Distributed Systems

3. Interprocess Communication

Werner Nutt

Interprocess Communication

3.1 Principles

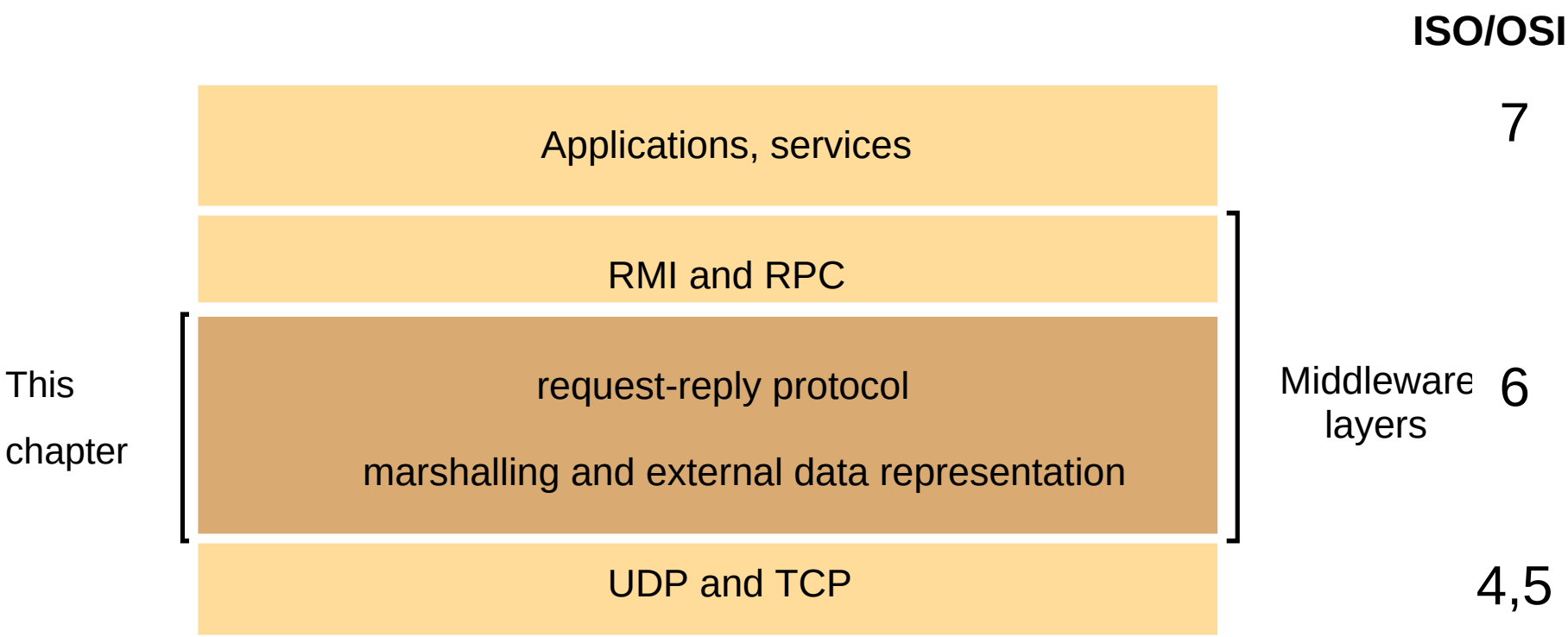
1. **Principles**
2. APIs for UDP and TCP
3. External Data Representation
4. Client Server Communication
5. Group Communication

Middleware

- Middleware offers an infrastructure that enables application processes to communicate with each other
- Processes issue requests to the transportation layer

  (i.e., the application takes the initiative, not the middleware)
- Applications access the middleware via APIs, e.g.,
  - creation and manipulation of sockets
- Integration into programming languages
  - remote procedure call (RPC)
  - remote method invocation (RMI)
- For higher level APIs, data has to be transformed before it can be shipped ("data marshalling")
- Protocols for Client/Server Interaction ("Request/Reply")

Middleware Layers



Characteristics of IPC

- Message Passing Primitives: Send, Receive
- Message = <Destination, Content>
- Destination = <Network address, Port>
  - Port = destination within a host that identifies a receiving process
  - Ports are uniquely identified by their port number
  - Hosts are uniquely identified ... *(or not?)*

Assigned Port Numbers

FTP Data	20
FTP Control	21
SSH	22
Telnet	23
SMTP	25
Domain Name Server	42
Whois	43
HTTP	80
POP3	110
IMAP4	143
BGP	179
HTTPS	443
IMAP4 over SSL	993

Assigned by IANA
(= Internet Assigned
Numbers Authority)

Numbers between 0 and 1023 are "well-known" ports — opening a port for such numbers requires privileges

