Università degli Studi di Bologna
Facoltà di Ingegneria

Principles, Models, and Applications for Distributed Systems M

C/S applications using Java Sockets

Luca Foschini

SOCKET for COMMUNICATION

Need for a standard to connect distributed devices
distinct, different, highly heterogeneous

We will use sockets as a communication standard.

Sockets permit to access communication mechanisms offered by the operating system.

socket represents the local host **(endpoint)** in a **bi-directional** communication channel across network (from/to other hosts).

Client and Server applications running on distinct devices can communicate using **two different types** of communication with different costs and quality guarantees.

Sockets originated in Unix BSD 4.2

COMMUNICATION TYPES

- **connection-oriented**: there is a stable connection between Client and Server (state maintained only at endpoints, e.g.: Web)

STREAM socket

- **connectionless**: there is no connection, each message is dispatched independently from others (e.g.: mail system)

DATAGRAM socket

INTERNET SOCKET classes:

Connection-oriented using **TCP** Internet protocol

- `Socket` class, Client side
- `ServerSocket` class, Server side

connectionless using **UDP** Internet protocol

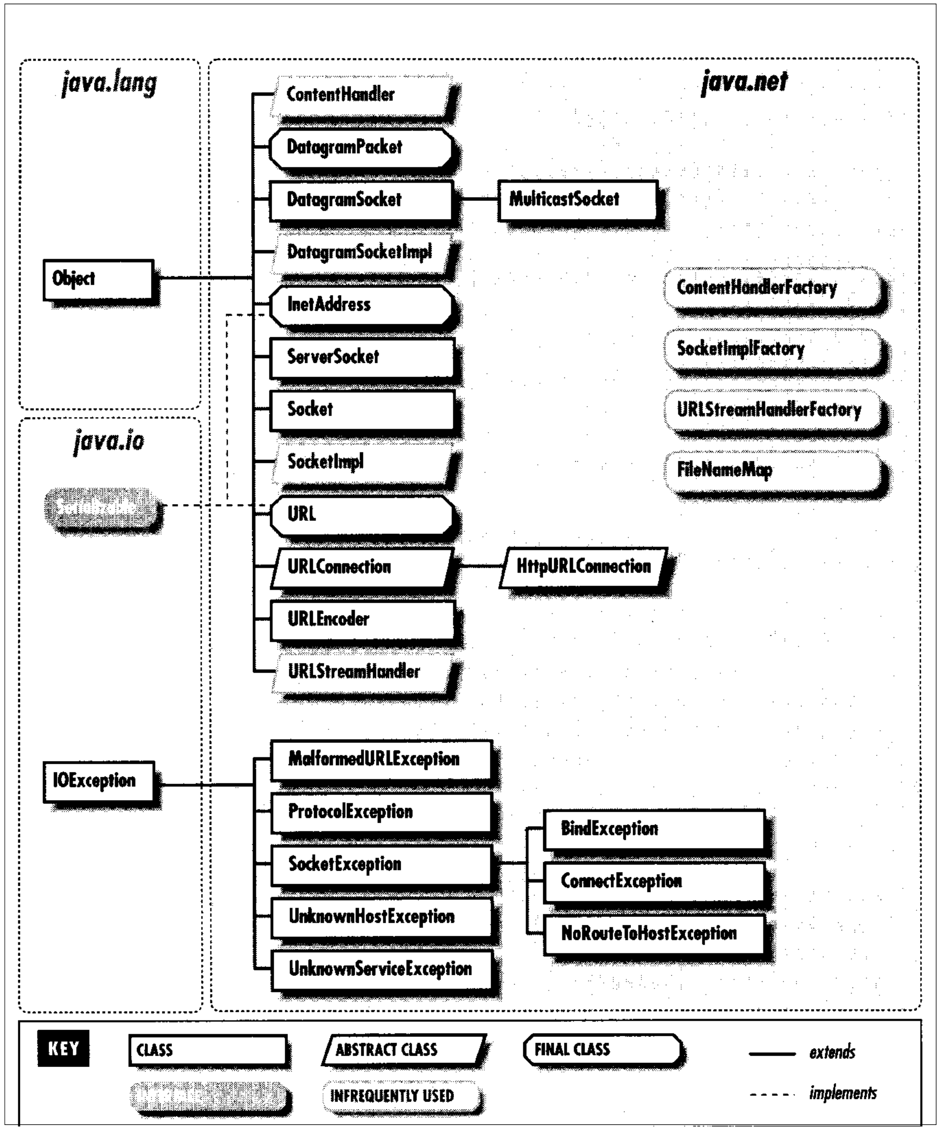
- `DatagramSocket` class for both Client and Server

