Distributed Systems based on Sockets

Polytech/INFO 4, 2022-2023

Fabienne Boyer, Olivier Gruber,
UFR IM2AG, LIG, Université Grenoble Alpes
Fabienne.Boyer@imag.fr

POLYTECH© GRENOBLE

UNIVERSITÉ Grenoble Alpes

Outline

- Introduction to sockets
- Point-to-point communication with TCP sockets
- Point-to-point communication with UDP sockets
- Group communication with sockets
- Client-server programming with sockets

Internet Protocol

- **IP (Internet Protocol)**
  - The Internet protocol is not the Web
    - The Web refers to HTTP, built on top of TCP/IP
  - It corresponds to the network layer of the OSI model
  - It manages addressing, routing and transport of data packets

- **IP addresses**
  - 4 bytes(IPv4), naming a host machine (e.g. 192.168.2.100)
    - Addresses on 16 bytes for IPV6
  - IP addresses are location dependent

DNS (Domain Name System)

- **IP Address resolution**
  - manages the translation between a host name and its IP address

- **Discussing DNS design**
  - A fairly complex world-wide distributed system in itself
  - Allows name aliases (multiple names for an address)
  - And the reverse (multiple addresses for a name)
  - Organized as hierarchical zones across the world
    - A zone is managed by a DNS server
    - Servers are replicated for high availability (following a master-slave design)

Ports

- **An IP address and a port names a communication end point**
  - Ports refer to communication channels on the local machine
- **Port numbers are managed by the operating system**
  - Ports between 1 and 1023 are well-known (513=rlogin, 25=telnet, ..)
  - Ports between 1024 and 49151 can be registered with the Internet Corporation
  - Ports between 49152 and 65535 are dynamic
- **Dynamic ports are allocated on-demand to processes**
  - A port may be allocated to only one process at a time

Sockets

- **Provides 3 protocols for sending/receiving data over IP**
  - TCP protocol
    - Stream oriented
    - Lossless
    - Ordered
    - Connection-oriented
  - UDP protocol
    - Packet based
    - Not lossless, no order
    - Efficient
  - Group protocol
    - Packet based
    - Not lossless, no order

Sockets

- **Provides 3 protocols for sending/receiving data over IP**
  - TCP protocol
    - Stream oriented: components exchange streams of bytes
    - Lossless: 0 bytes lost
    - Ordered: 0 bytes reordered
    - Connection-oriented
  - UDP protocol
    - Packet based: components exchange messages
    - Not lossless, no order: packets may be lost or reordered
    - Efficient
  - Group protocol
    - Packet based: components broadcast messages
    - Not lossless, no order: packets may be lost or reordered

Typical applications over TCP and UDP

- TCP
  - Applications that do not support loss or reordering
  - Transferring files (ftp for instance)
  - Downloading web pages
  - ..
- UDP
  - Applications requiring high bandwidth & accepting loss or reordering
  - Transmission of video/sound in real time

    Ex: VoIP (Skype)

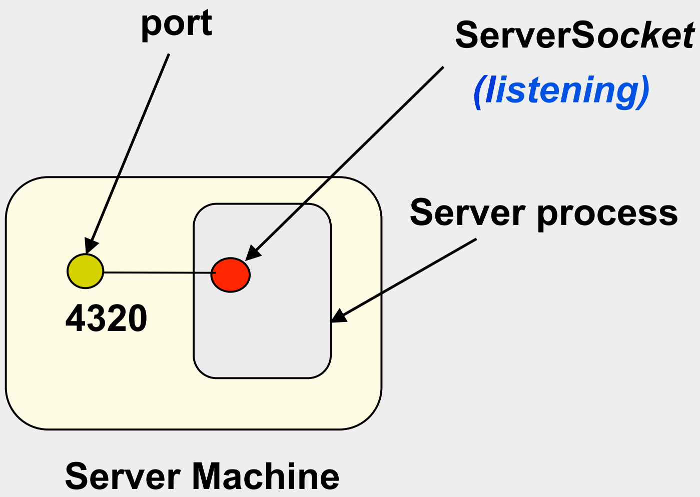
    Out of sequence or incomplete frames are just dropped

Outline

- Introduction to sockets
- **Point-to-point communication with TCP sockets (in Java)**
- Point-to-point communication with UDP sockets
- Group communication with sockets
- Client-server programming with sockets