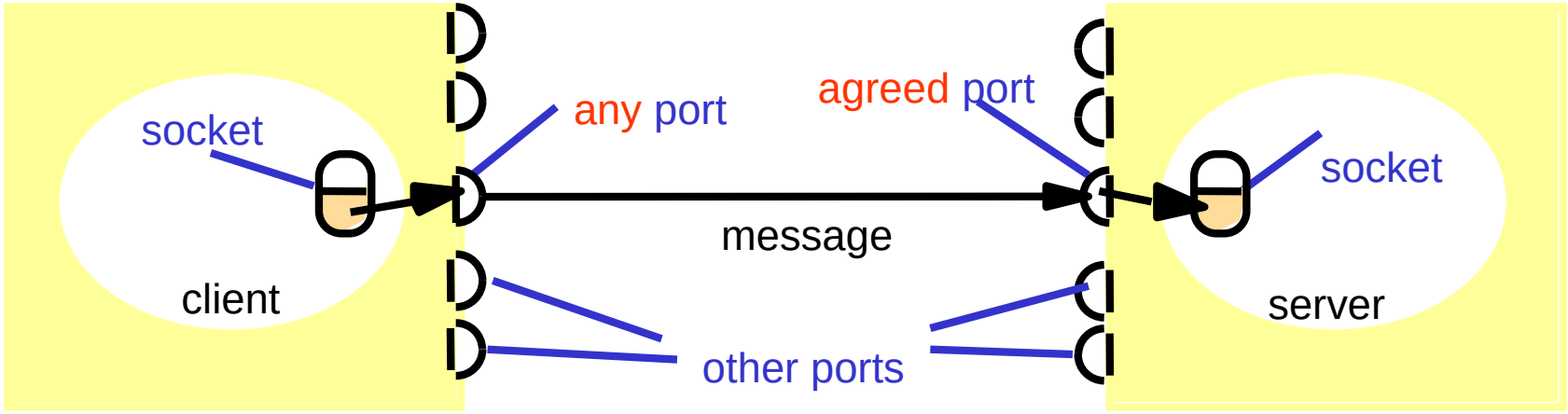
can be found
- on the Web
- in "/etc/services"
under Linux and MAC/OS

Sockets

- Characteristics
  - Endpoint for inter-process communication
  - Message transmission between sockets
  - A socket is associated with either UDP or TCP
  - Sockets are bound to ports
  - One process can use many ports
  - Processes don't share sockets *(unless for IP multicast)*

- Implementations
  - originally BSD Unix, but available in Linux, Windows,…
  - APIs in programming languages (e.g., **`java.net`**)

Sockets and Ports



Internet address = 138.37.94.248

Internet address = 138.37.88.249

Socket = Internet address + port number
Only one receiver but many senders per port
Advantage: several points of entry to process
Disadvantage: location dependence

Communication Primitives

- Send
  - send a message to a socket associated to a process
  - can be blocking or non-blocking
- Receive
  - receive a message on a socket
  - can be blocking or non-blocking
- Broadcast/Multicast
  - send to all processes/all processes in a group

Receive

- Receive is usually blocking
  - destination process blocked until message arrives
  - most common case

- Variations
  - conditional receive

    (continue until receiving indication that message arrived or finding out by polling)
  - timeout
  - selective receive

    (wait for message from one of a number of ports)

Send in Asynchronous Communication

- Characteristics
  - non-blocking *(process continues after the message sent out)*
  - buffering needed *(at receive end)*
  - mostly used with blocking receive
  - efficient implementation

- Problems
  - buffer overflow
  - error reporting *(difficult to match error with message)*

Maps closely onto connectionless service

Send in Synchronous Communication

- Characteristics
  - blocking *(sender suspended until message received)*
  - synchronisation point for sender & receiver
  - easier to understand

- Problems
  - failure and indefinite delay causes indefinite blocking *(use timeout)*
  - multicasting/broadcasting not supported
  - implementation more complex

Maps closely onto connection-oriented service

Interprocess Communication

3.2 APIs for UDP and TCP

1. Principles
2. **APIs for UDP and TCP**
3. External Data Representation
4. Client Server Communication
5. Group Communication

Java API for Internet Addresses

- Class **`InetAddress`**
  - uses DNS (Domain Name System)

```
InetAddress serverAdd =
 InetAddress.getByName("www.inf.unibz.it");
```

  - throws **`UnknownHostException`**
  - encapsulates details of IP address
    (4 bytes for IPv4 and 16 bytes for IPv6)

UDP Packet Structure

UDP = User Datagram Protocol

16-bit source port number	16-bit destination port number
16-bit UDP packet length	16-bit UDP checksum
Payload	

Java API for UDP

- Simple send/receive
  - with messages possibly lost/out of order

Payload (= array of bytes)	Payload length	Destination IP address	Destination Port

- Class **`DatagramPacket`**
  - packets may be transmitted between sockets
  - packets are truncated if too long
  - provides **`getData`**, **`getPort`**, **`getAddress`**, **`getLength`**