Java classes hierarchy

The **`java.net`** networking package contains all classes needed to use sockets in Java.

Every Java Virtual Machine (JVM) release noticeable extended its classes and interfaces.

Its philosophy and mechanism do not change.



NAME SYSTEM

We need a name system to identify each element

*A distributed application comprises **processes distinct by their locality** that **exchange messages to communicate and cooperate** to **achieve coordinated results***

First problem: mutual process identification (Client and Server) across network

Each process must be associated to a **GLOBAL NAME**

unique, unambiguos, and simple

Node “**name**” + process “**name**” on that node

Process end points (sockets) typically are local w.r.t. the process itself

This first problem is solved by low protocol layers (transport and network). For Internet sockets, **transport names** (TCP, UDP) and **network** (IP).

SOCKET NAMES

A transport endpoint is uniquely identified by:

- an **IP address** (4 bytes / 32 bit) ⇨ IP level
- a **port** (16 bit integer) ⇨ TCP and UDP abstracion

GLOBAL NAME usage

Messages sent to a specific port of a specific machine (they do not directly reach a process)

To reach a **resource, we need a LOCAL NAME (socket)**

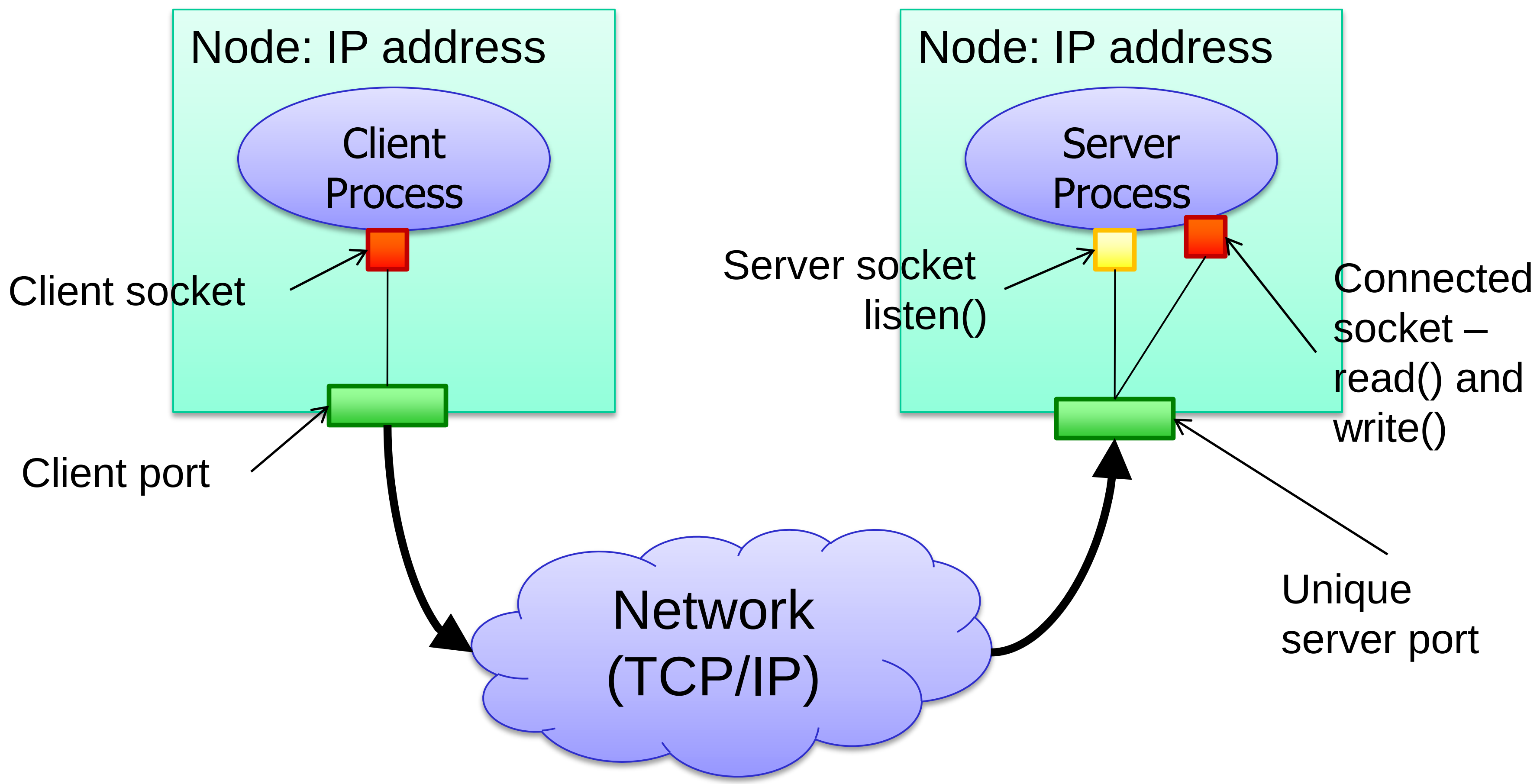
The process (one or more) binds to a port to receive (and send) messages