TCP Sockets – Steps Involved

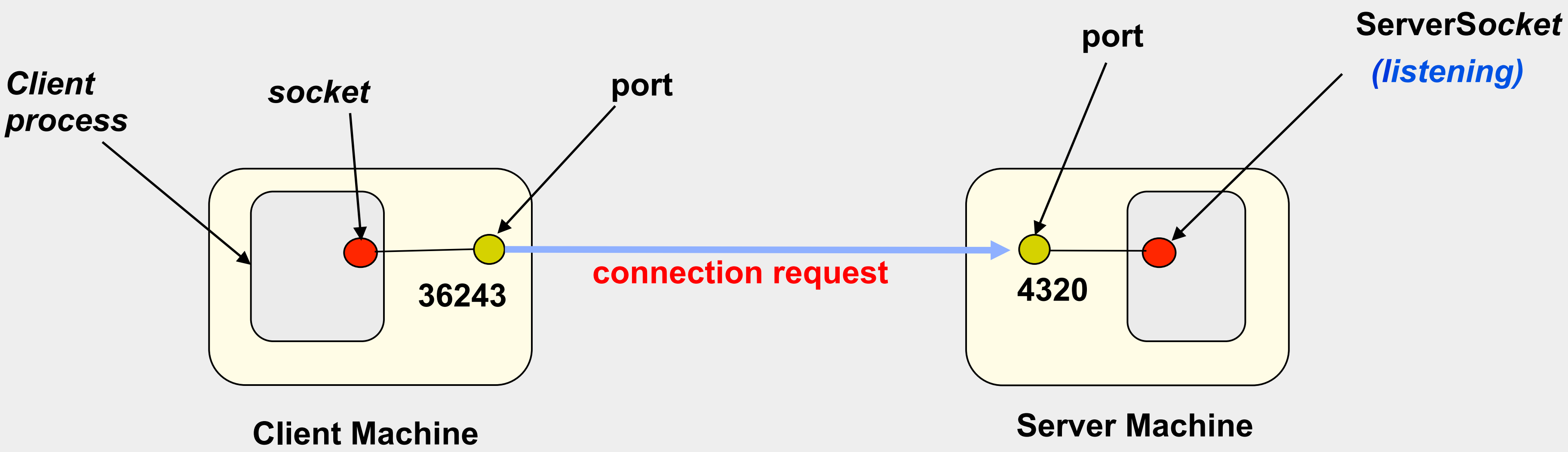
- **Server side**
  - Assume the port of the server is decided
  - Create a ServerSocket on the desired port to listen to connection requests
  - Loop: wait for a connection request, accept it, then communicate with the client



TCP Sockets – Steps Involved

- **Client side**
  - Create a Socket to connect to the server socket

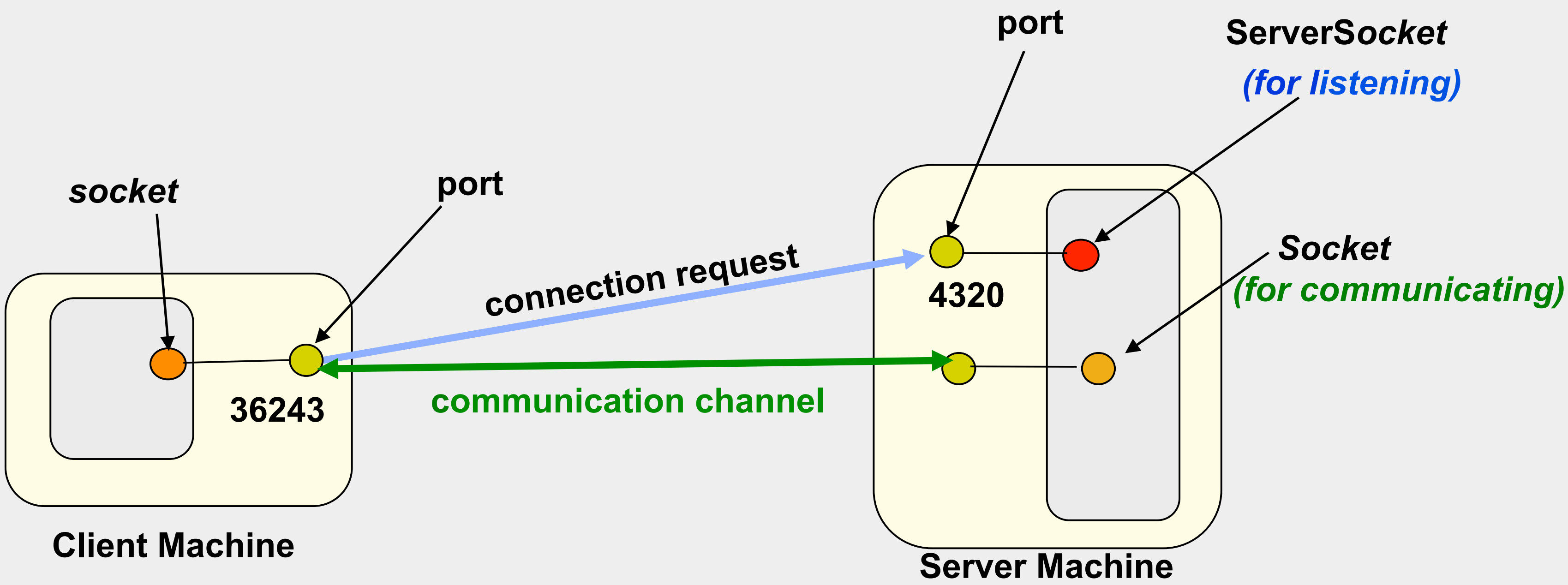
    (automatically allocates a port & sends a connection request to the server)