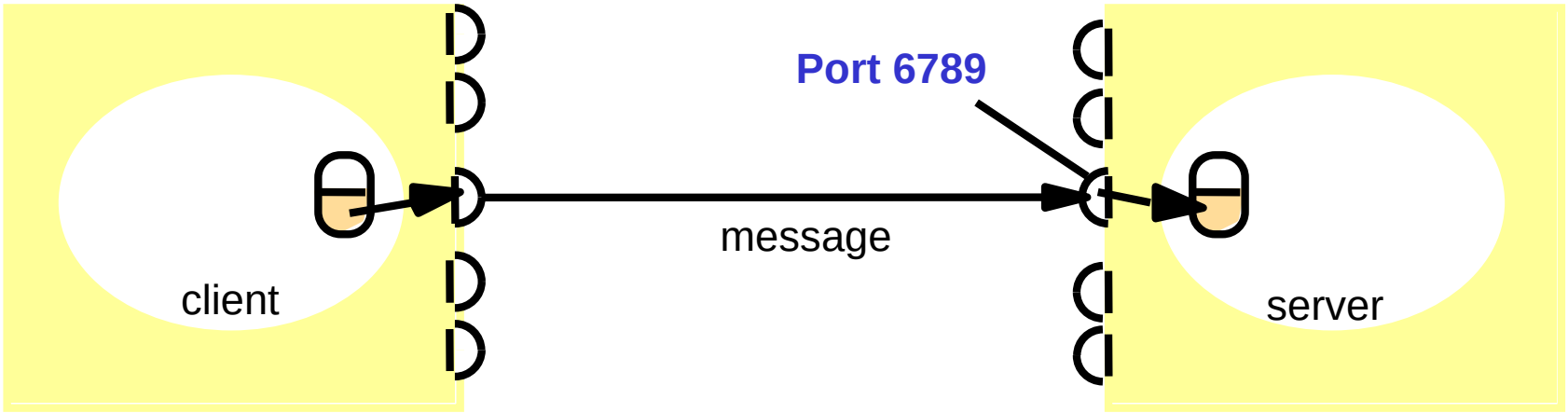
Java API for UDP Sockets

Class **`DatagramSocket`**

- *socket constructor*
  - bound to free port if no arg
  - arguments ***`InetAddress`***, ***`Port`***
- **`send`** a ***`DatagramPacket`***, non-blocking
- **`receive`** ***`DatagramPacket`***, blocking
- **`setSoTimeout`** (receive blocks for time T and throw ***`InterruptedIOException`***)
- **`close`** ***`DatagramSocket`***
- throws **`SocketException`** if port unknown or in use
- **`connect`** and **`disconnect`** (!!??)
- **`setReceiveBufferSize`** and **`setSendBufferSize`**

In the Following Example …

- UDP Client
  - sends a message and gets a reply
- UDP Server
  - repeatedly receives a request and sends it back to the client



See website of textbook for Java code (www.cdk4.net)

UDP Client Example

```
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);
          } 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 Example

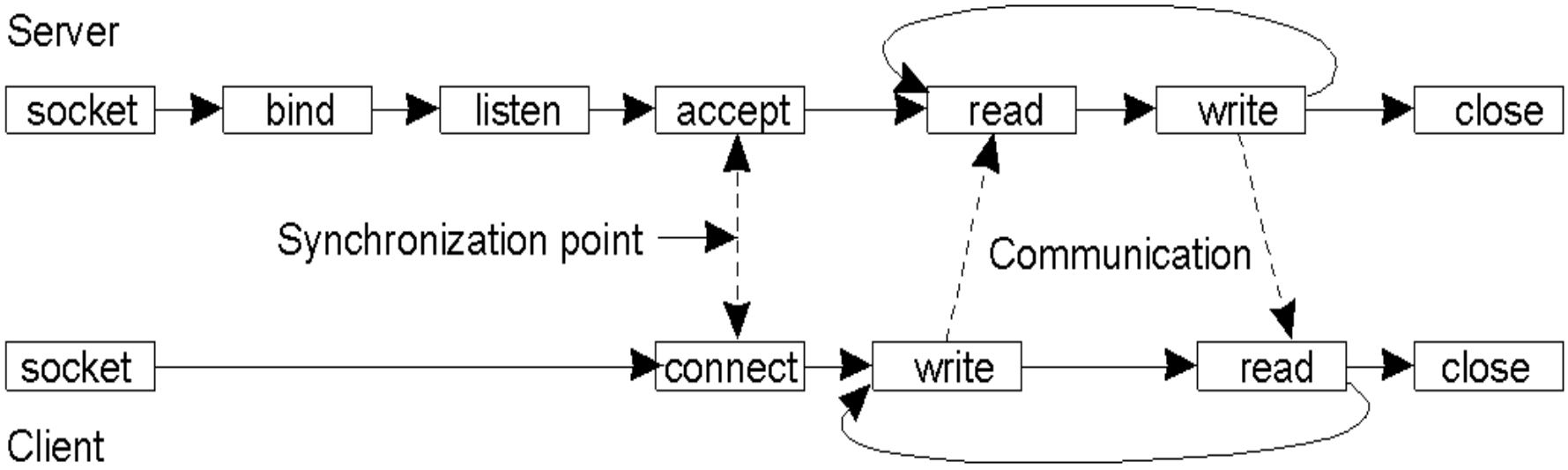
```
public class UDPServer{
    public static void main(String args[]){
        DatagramSocket aSocket = null;
        try {aSocket = new DatagramSocket(6789);
            byte[] buffer = new byte[1000];
            while(true) {
            DatagramPacket request = new DatagramPacket(buffer,
                                                        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 Primitives for TCP/IP

System Calls	Meaning
Socket	Create a new communication endpoint
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection

Sockets appeared first in Berkeley UNIX as an interface to the transport layer

Life Cycle of Berkeley TCP Sockets



Java API for TCP

- Data stream abstraction
  - enables reliable transfer *(send can be blocking)*
  - marshaling/unmarshaling of data
  - access to TCP parameters: ReceiveBufferSize, SendBufferSize
- Classes **`Socket`** and **`ServerSocket`**
  - **`Socket`** asks for connection
  - **`ServerSocket`** listens and returns **`Socket`** when contacted
- Port numbers
  - explicit for **`ServerSocket`**, transparent for **`Socket`**