This **double name system** allows to identify a specific process without knowing its *local process identifier* (PID)

(IP network layer, PORT transport layer)

An end point for a communication flow is made of an IP and a port

SOCKET NAMES



GLOBAL SOCKET NAMES

IP numbers ⇨ IP address: e.g. `137.204.57.186`

Port numbers ⇨ port, 4 hex digits: **`XXXXh`** (dec. `1 - 65535`)

often represented as a single decimal number, e.g. `153, 2054`

Port function is to **identify a service**

Port numbers lower than 1024 are reserved (well-known port)

services offered by processes bound to those ports are standard

For example, Web is identified by port 80, i.e. the server process of a web site binds to port 80, from which it receives request for html pages.

Other well known TCP ports, server side:

`port` **21** ftp,

`port` **23** telnet,

`port` **25** mail,…

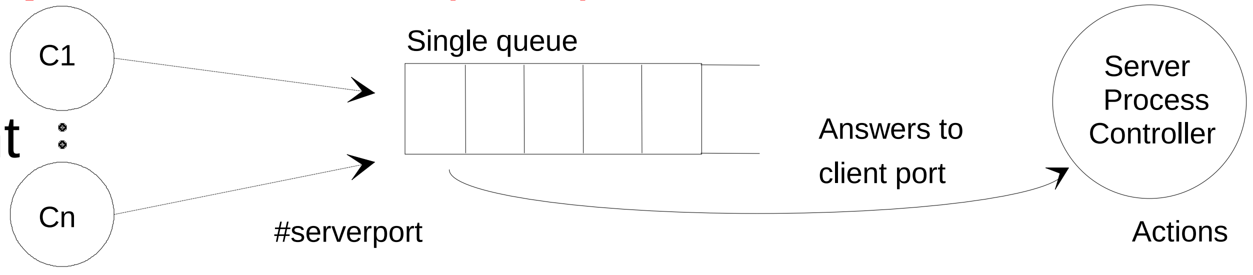
SEQUENTIAL SERVER

Server that manages a single request at a time

Connectionless sequential server (UDP)

