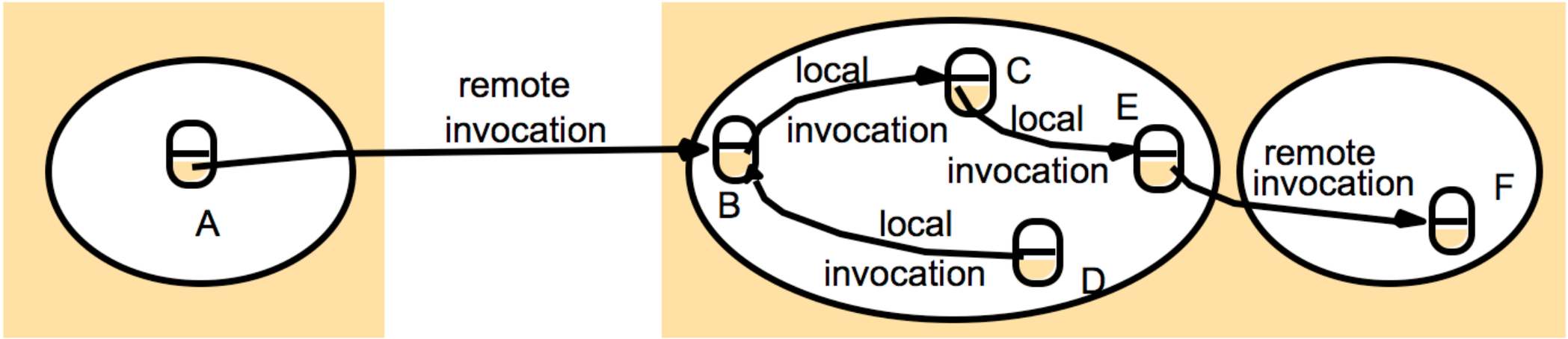
- The receiving object executes the appropriate method and then returns control to the invoking object, sometimes supplying a result

- An invocation of a method can have 3 possible effects:
  - the state of the receiver may be changed
  - a new object may be instantiated (i.e., by using a constructor in Java)
  - further invocations on methods in other objects may take place

How to extend the “traditional” object model to make it applicable to distributed systems?

The Distributed Object Model

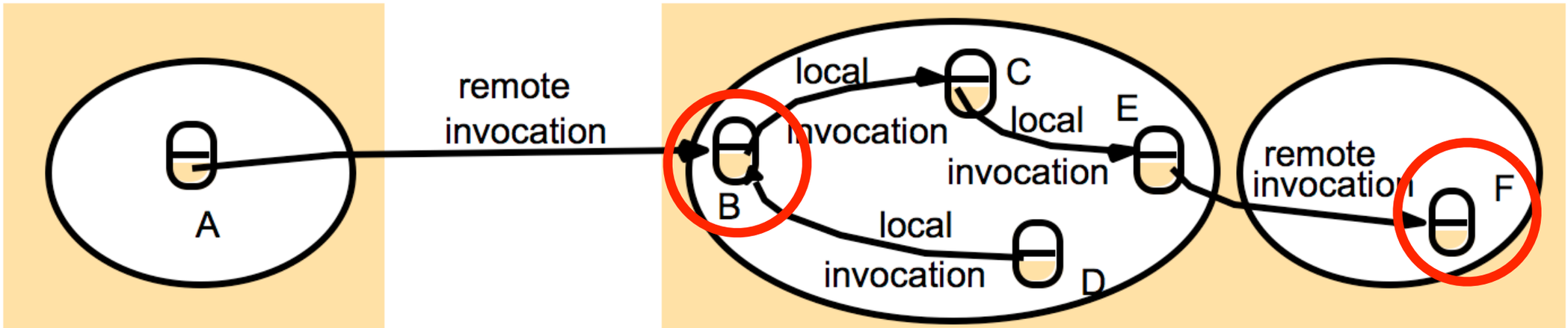
- Each process contains a collection of objects
  - some of which can receive both local and remote invocations
  - whereas the other objects can receive only local invocations