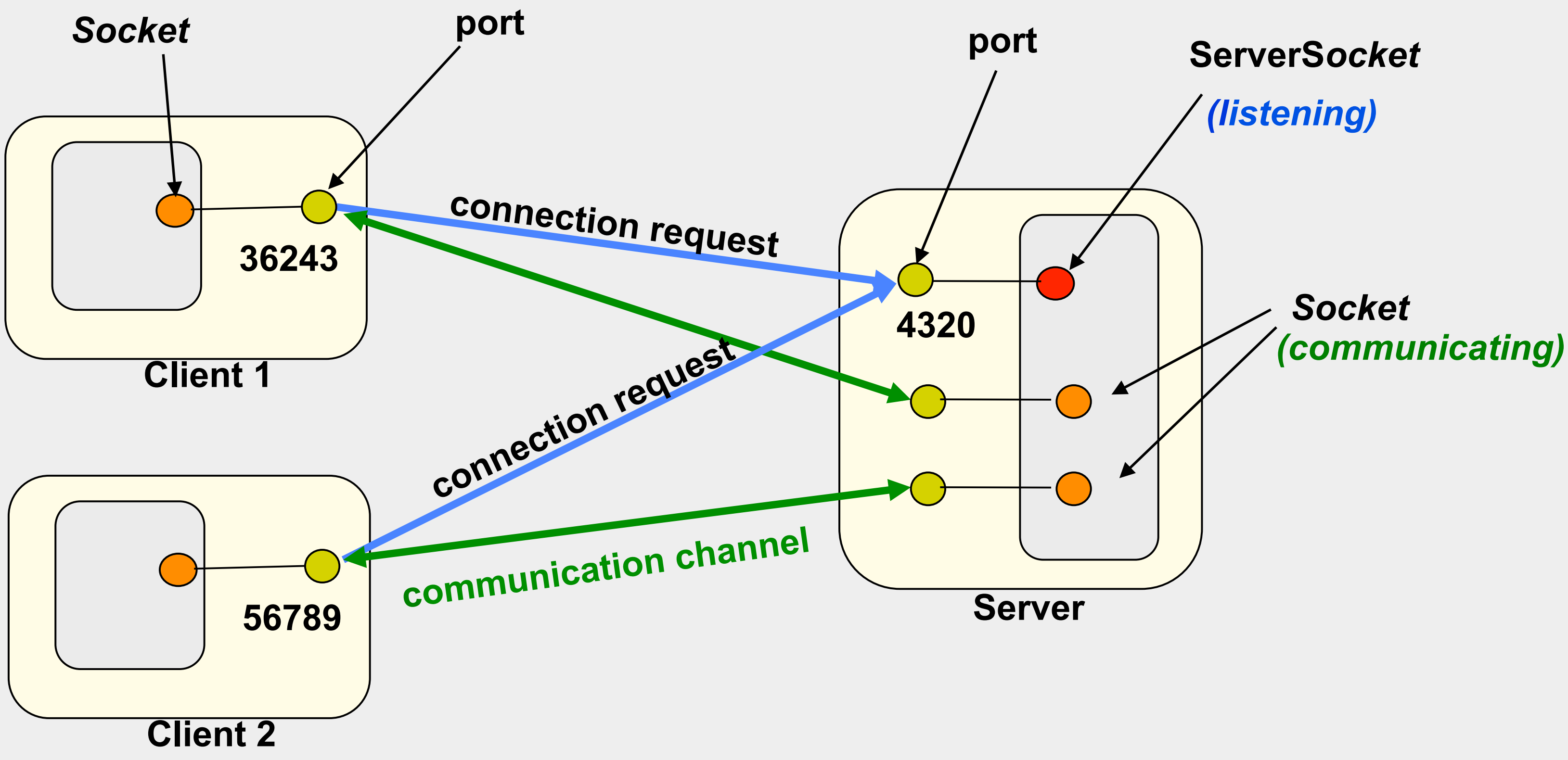
TCP Sockets – Steps Involved

- **Server side**
  - Accept the connection from the client

    (a couple (port, Socket) is automatically allocated to communicate with the client)



TCP Sockets – Steps Involved



Java classes related to TCP sockets

- **java.net package**
  - ServerSocket class
    - Represent a listening socket on a server (to accept connection requests)
    - Configured with a backlog (maximum number of queued connection requests, to avoid queuing too much connection requests)
  - Socket class
    - Represent a communication socket
    - Both on server and client sides
    - Configured with different parameters (e.g., TCP_NODELAY to avoid buffering data written to the network , see java.net.SocketOptions)
  - Other utility classes (InetAddress, SocketAddress, ..)