Stateless services

Failure management not relevant

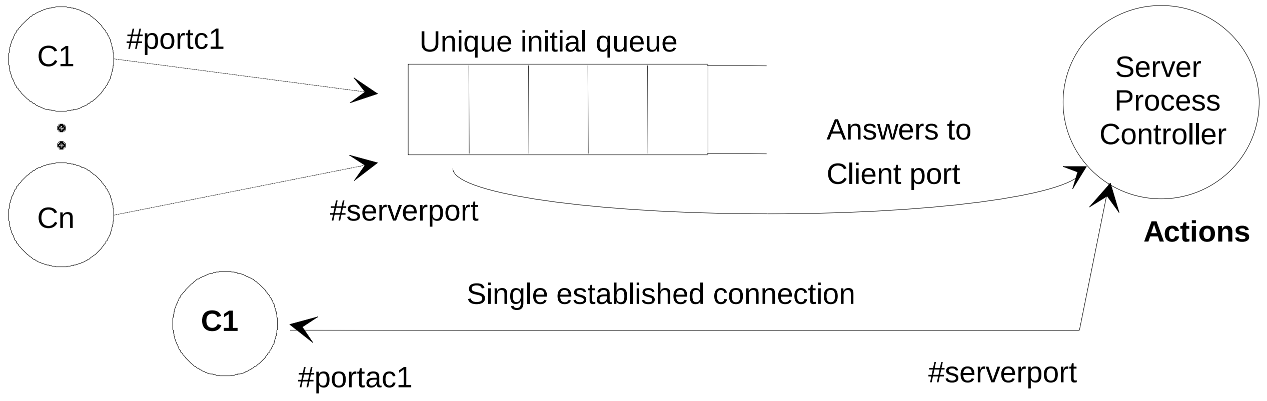


Connected sequentail server (TCP)

Reliable services

Limited state

Overhead due to connection control



PARALLEL SERVER

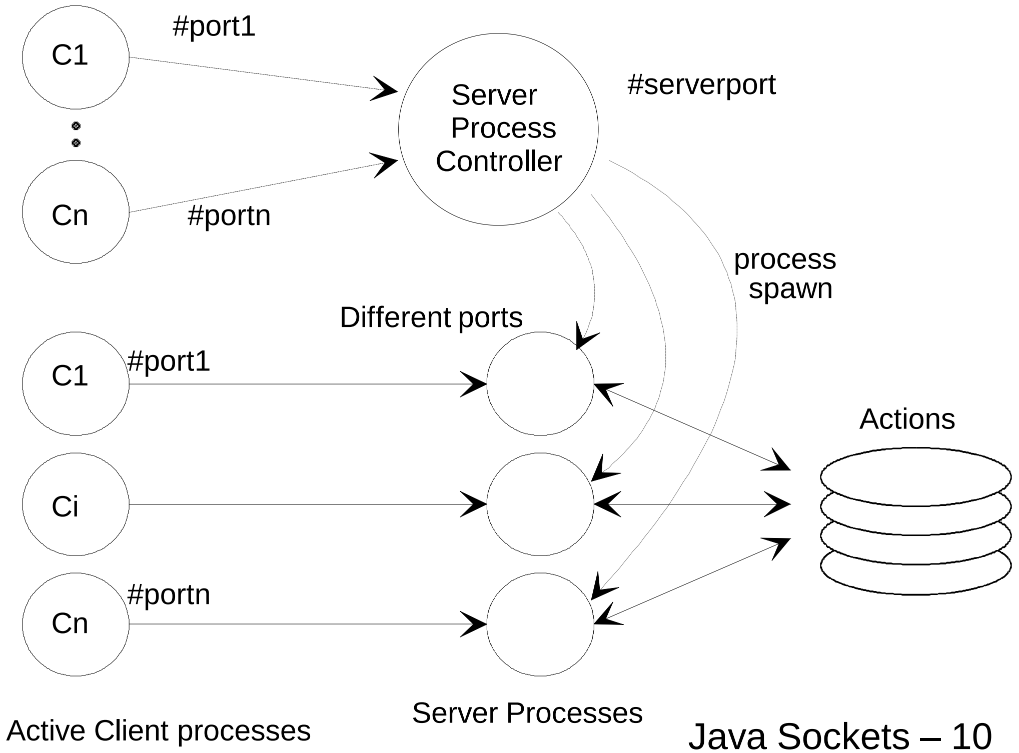
Concurrent server with multiple requests at a time (multi-process)