- 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 Objects

- Remote objects: objects that can receive remote invocations



- **Fundamental concepts** of the distributed object model:
  - [**Remote Object References**] other objects can invoke the methods of a remote object if they have access to its ***remote object reference***
  - [**Remote Interfaces**] every remote object has a ***remote interface*** that specifies which of its methods can be invoked remotely

Remote Object Reference

- A remote object reference is an identifier that can be used throughout a distributed system to refer to a particular *unique* remote object

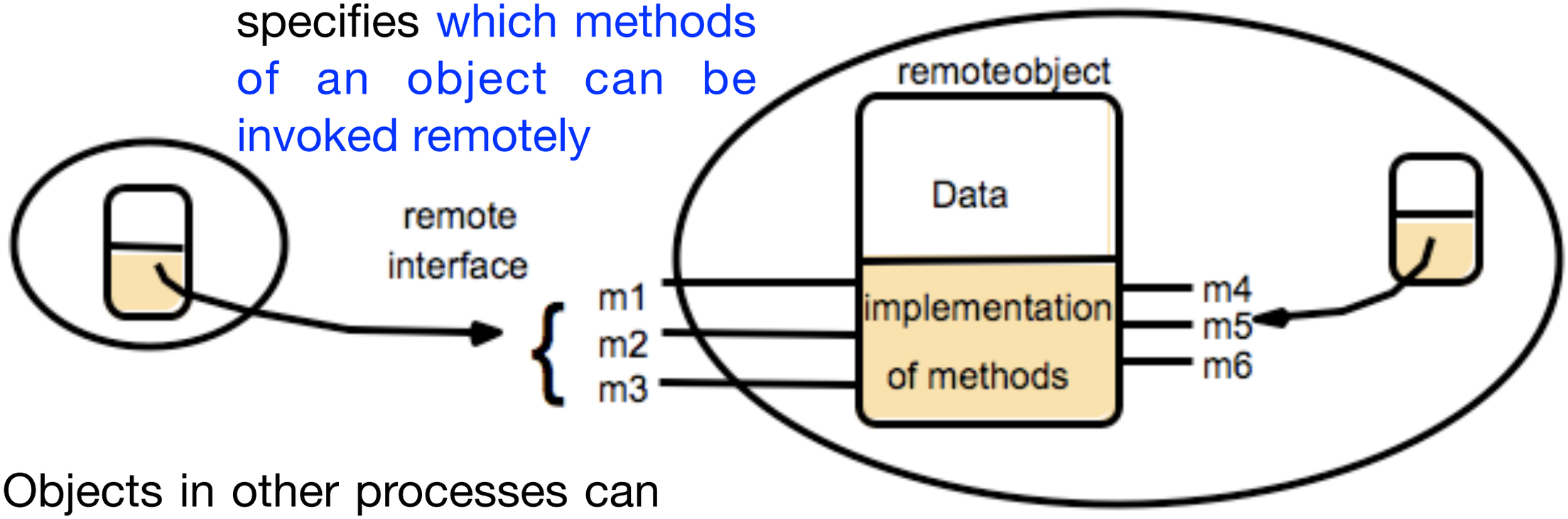
- A remote object reference is passed in the invocation message to specify which object is to be invoked

- Remote object references are analogous to local ones in that:
  - 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

Remote Interface

The remote interface specifies which methods of an object can be invoked remotely