Java API for TCP

Class **`ServerSocket`**:

- **`bind`** to a **`SocketAddress`** if unbound
- **`accept`**: listen and return a **`Socket`**
  when a connection request arrives (blocking)
- **`close`**

Java API for TCP

Class **`Socket`**:

- **`connect`** to ***`SocketAddress`***
- **`getRemoteSocketAddress`** since that was chosen by the TCP system on the other side
- **`getInputStream`**, **`getOutputStream`**
  - use them for reading and writing
  - which is/may be blocking
- **`DataInputStream`**, **`DataOutputStream`**:
  - wrapper classes for streams
  - have methods for marshaling/ unmarshaling
- **`isConnected`**
- **`close`**

TCP Client Example

```
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
             String data = in.readUTF();
             System.out.println("Received: "+ data) ;
             s.close();
          } 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)….}
}
```

TCP Server Example

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

// this figure continues on the next slide

Example Server (cntd.)

```
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:s a"+e.getMessage());}
        } finally {try {clientSocket.close();}catch (IOException e)…..}
}
```

Interprocess Communication

3.3 External Data Representation

1. Principles
2. APIs for UDP and TCP
3. **External Data Representation**
4. Client Server Communication
5. Group Communication

External Data Representation

The transport layer is only concerned with the (reliable?) transmission of sequences of bytes …

… but what about data types and data structures?

Problems:

- Integers: 1'complement vs. 2'complement
- Real/Float: IEEE 754 standard vs. IBM Mainframes
- Byte order in words: big-endianness vs. little-endianness
- Nested strucs …

Little and Big Endians

Common file formats and their endian order are as follows:

- Little Endan
  - BMP bitmaps (Windows and OS/2 Bitmaps)
  - GIF
  - QTM (Quicktime Movies)
  - Microsoft RTF (Rich Text Format)
- Big Endian
  - Adobe Photoshop
  - JPEG
  - TIFF (actually both, endian identifier encoded into file)
  - MacPaint

Marshalling and Unmarshalling

- Marshalling: Encode data items so that they can be written onto a stream
- Unmarshalling: Read an encoding from a stream and reconstruct the original items

Needed for transmission and storing data in a file

Examples

- CORBA: CDR (= Common Data Representation) for primitive and structured data types that occur in remote method invocations
- Java: Serialization (applicable to all classes that implement the interface **`Serializable`**, uses reflection ➔ *next chapter*)

Example: Marshalling in CORBA

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

IDL declaration of a **Person** struct

Marshalling in CORBA (cntd.)

- Primitive Types
  - short, long, string, float, double, …
  - endian order
    - determined by sender
    - flagged in each message
- Constructed Types
  - marshalling operations are generated from IDL types by CORBA interface compiler

Marshalling in CORBA (cntd.)

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	`1934`	*unsigned long*

The flattened form represents a **`Person`** struct with value: {'Smith', 'London', 1934}

Why can one reconstruct the original struct from this byte sequence?

Example: Serialization in Java

```
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;
        }
        // … methods for accessing instance variables …
}
```

Stating that a class implements "Serializable" makes its instances serializable

Serialization/deserialization process has no knowledge of object types, uses "reflection"

Sketch: Serialization in Java

Person p = new Person("Smith", "London", 1934);

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*
1934	5 Smith	6 London	h1	*values of instance variables*

h0 is a class handle and h1 is an instance handle (i.e., can be used by other serialized objects)

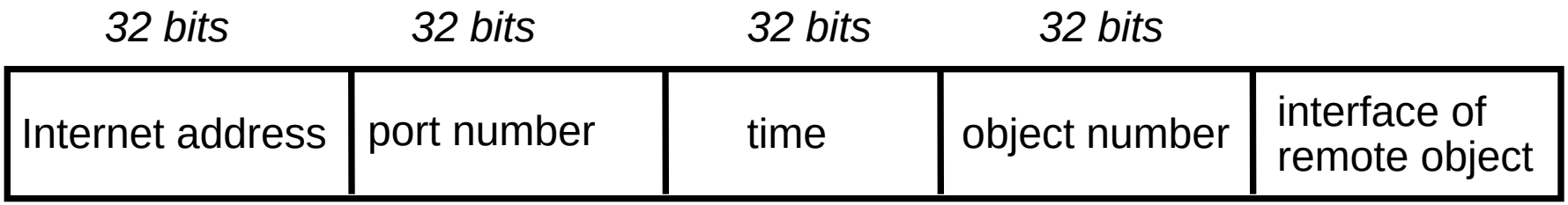
- Wrapper classes: **`ObjectOutputStream, ObjectInputStream`**
- Methods: **`writeObject(`*`Object`*`), readObject()`**

Why does CORBA not mention types and classes, but Java does?