Uses multiple processes, a master server spawns an internal process for each service

Have to guarantee that the cost due to process spawn does not exceed the gain of having a dedicated process

Solution that uses processes created beforehand to manage fast service requests.

Fixed initial number of processes, other created on-demand and kept running up to a certain time-span.

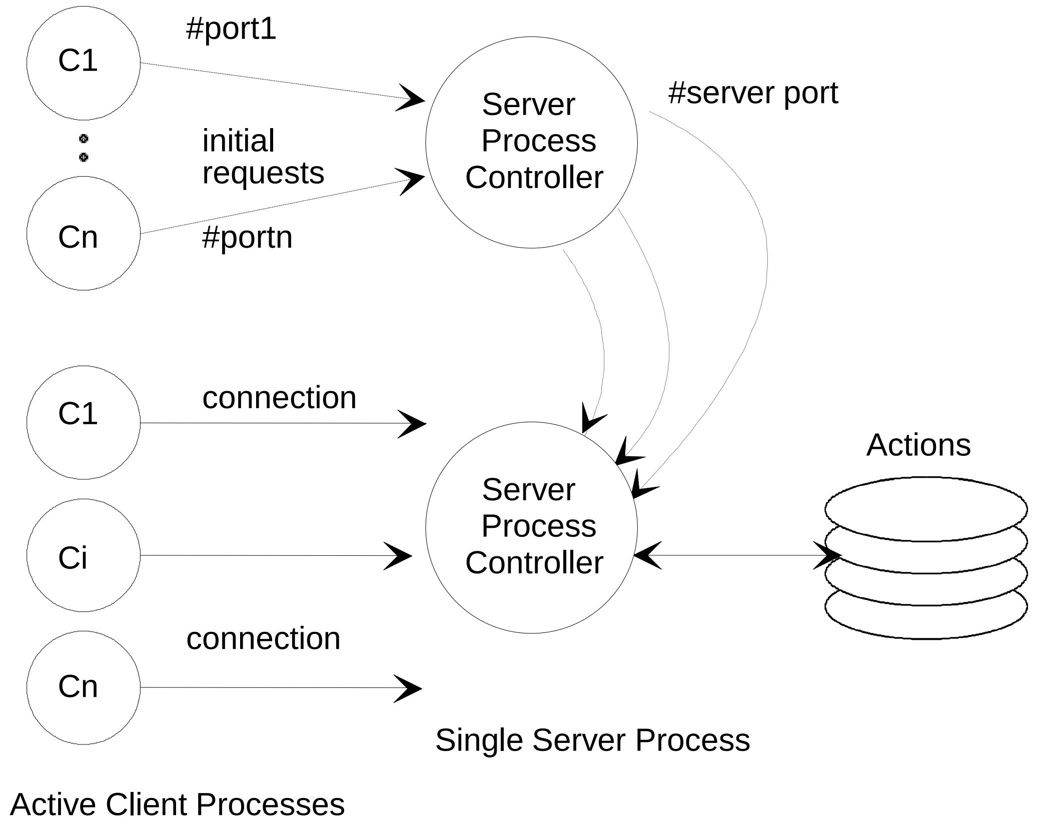


CONCURRENT SERVER

Concurrent server with multiple requests at a time (single-process)