Example of Java Server on TCP

```
import java.net.*;
..
// SERVER SIDE
int port = 4320;
int backlog = 3;

ServerSocket listenSoc = new ServerSocket(port, backlog);
// server loop
while (true) {
   // wait for a connection request
    Socket soc= listenSoc.accept();          // appel bloquant

    // communicate with the client
    <receive bytes from client through soc.getInputStream()>
    <send bytes to client through soc.getOutputStream()>
}
```

Example of Java sockets/TCP

```
import java.net.*;

// SERVER SIDE

int port = 4320;

int backlog = 3;

ServerSocket listenSoc
ServerSocket(port);

// server loop

while (true) {
   // wait for a connec
   Socket soc= server.a
   // communicate with
   ...
```

...

```
import java.net.*;

//CLIENT SIDE

String serverHost = "goedel.imag.fr";

int serverPort = 4320;


// connect to the server

Socket soc = new Socket(serverHost, serverPort);


// communicate with the server
<send bytes to server through soc.getOutputStream()
<recv bytes from server through soc.getInputStream()
```

Stream-based communication in Java

java.io package

- InputStream / OutputStream
  - abstract classes that represent streams of bytes
- **DataInputStream / DataOutputStream**
  - To manipulate streams of Java primary types
- ObjectInputStream / ObjectOutputStream
  - To manipulate streams of Java objects
- FileInputStream / FileOutputStream
  - To read data from a file or write data to a file
- FilterInputStream / FilterOutputStream
  - To transform data along the way
- BufferedInputStream / BufferedOutputStream
  - To bufferize bytes

Using Streams

- **Only wrap a stream into one upper stream**
  - Ex: DataInputStream upon InputStream
- **Only manipulate the upper stream**
  - Read, write, flush, close only the upper stream
- **Be sure that streams get closed in your code**

```
try {
  OutputStream os = soc.getOutputStream();
  DataOutputStream dos = new DataOutputStream (os);
  dos.writeUTF("A simple sentence");
  ...
} catch ( Exception e ) {
  ...
} finally {
   ..
   if (dos != null) try { dos.close();} catch (Exception e) {}
   ..
}
```

Continuing with our Client/Server over TCP

```java
import java.io.*;
..
// SERVER side
int port = 4320;
int backlog = 3;
ServerSocket listenSoc = new ServerSocket(port, backlog);

while (true) {
     // wait for a connection request
     Socket soc = listenSoc.accept();

     // the server sends the date to the client
     Date date = new Date();
     byte[] b = date.toString().getBytes();
     OutputStream os = soc.getOutputStream();
     os.write(b);
}
```

Pursuing on our Client/Server over TCP

```
import java.io.*;

// SERVER side
...

while (true) {
   // wait for a connecti
   Socket soc = server.ac

   // send the date to th
   Date date = new Date()
   byte[] b = date.toStri
   OutputStream os = socC
   os.write(b);
}
```

```
import java.io.*;

// CLIENT side
String date = "";
..
// connect to server
Socket soc = new Socket(serverHost,serverPort);

// receive the date from server
InputStream is = soc.getInputStream();
byte[] b = new byte[100];
int nb = is.read(b);
if (nb>0) date = new String(b);
System.out.println("Date: " + date);
```

Not so simple..

```
// SERVER side
OutputStream os = soc.getOutputStream();
Date date = new Date();
byte[] b = date.toString().getBytes();
os.write(b);


// CLIENT side
InputStream is = soc.getInputStream();
byte[] b = new byte[100];
int nb = is.read(b);
If (nb>0) date = new String(b);
```

Is this correct ?

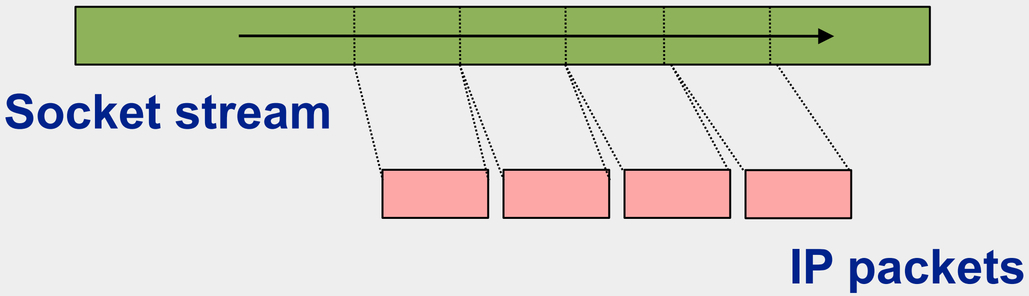
Not so simple..

```
// SERVER side
OutputStream os = soc.getOutputStream();
Date date = new Date();
byte[] b = date.toString().getBytes();
os.write(b);


// CLIENT side
InputStream is = soc.getInputStream();
byte[] b = new byte[100];
int nb = is.read(b);
If (nb>0) date = new String(b);
```

Is this correct ? No..