Remote Object References

Remote objects must be uniquely identifiable within a DS to be invoked

(The figure sketches one approach)



- *Why is there time? Is the port number not sufficient?*
- *How well does this scale if objects can migrate between processes?*

Interprocess Communication

3.4 Client Server Communication

1. Principles
2. APIs for UDP and TCP
3. External Data Representation
4. **Client Server Communication**
5. Group Communication

Communication Types

- Asynchronous: sender continues after submission

- Synchronous: sender is blocked until
  - message is stored at receiver's host
  - message is received
  - reply is received

Client Server Communication

Typical example of interprocess communication

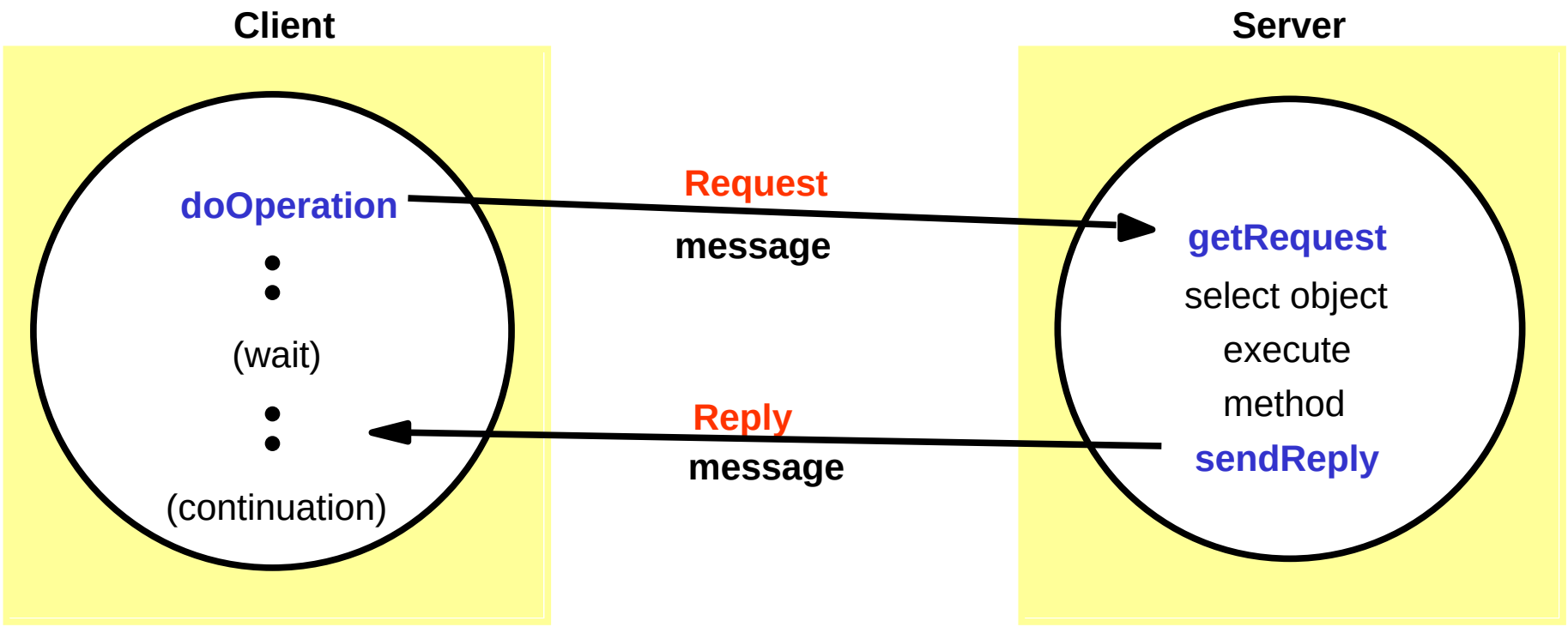
- Based on a request-reply protocol
- Most RPC (= Remote Procedure Call) and RMI (= Remote Method Invocation) systems are supported by a similar protocol at the message level

Should this be synchronous communication or not?

Our toy protocol consists of three primitive operations

- doOperation
- getRequest
- sendReply

Client Server Communication



Client Server Communication

public byte[] doOperation (RemoteObjectRef o, int methodId, byte[] arguments)

- sends a request message to the remote object and returns the reply
- arguments specify the remote object, the method to be invoked and the arguments of that method

Client Side

public byte[] getRequest ();

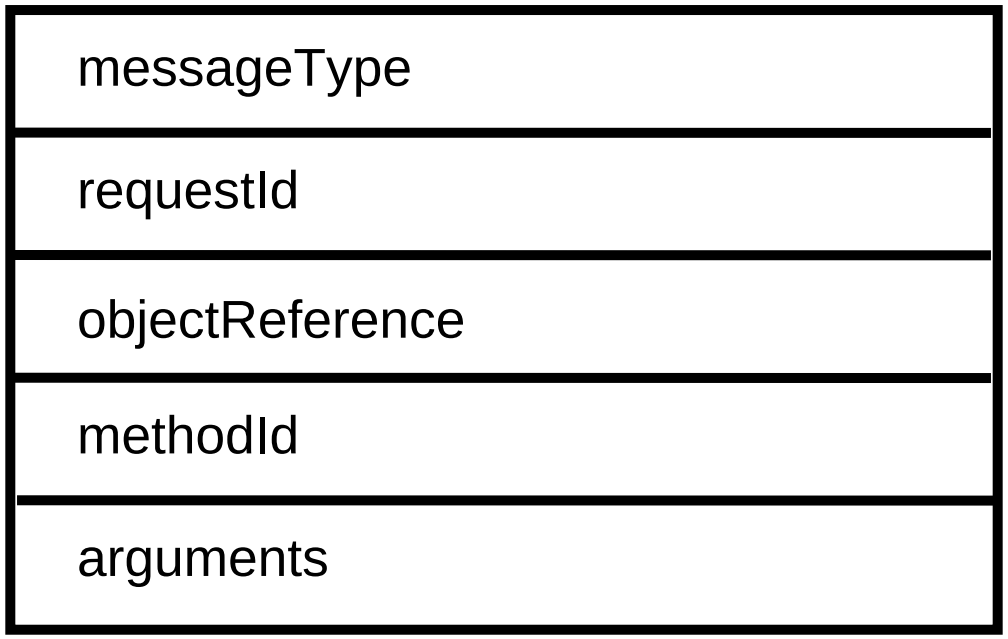
- acquires a client request via the server port