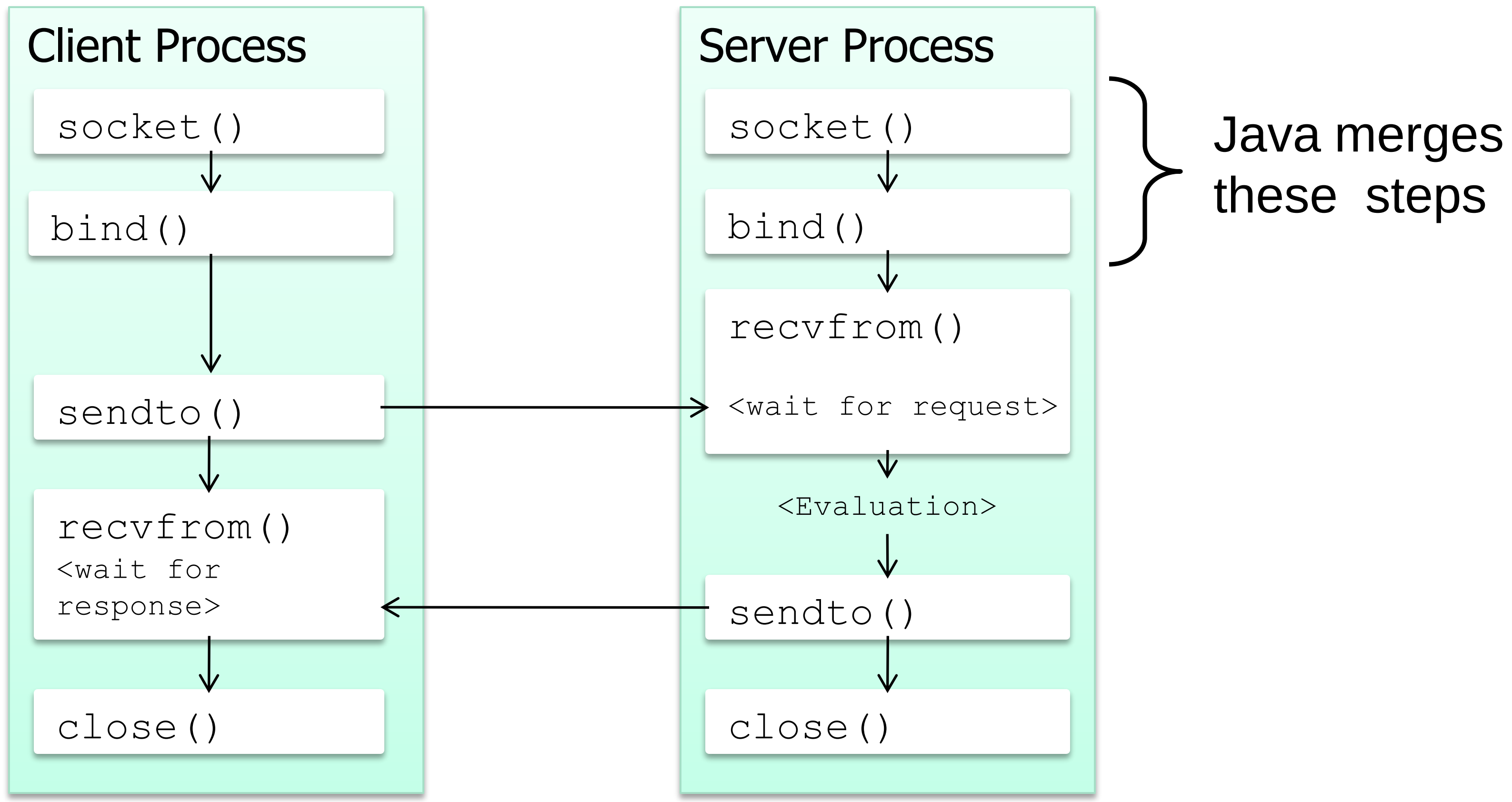
Hard to realize in Java, with a single process that provides different services.