The class of a remote object implements the methods of its remote interface



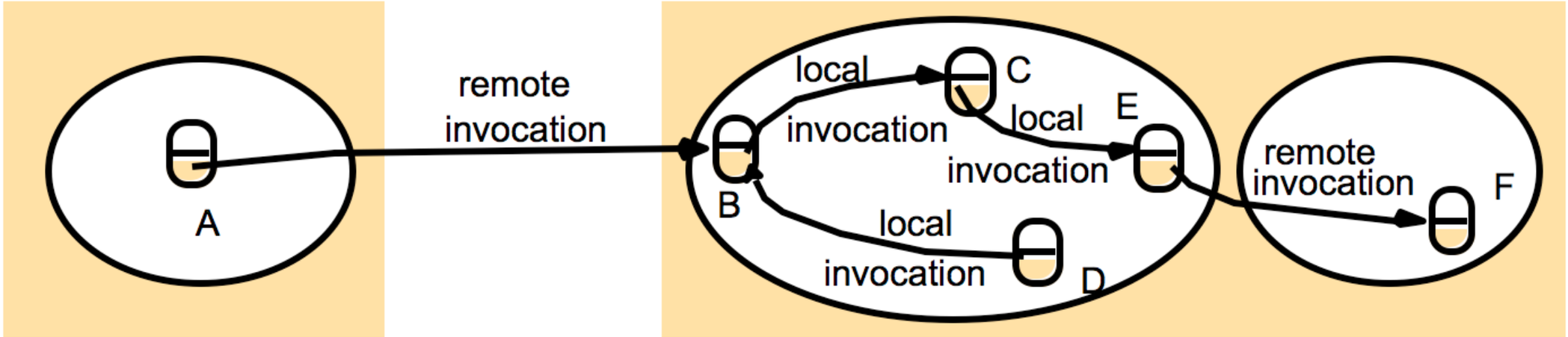
Objects in other processes can invoke *only* the methods that belong to the remote interface of a remote object

Local objects can invoke the methods in the remote interface as well as other methods implemented by a remote object

Actions... in a *Distributed* Object System

- As in the non-distributed case: an action is initiated by a method invocation, which may result in further invocations on methods in other objects

- BUT in the distributed case: the objects involved in a chain of related invocations may be located in different processes or different devices

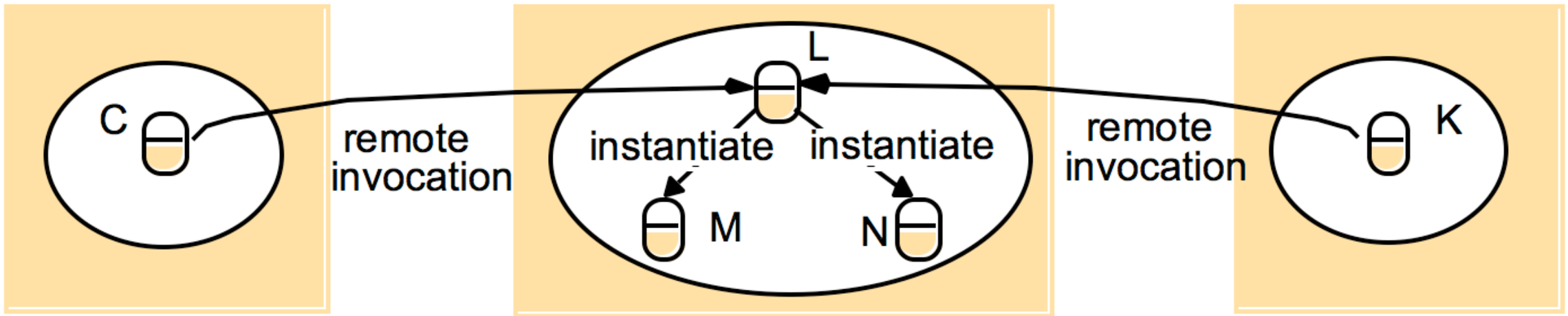


- When an invocation crosses the boundary of a process or computer, RMI is used and the remote reference of the object must be available to the invoker

- Remote object references may be obtained as the *results of remote method invocations* (example: A might obtain a remote reference to F from B)

Creation of Remote Objects

- When an action leads to the instantiation of a new object, that **new object will normally live within the process where the instantiation is requested**



- If a newly instantiated object has a remote interface, it will be a *remote object* with a *remote object reference*

Exceptions

- Any remote invocation may fail for reasons related to the invoked object being in a different process or computer from the invoker