1) data is
sent by packets, it may
just not be fully received

2) strings encoding may
differ on client and server



Checking that all bytes are received

```
// SERVER side
OutputStream os = soc.getOutputStream();
DataOutputStream dos = new DataOutputStream(os);
Date date = new Date();
byte[] b = date.toString().getBytes();
dos.writeInt(b.length);
dos.write(b);



// CLIENT side
InputStream is = soc.getInputStream();
DataInputStream dis = new DataInputStream(is);
int length = dis.readInt();
byte[] b = new byte[length];
dis.readFully(b);
date = new String(b);
```

Send the length of data as prefix (or use a marker at the end of data)

Use writeInt method of DataOutputStream (endianness proof)

Example of using an end-mark

```
// Echo SERVER (exchanging lines of characters)
...
while (true) {
 Socket soc = server.accept();

 InputStream is = soc.getInputStream();
 InputStreamReader isr = new InputStreamReader(is);
 BufferedReader br = new BufferedReader(isr);

 OutputStream os = soc.getOutputStream();
 OutputStreamReader osr = new OutputStreamReader(os);
 BufferedWriter bw = new BufferedWriter(osr);

 String line = br.readLine();
 bw.write(line);
 bw.newLine();
 bw.close();
}
```

"\n" is used as the end-mark in the method readLine())

Paying attention to String encoding

```
// SERVER side
OutputStream os = soc.getOutputStream();
DataOutputStream dos = new DataOutputStream(os);
Date date = new Date();
byte[] b = date.toString().getBytes("UTF-8");
dos.writeInt(b.length);
dos.write(b);



// CLIENT side
InputStream is = soc.getInputStream();
DataInputStream dis = new DataInputStream(is);
int length = dis.readInt();
byte[] b = new byte[length];
dis.readFully(b);
String date = new String(b,"UTF-8");
```

Encoding may differ from one VM to another

Advice is to use UTF-8 / UTF-16

Paying attention to String encoding

```
// SERVER side
OutputStream os = soc.getOutputStream();
DataOutputStream dos = new DataOutputStream(os);
Date date = new Date();
dos.writeUTF(date.toString());



// CLIENT side
InputStream is= soc.getInputStream();
DataInputStream dis = new DataInputStream(is);
String date = dis.readUTF();
```

Other option for exchanging strings

- ***writeUTF***
- ***readUTF***

(manage the length & use UTF-8)

Flushing data to send

```
// SERVER side
OutputStream os = soc.getOutputStream();
DataOutputStream dos=new
DataOutputStream(os);
Date date = new Date();
dos.writeUTF(date.toString());
dos.flush();



// CLIENT side
InputStream is = soc.getInputStream();
DataInputStream dis = new
DataInputStream(is);
String date = dis.readUTF();
```

Do we need to flush the data to send?

- ***Not obviously at each write***
- ***Can be made when bytes to send have been accumulated in the stream***
- ***It forces the transfer of the data into the low level buffers***
- ***Closing a stream forces a flush***

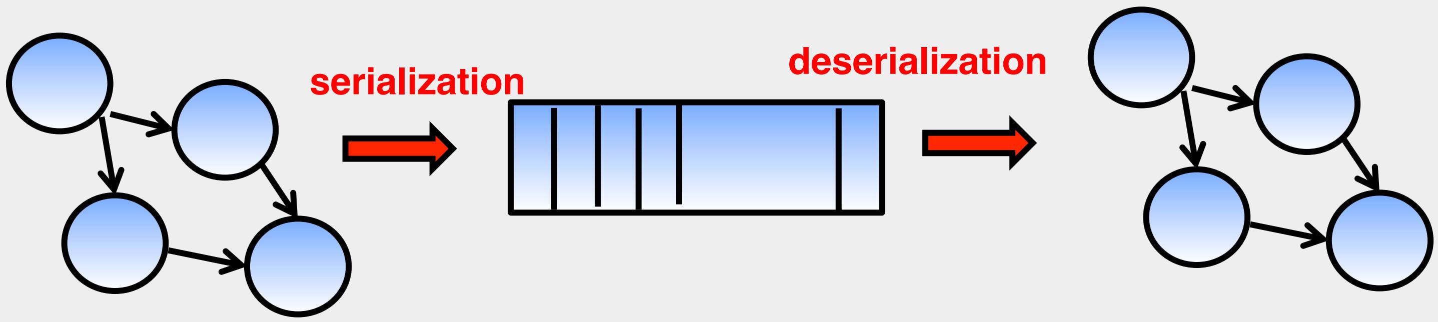
Sending and receiving objects

```
// SERVER side
...
while (true) {
 Socket soc = server.accept();
 OutputStream os = soc.getOutputStream();
 Date date = new Date();
 ObjectOutputStream oos=new ObjectOutputStream(os);
 oos.writeObject(date);
 oos.close();
}




// CLIENT side
Socket soc= new Socket(serverHost,serverPort);
InputStream is = soc.getInputStream();
ObjectInputStream ois = new ObjectInputStream(is);
Date date = (Date) ois.readObject();
ois.close();
```

We can also exchange any object that implements the Serializable interface

Object Serialization

- *Object → byte[] , byte[] → Object*
- **Any class may implement the Serializable interface**
  - If it does, instances of that class can be serialized
- **Serialization is a deep copy**
  - Recursive serialization along object references
  - Sharing is respected



- **We'll see more on serialization later**

Outline

- Introduction to sockets
- Point-to-point communication with TCP sockets (in Java)
- **Point-to-point communication with UDP sockets (in Java)**
- Group communication with sockets
- Client-server programming with sockets