Solution using a **single process server** that manages many requests at the same time



DATAGRAM SOCKET: C-to-Java primitives mapping

Datagrams are **simple messages** that allow Client/Server interactions.



DATAGRAM SOCKET

DATAGRAM sockets enable message exchange between two threads without establishing a connection

It is unreliable and lossy (in case of network problem) and delivery may be out-of-order (due to UDP protocol specification)

Same DATAGRAM socket class for both Client and Server

`java.net.DatagramSocket`

`public final class DatagramSocket extends Object`

One constructors is (other ones available, see Javadocs… ☺):

```
DatagramSocket( InetAddress localAddress,
   int localPort) throws SocketException;
```

DatagramSocket constructor builds a UDP socket and locally binds it to the specified IP address and port: the socket is ready to send and receive packets.

DATAGRAM SOCKET

MESSAGE EXCHANGE using sockets, based on rudimental communication mechanisms: **send** and **receive** user packets

On a `sock` instance of `DatagramSocket` one can:

```
void send(DatagramPacket p);
void receive(DatagramPacket p);
```

These methods are real **communication functions**: *the former sends a message (datagram), the latter blocks until it receives the first available datagram*.

- **send** function assures only that the message is dispatched to the kernel, which will arrange the real send (**asynchronous w.r.t. receive**)
- **receive** function, which delegates real reception to the kernel, blocks the receiver until it gets data (**synchronous with the receiver**)
- A single datagram unblocks the `receive` function.

DATAGRAM SOCKET: SEND and RECEIVE

send and receive functions need an initialized socket and a support structure:

`sock.send(DatagramPacket p);`

`sock.receive(DatagramPacket p);`

used as input by `receive` and as output by `send`.

There are many other support classes

For instance, **`DatagramPacket`** and other constants

- e.g. `integers` for port numbers, `constants` for IP names, integer as IP addresses and as `String` as domain names

`InetAddress`

Many constructors throw specific exceptions:

`SocketException, SecurityException, …`

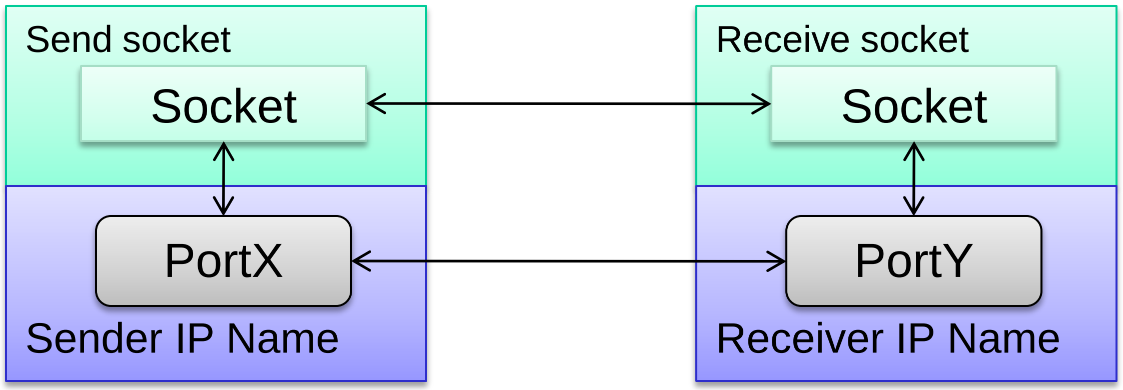
COMMUNICATION MODEL

Before exchanging messages, datagram socket must be correctly created and need to know each other

Sender must insert the receiver address inside the message

Messages need information at different levels:

Application	message (and its size)
Control	destination node and port of the receiving socket



No delivery guarantees due to support protocol (UDP and IP).

SUPPORT CLASSES

`DatagramPacket` class

Class to **build a datagram** that specify what **to send** (data) and **to whom** (control).