  Example: 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

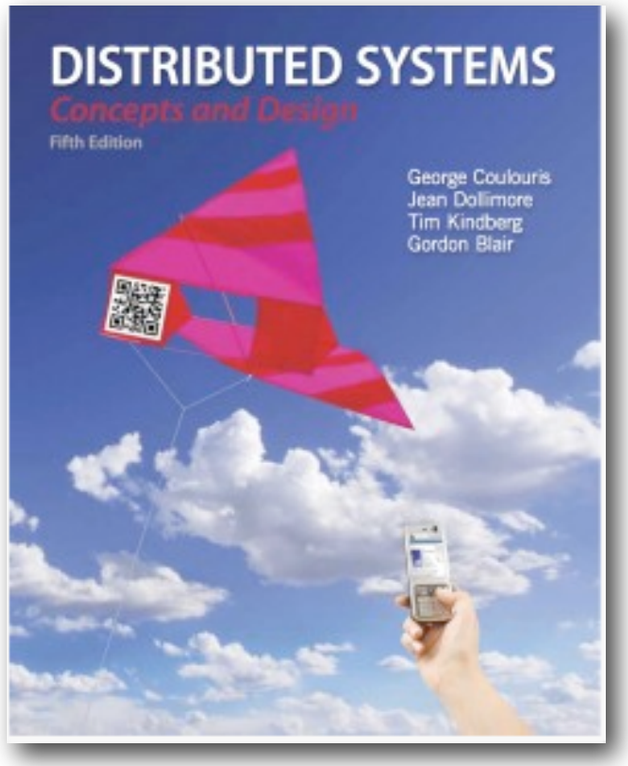
- Remote method invocation should be able to raise exceptions!

  - Timeouts that are due to distribution

  - Exceptions raised during the execution of the method invoked:

    - attempt to read beyond the end of a file
    - attempt to access a file without the correct permissions
    - ...

RMI Invocation Semantics



Local Method Invocation Semantics

- Local method invocations are executed **exactly once**

  exactly once invocation semantics = every method is executed exactly once

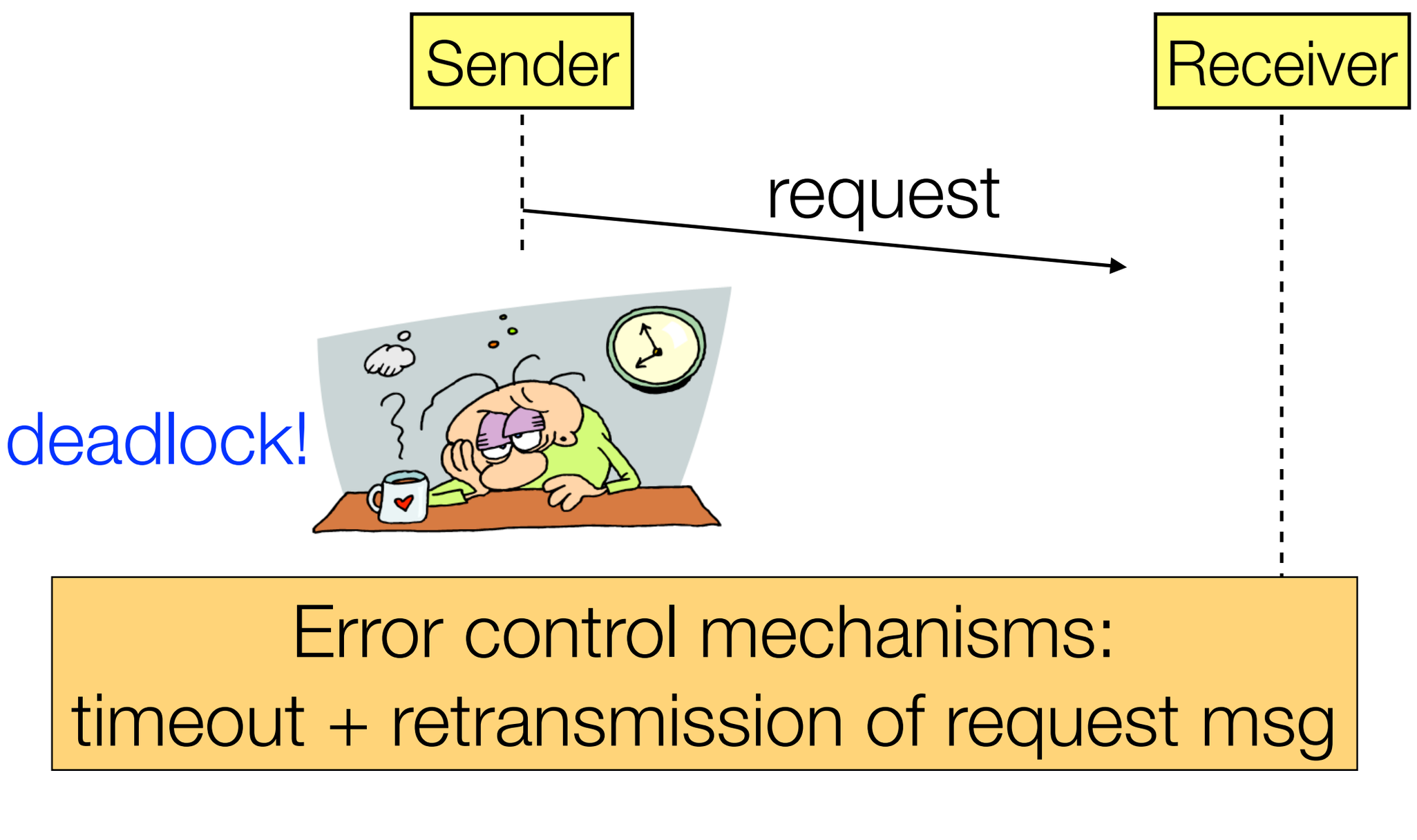
- This cannot always be the case for remote method invocation!