Java sockets over UDP

- **Communicating in the unconnected mode**
  - UDP protocol allows to send packets of data, called datagrams
  - A datagram is an independent self-contained message whose arrival and arrival time are not guaranteed

Java classes related to UDP

- **java.net.DatagramPacket**
  - Represents a data packet
    - Essentially a byte buffer
    - Maximum length given by DatagramSocket. getReceiveBufferSize()
  - Includes an InetAddress and port number

- **java.net.DatagramSocket**
  - Used for sending and receiving datagram packets

Example of Java sockets/UDP

```
import java.net.*;

// SERVER side

int port = 1234;
DatagramSocket soc = new DatagramSocket(port);

while (true) {
    // allocate a datagram packet and wait for a client request
    byte[] buf = new byte[256];
    DatagramPacket packet = new DatagramPacket(buf, buf.length);
    serverSoc.receive(packet);
    String request=new String(packet.getData());
    ..

    // Send the reply to the client
    byte reply = new byte[128];
    ..
    InetAddress clientAddr = packet.getAddress();
    int clientPort = packet.getPort();
    packet = new DatagramPacket(reply, reply.length, clientAddr, clientPort);
    soc.send(packet);
}
```

Example of Java sockets/UDP

```
import java.net.*;

// SERVER side

int port = 1234;
DatagramSocket serverSoc =

while (true) {
   // allocate a datagram
   byte[] buf = new byte[
    DatagramPacket packet
    serverSoc.receive(pack

    // send a reply to th
    byte[] reply = ..
     InetAddress clientAdd
   int clientPort = packe
    packet = new Datagram
    soc.send(packet);
}
```

```
import java.net.*;
int serverPort = 1234;
String serverHost = ...;

// SERVER side

// Create a datagram socket
DatagramSocket soc = new DatagramSocket();

// Send the request to the server
byte[] buf = …
InetAddress serverAddr=InetAddress.getByName(serverHost);
DatagramPacket packet = new DatagramPacket(buf,
buf.length, serverAddr, serverPort);
clientSoc.send(packet);

// Receive the reply from the server
packet = new DatagramPacket(buf, buf.length);
soc.receive(packet);
String reply=new String(packet.getData());
System.out.println(reply);`
```

Outline

- Introduction to sockets
- Point-to-point communication with TCP sockets (in Java)
- Point-to-point communication with UDP sockets (in Java)
- **Group communication with sockets**
- Client-server programming with sockets

Java Multicast

- **Based on UDP sockets**
  - Datagram packets (same as before)
  - Java class MulticastSocket extends DatagramSocket

- **Relies on IP-level multicast**
  - Multicast IP adresses
  - Class D addresses are reserved for multicast
  - In the range 224.0.0.0 to 239.255.255.255
  - A multicast group is just a multicast address and port

Java Multicast

- **Multicasting to a group**
  - Create a datagram packet
  - Make a normal UDP send, to the group InetAddress and port
- **Joigning a multicast group**
  - Create the multicast socket with the group port
  - Join the multicast group, use the multicast address
  - Receive messages multicasted to the group
- **Leaving a multicast group**
  - Explicit departure

Joining a group and receiving messages

```
import java.net.*;

// Multicast group
int groupPort = 5000;
InetAddress groupAddr = InetAddress.getName(“225.4.5.6”);

// Create a socket and join the group
MulticastSocket soc = new MulticastSocket(groupPort);
soc.joinGroup(groupAddr);

// Receiving
byte buf[] = new byte[1234];
DatagramPacket packet = new DatagramPacket(buf, buf.length);
soc.receive(packet);

// When done, leave the multicast group and close the socket
soc.leaveGroup(groupAddr);
Soc.close();

}
```

Sending to a group

```
import java.net.*;

// Multicast group
int groupPort = 5000;
InetAddress groupAddr = InetAddress.getName(“225.4.5.6”);

// Create the socket
// but we don’t bind it and we don’t join the multicast group
MulticastSocket soc = new MulticastSocket();

byte buf[] = new byte[10];
For (int i=0; i<buf.lenght; i++)
  buf[i] = (byte)i;

// Create a datagram packet and send it
DatagramPacket packet = new DatagramPacket(buf, buf.length, groupAddr,
groupPort);

// Send the packet
Byte ttl = 1;
soc.send(packet, ttl);

// When doneclose the socket
soc.close();
}
```

Java Multicast

- **Limitations**
  - IP multicast is supported by many routers
    - But most Internet providers forbid IP multicast
    - Over the public Internet, multicast is simply not available
  - Usable on local LAN

- **If you need multicast features, use a Middleware solution**
  - Middleware built above IP or UDP or TCP/IP
    - Using point to point messages
    - Provides different properties(ordererd, reliable, ..)