public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);

- sends the reply message reply to the client at its Internet address and port

Server Side

Request-reply Message Structure



int (0=Request, 1= Reply)

int

RemoteObjectRef

int or Method

array of bytes

Why is there a requestId ?

Request-reply: Questions

- Which transport protocol would be more suitable, UDP or TCP?
- Why?
- How are request-reply protocols usually implemented?

Datagram-based RRP

What can go wrong?

What are the remedies?

Interprocess Communication

3.5 Group Communication

1. Principles
2. APIs for UDP and TCP
3. External Data Representation
4. Client Server Communication
5. **Group Communication**

Group Communication

Multicast transmission

a message sent to a specified group of recipients

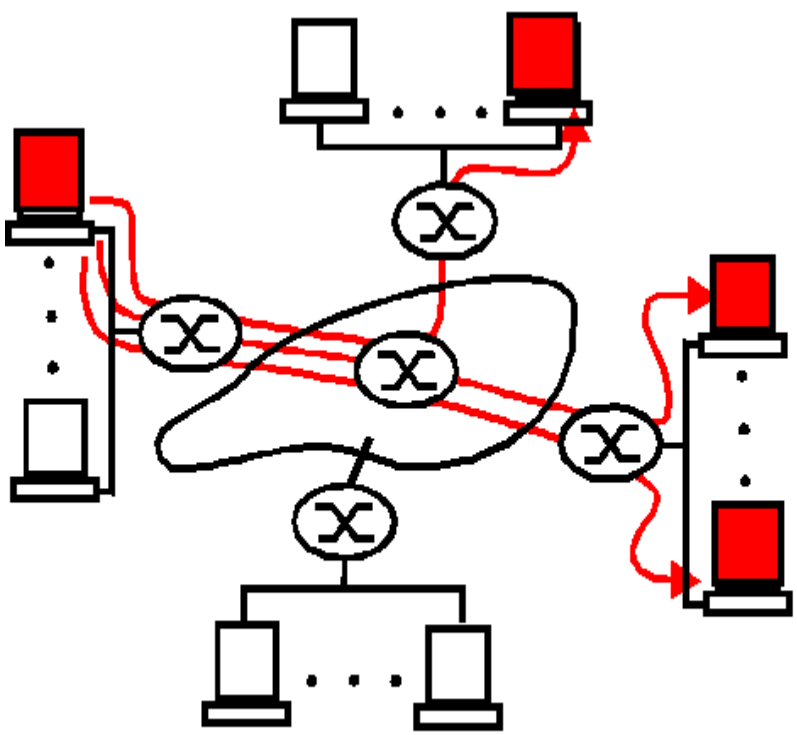
(as opposed to unicast and broadcast)

Examples

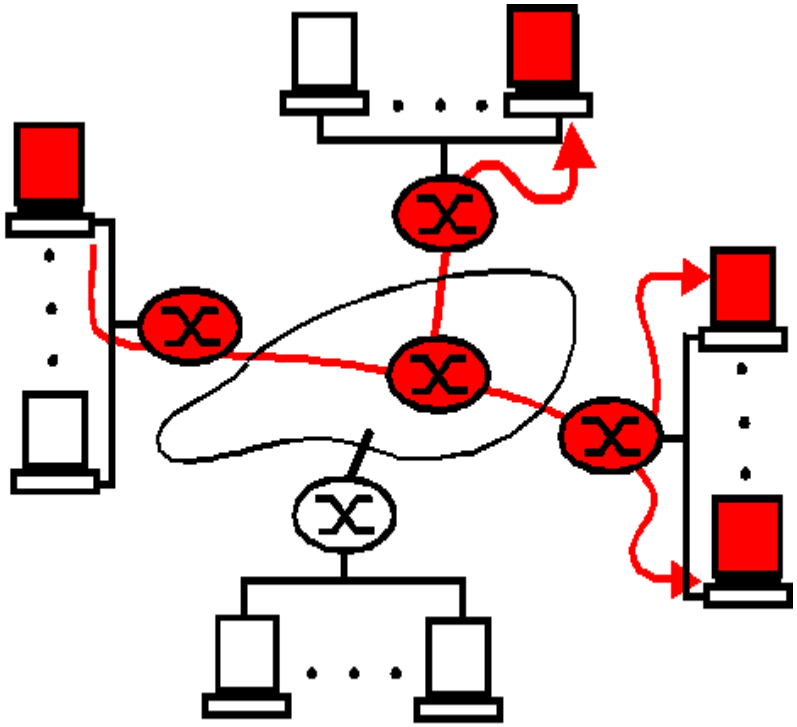
- Fault tolerance based on replicated services
  - requests go to all servers
- Spontaneous networking
  - all nodes of the network receive messages
- Better performance through replicated data
  - the updated data goes to all storing the data
- Event notification