- Request-reply protocols, such as RMI, can be implemented in different ways to provide different delivery guarantees

- These choices lead to a variety of possible semantics for the reliability of remote invocations as seen by the invoker

Main Design Choices for Implementing RMI

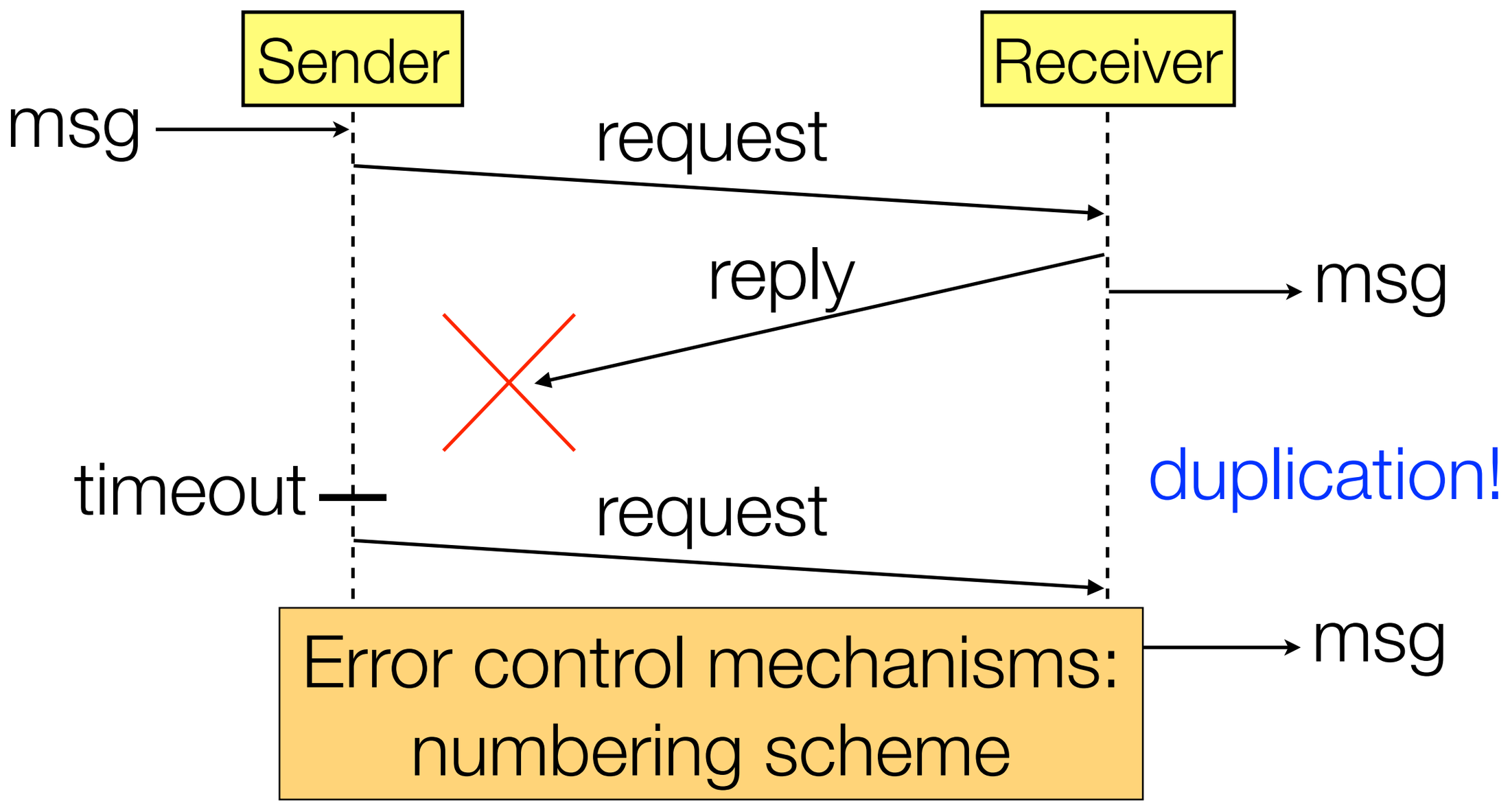
- Retry request message: whether to retransmit the request message until either a reply is received or the server is assumed to have failed