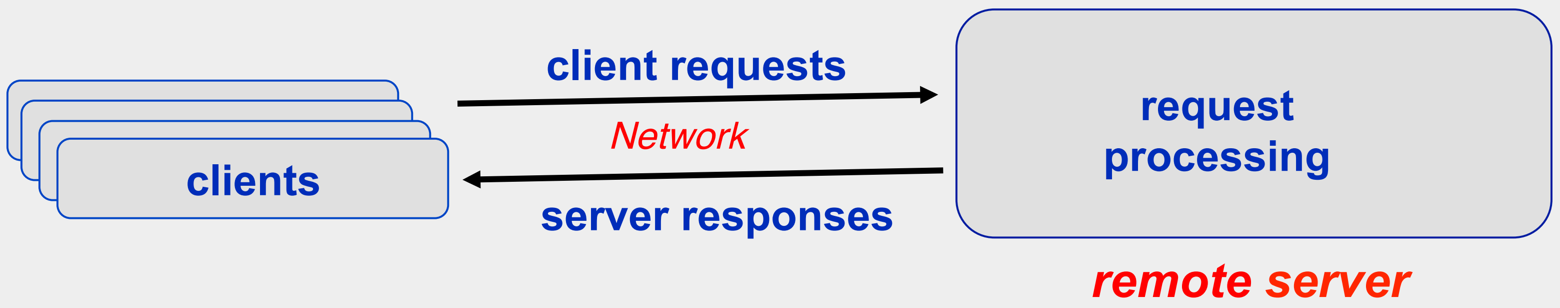
Outline

- Introduction to sockets
- Point-to-point communication with TCP sockets (in Java)
- Point-to-point communication with UDP sockets (in Java)
- Group communication with sockets
- **Client-Server programming with sockets**

Socket-based Server Design

- Server: a process that receives requests from remote clients, "execute" the requests and replies to the clients

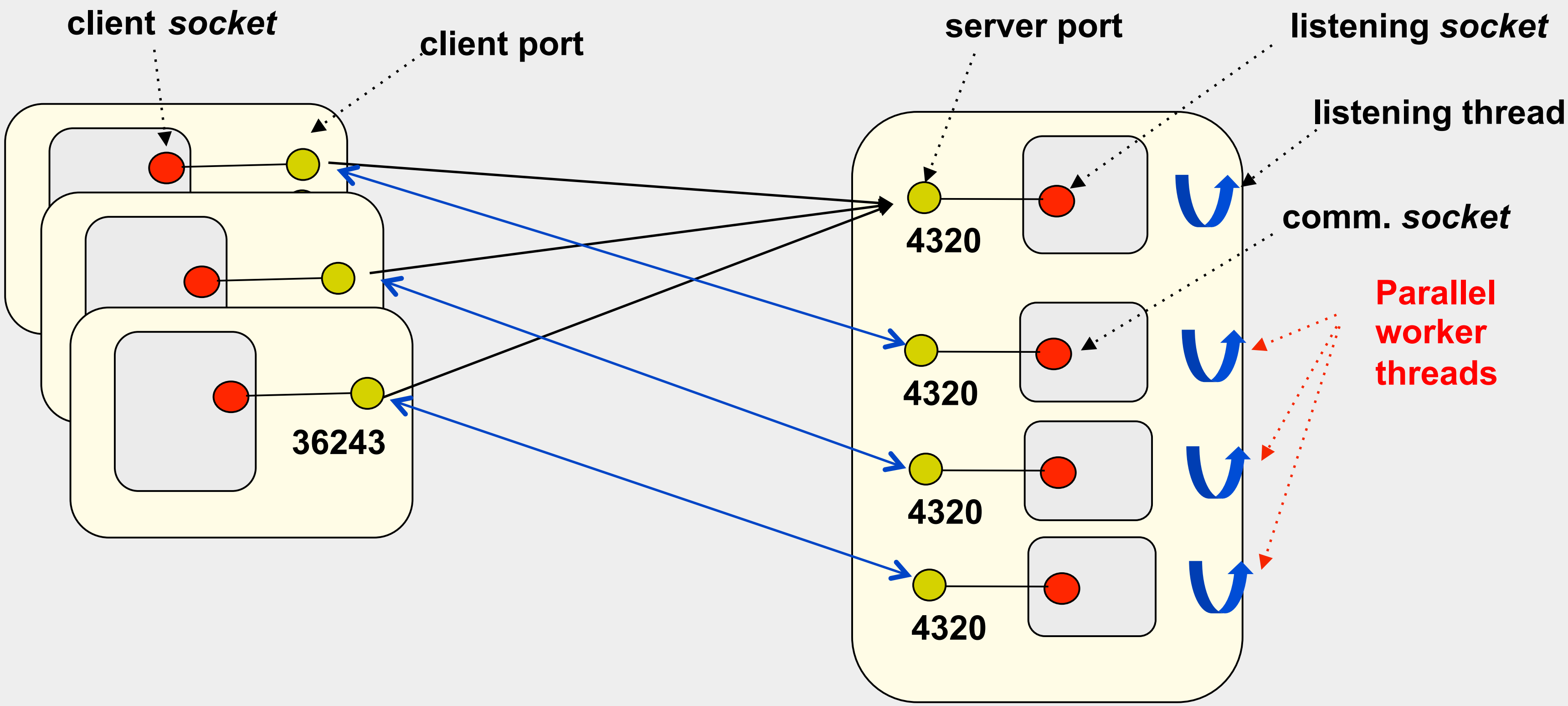
- Socket-based: clients and server communicate through the socket layer