Requirements for delivery guarantees differ

Two Implementations of Multicast



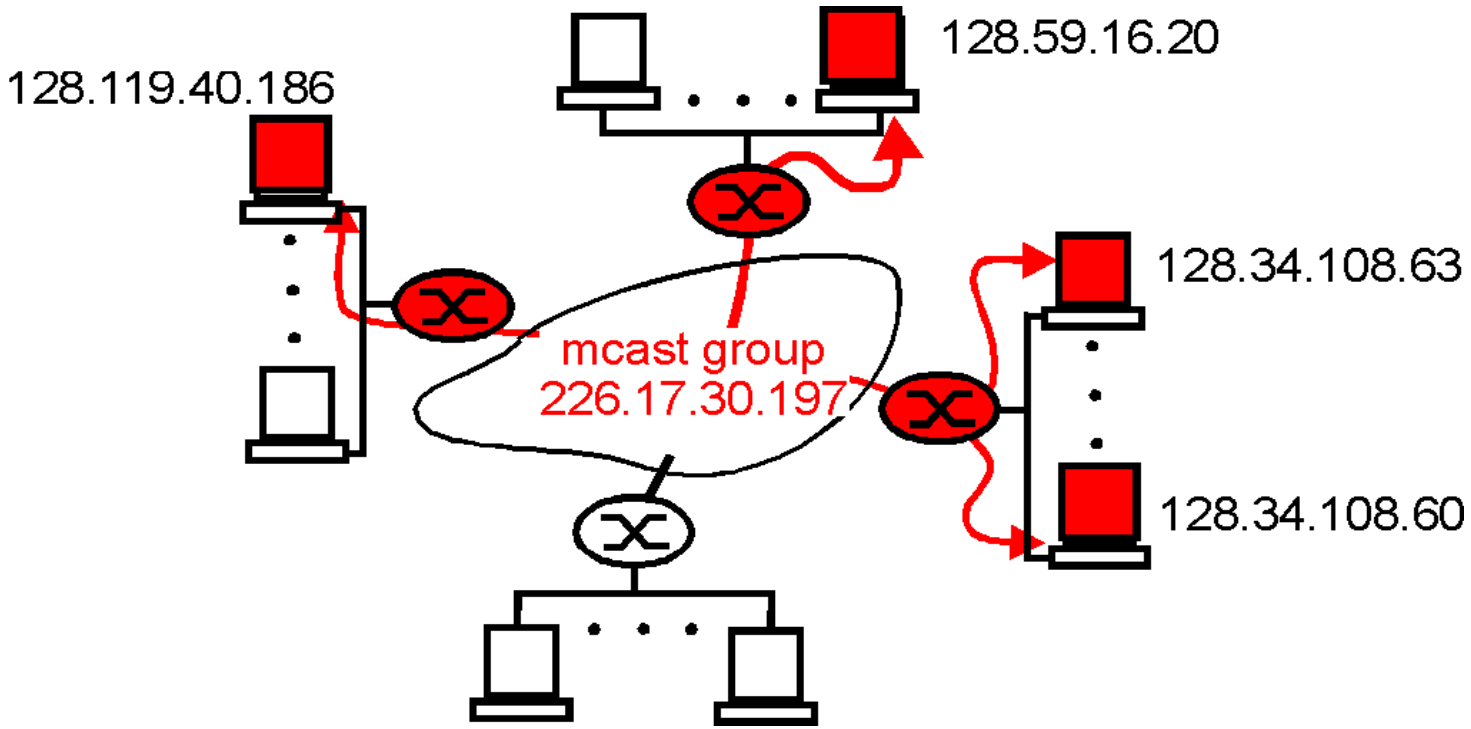
multicast via unicast



network multicast

IP Multicast

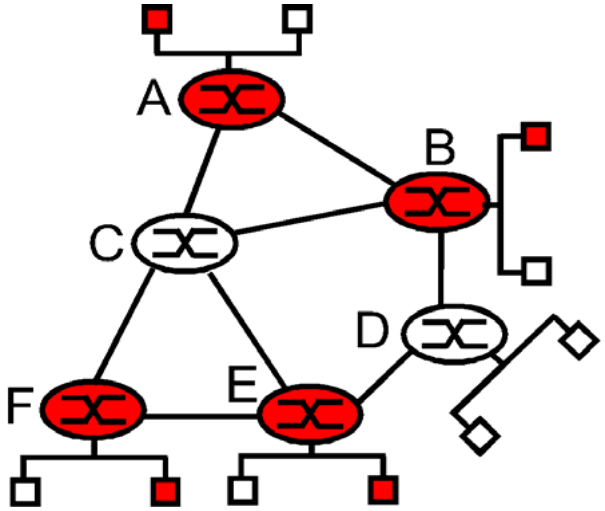
- Multicast groups are specified by an IP address of class D and a port number *(multicast address)*
- Available only for datagrams (UDP)
  - Time To Live (TTL) specifies range of the multicast