Error control mechanisms:
timeout + retransmission of request msg

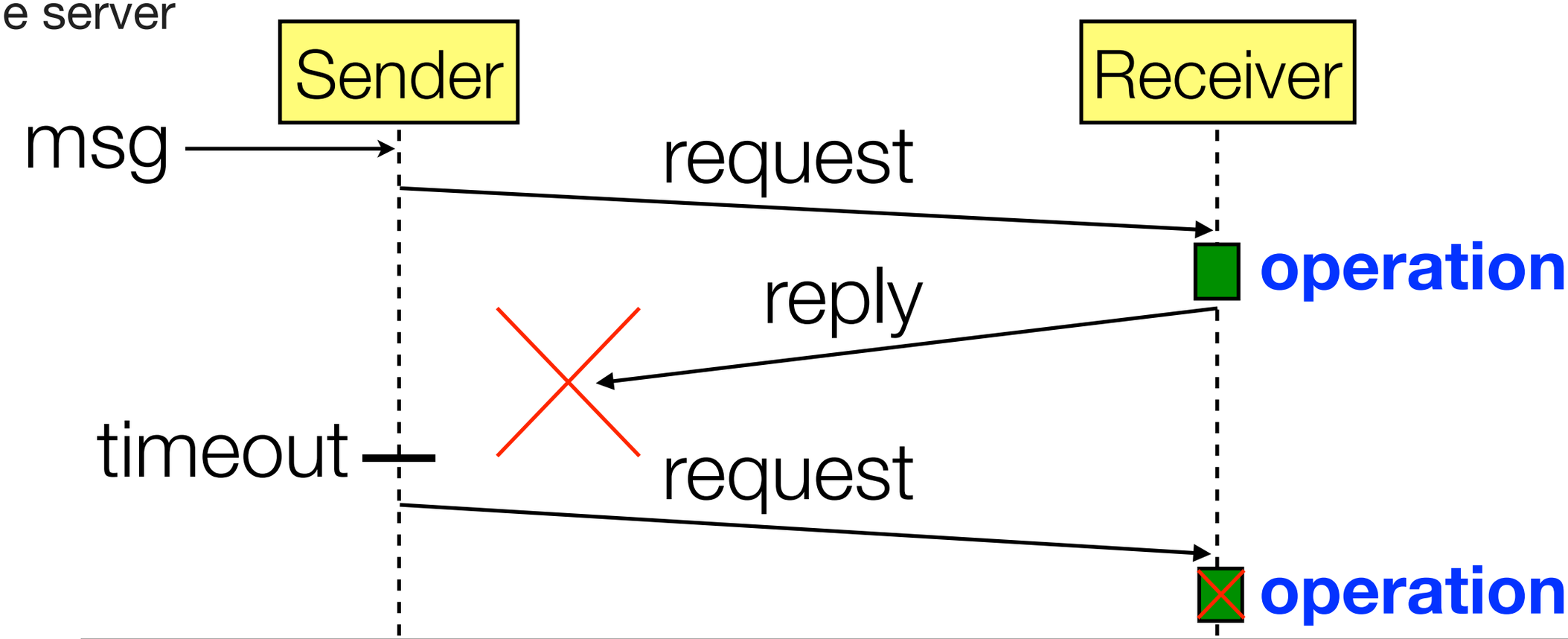
Main Design Choices for Implementing RMI