Server design

- **Three main models**
  - Sequential: a given thread processes incoming requests in sequence
  - Parallel : multi-threaded (most common case) or multi-processes
    Replicated: a same request is processed by several threads or processes

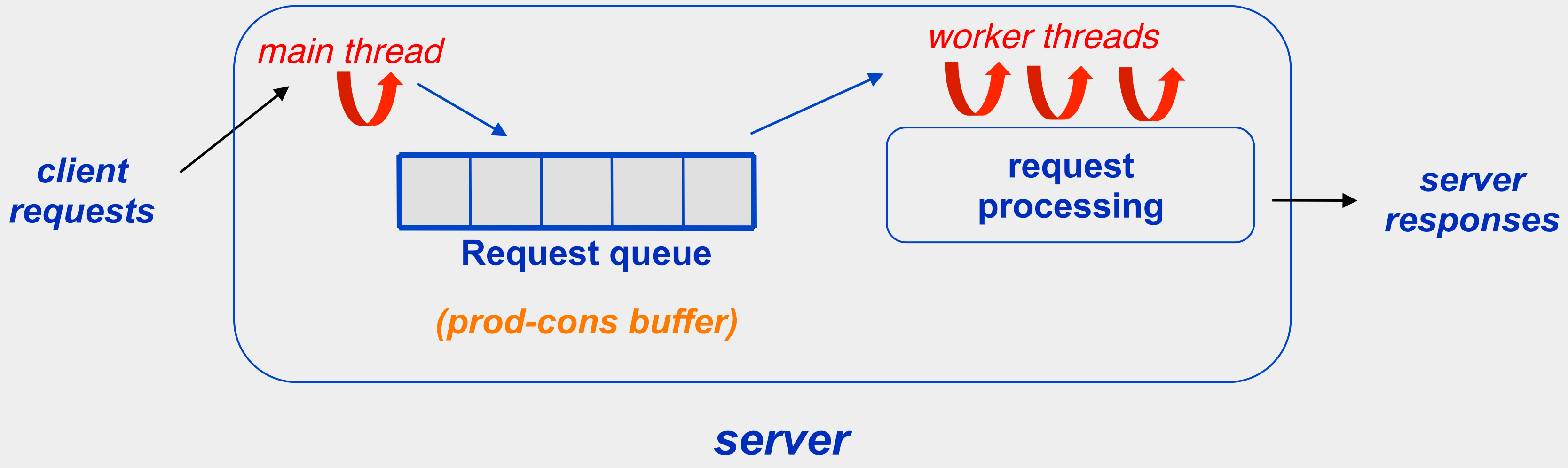
Multi-threaded TCP Server



Multi-threaded TCP server: basic design

```
class MultiThreadedTCPServer {
  …
  public static void main(String[] args) throws IOException {
         initComm();
         while (true) {
                  Socket soc= socListen.accept();
                  // create a new worker for each client
                  Worker worker = new Worker(soc).start();
         }
 ..

Class Worker extends Thread {
  Worker (Socket soc) {..}
  public void run(){
   // receive request from soc, process it and reply to client
   // do this as many times as required  (session-oriented communication)
   // at the end, close soc
  }
}
```

Multi-threaded TCP server: pool-based design



Multi-threaded TCP server: pool-based design

```
class MultiThreadTCPServer {
  public static void main(String[] args) throws IOException {
         initComm();
         ProdCons clientsBuffer = new ProdCons(..);
         while (true) {
                  Socket soc= socListen.accept();
                  clientsBuffer.put(soc);
         }
 ..

Class Worker extends Thread {
 Message m;
 Worker (ProdCons clientsBuffer) {this.clientsBuffer = clientsBuffer;}

 public void run(){
   while (true){
    Socket soc= clientsBuffer.get();
   // receive request from soc, process it and reply to client
   // do this as many times as required (session-oriented communication)
   // at the end, close soc
  }
}
```

Multi-threaded UDP server

```
class MultiThreadServer {
  public static void main(String[] args) throws IOException {
         initComm();
         while (true) {
                 Message message = receiveMessage();
                 Worker worker = new Worker(message).start();
         }
 ..

Class Worker extends Thread {
 Message m;
 Worker (Message m) {this.message = m;}
 public void run(){
   // get request and client port from m,
   // process request
   // reply to client
}
..
```

Multi-threaded UDP server: pool-based design

```
class MultiThreadUDPServer {
  public static void main(String[] args) throws IOException {
          initComm();
          ProdCons messagesBuffer = new ProdCons(..);
          while (true) {
                  Message message = receiveMessage();
                  messagesBuffer.put(message);
          }
 ..

Class Worker extends Thread {
 Message m;
 Worker (ProdCons messagesBuffer) {this.messagesBuffer = messagesBuffer;}

 public void run(){
   while (true){
   Message message = messagesBuffer.get();
   // process the message
   // reply to client
  }
}
..
```