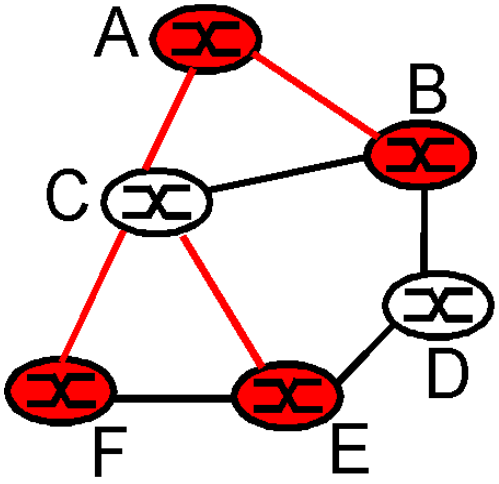
Multicast Protocols

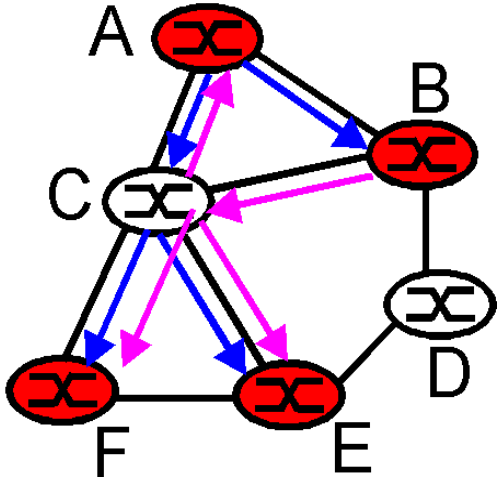
- Internet Group Management Protocol (IGMP)
  - for interaction between host and nearest router
  - allows hosts to join and leave a multicast address dynamically
  - routers query hosts for their group membership (soft state registration: expires if not confirmed)
- Routing within AS (= Autonomous Systems):
  - for each group, construct a tree connecting the routers involved in the group
  - approaches based on distance vector (MDVRP) and shortest path (MOSPF) technique
- Routing across AS
  - MDVRP and multicast version of BGP (BGMP)

Multicast Routing



Scenario:
Multicast hosts,
their attached routers,
and other routers

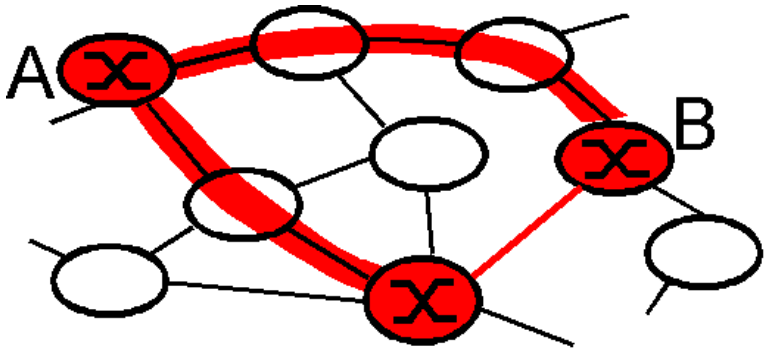




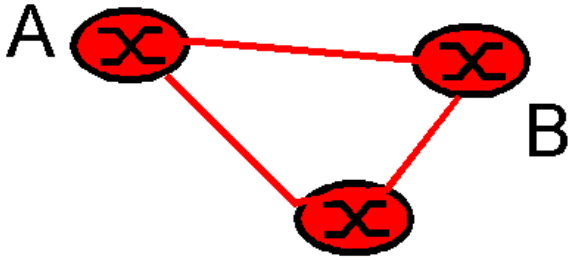
Approaches:
- single shared tree
- one source-based tree for each router

Tunneling

- Crux: not all routers support multicast
- Solution: multicast-enabled routers form a virtual network *("overlay network")*



physical topology



logical mcast topology

- Nodes communicate by "tunneling"
  - packets to multicast IP addresses are sent as payload to the next multicast-capable router

Multicast in Java

```
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[0]);
            s = new MulticastSocket(6789);
                    s.setTimeToLive(255); //TTL of messages
                s.joinGroup(group);
            byte [] m = args[0].getBytes();
            DatagramPacket messageOut =
                    new DatagramPacket(m, m.length, group, 6789);
            s.send(messageOut);
```

Multicast in Java (cntd.)

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

Exercise

- Would multicast be an option for implementing a chat system?
- Why is there no TCP version of multicast?
- Multicast messages can be read by everyone who joins a group. Should one enhance IP Multicast so that messages can only be received by authorised users?
- What guarantees can IP Multicast give regarding
  - reliability
  - ordering of messages?

References

In preparing the lectures I have used several sources.

The main ones are the following:

Books:

- Coulouris, Dollimore, Kindberg. Distributed Systems – Concepts and Design (CDK)

Slides:

- Marco Aiello, course on Distributed Systems at the Free University of Bozen-Bolzano
- Andrew Tanenbaum, Slides from his website
- CDK Website
- Marta Kwiatkowska, U Birmingham, slides of course on DS