- Duplicate filtering: when retransmissions are used, whether to filter out duplicate requests at the server



Main Design Choices for Implementing RMI

- Retransmission of results: whether to keep a history of result messages to enable lost results to be retransmitted without re-executing the operations at the server



Error control mechanisms:
numbering scheme + history of result msgs

Main Design Choices for Implementing RMI

- Combination of these choices lead to a variety of possible **semantics for the reliability of remote invocations**: **Maybe**, **At-least-once**, **At-most-once**

Retry request message

Duplicate filtering

+/-?

Retransmission of results

RMI Invocation Semantics: Maybe

- The remote method may be executed once or not at all
- Maybe semantics arises when no fault tolerance measures are applied
- Useful only for applications in which *occasional* failed invocations are acceptable
- This model can suffer from the following types of failure:
  - omission failures if the invocation or result message is lost
  - crash failures when the server containing the remote object fails

RMI Invocation Semantics: At-Least-Once

- 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 be achieved by the retrasmission of request messages, masking the omission failures of the invocation or result message

- This model can suffer from the following types of failure:
  - crash failures when the server containing the remote object fails
  - arbitrary failures, in cases when the invocation message is retransmitted, the remote object may receive it and execute the method more than once, possibly causing wrong values to be stored or returned

RMI Invocation Semantics: At-Most-Once

- 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, in which case the method will have been executed either once or not at all

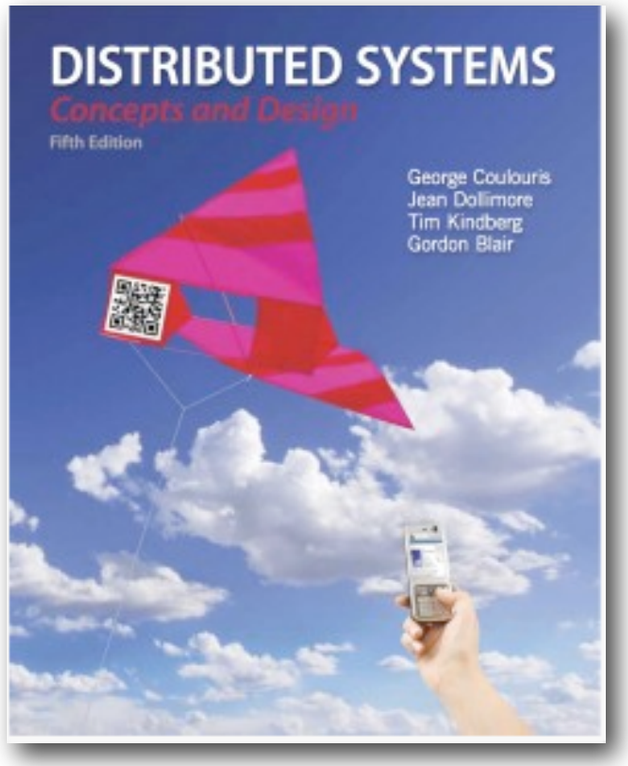
- Can be achieved by using a combination of fault tolerance measures (retransmission + duplicate filtering)
  - The use of retries masks any omission failures of the invocation or result messages
  - Arbitrary failures are prevented by ensuring that for each RMI a method is never executed more than once

RMI Invocation Semantics Summary

Fault tolerance measures			*Invocation semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*
Yes	No	Re-execute procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

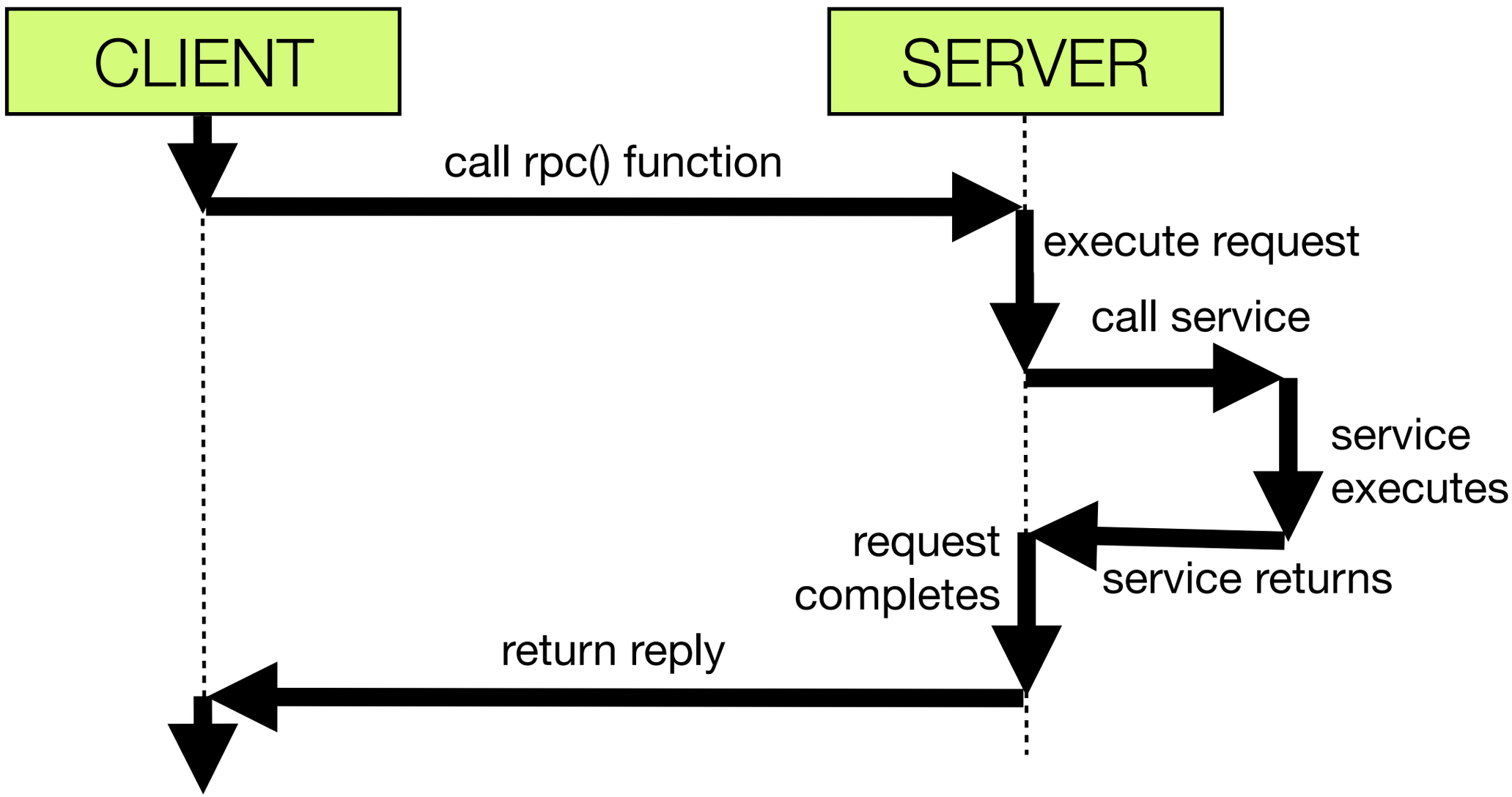
- In Java RMI the invocation semantics is *at-most-once*
- In CORBA is *at-most-once* but *maybe* semantics can be requested for methods that do not return results

Remote Procedure Call (RPC)



RPC (... in one slide...)

- RPC (Remote Procedure Call): *allows client programs to call procedures in server programs running in separate processes and generally in different computers from the client*



RPC vs RMI?

- A remote procedure call is very similar to a RMI in that a client program calls a procedure in another program running in a server process

- Server may be clients of other servers to allow chains of RPCs

- A server process must define in its service interface the procedures that are available for calling remotely

- RPC, like RMI, may be implemented to have one of the choices of invocation semantics previously discussed (maybe, at-least-one, at-most-one)