Data ⇨ specifies a byte array to write/read

Control ⇨ port value (int) and `InetAddress`

`InetAddress` class that represents IP addresses. It provides public static methods only:

```
public static InetAddress getByName (String hostname);
```

Determines the InetAddress for a given host (passing null returns local loopback address)

```
public static InetAddress[] getAllByName(String hostname);
```

Given the name of a host, returns an array of its IP addresses (when the same logic name corresponds to more than one IP address)

```
public static InetAddress getLocalHost();
```

Returns InetAddress for the local machine.

DATA for DATAGRAMPACKET

`DatagramPacket` contains user level application data

```
DatagramPacket( byte [] buf, // data as byte array
 int offset,                   // start offset
 int length,                   // data length
 InetAddress address, int port); // IP address and
port
```

There are other constructors and utility functions, for example:

`InetAddress getAddress()`	get associated IP address
`void setAddress(InetAddress addr)`	change IP address
`int getPort(),`	get associated port number
`void setPort(int port)`	set associated port number
`byte[] getData(),`	extract data
`void setData(byte[] buf), …`	put data

DATAGRAMPACKET

`DatagramPacket` is a complete container that helps developers

It has different usages when it is being sent or received:

```
sock.send        (DatagramPacketp)
```

– when sending, we must reserve some space where the user can write desired data and some space to record control data relative to the receiver (chosen by the sender)

Only after this set up step, we send the packet...

```
sock.receive     (DatagramPacketp)
```

– when receiving, we must set up needed structures to receive both user and control data

Only after reception, we can work on the packet and extract desired data

A packet can be re-used.

COMMUNICATION SCHEMA

Create socket

```
DatagramSocket socket = new DatagramSocket();
```

Sender:

Prepare data and send

```
byte[] buf = {'C','i','a','o'};
InetAddress addr = InetAddress.getByName("137.204.59.72");
int port = 1900;
DatagramPacket packet = new
 DatagramPacket(buf, buf.length, addr, port);
socket.send(packet); // send packet
```

Other send or receive operations.

COMMUNICATION: RECEIVE

Create socket:

```
DatagramSocket socket = new DatagramSocket(1900);
```

Receiver: **prepare, wait, receive**

```
int  recport;  InetAddress recaddress;
byte[] buf = new byte[200]; byte [] req;
DatagramPacket packet = new
 DatagramPacket(buf, buf.length, recaddress, recport);
packet.setData(buf);  // attach res as application data area of the packet
socket.receive(packet);  // receive packet (synchronous wait)
// extract application and control data from datagram
recport =       packet.getPort();
recaddress =    packet.getAddress();
req =           packet.getData();
// use data
```

COMMUNICATION: RECEIVE AND SEND

Note that upon receiving a UDP packet, the resulting DatagramPacket instance **is associated with the remote host that sent it**.

Thus, if a process receives a DatagramPacket and wants to reply to the sender, it can change the application data of the packet and send it without setting again control data: **the destination host (set automatically by the operating system) will be the host already associated with the packet**.

```
socket.receive(packet); // the received packet is
                        // already associated to
                        // the sender
byte[] replyData = {'r', 'e', 'p','l', 'y'};
packet.setData(replyData);
socket.send(packet);
```

ByteArrayOutputStream and ByteArrayInputStream

Working directly with a byte array (`byte[]`) is time-consuming and error prone (one should set bytes one by one). Java provides **`ByteArrayOutputStream`** and **`ByteArrayInputStream`** as utility classes to write/read data from a byte array.

`Data(Output/Input)Stream` can wrap `ByteArray(Output/Input)Stream` to ease writing/reading Java object on a byte array.

How to write primitive Java types on a byte array stream:

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
DataOutputStream dos = new DataOutputStream(baos);
dos.writeInt(42); // wrap as data stream, and use it
dos.writeUTF("Test string"); // write some data (String)
dos.flush(); dos.close(); // !!! flush and close
byte[] buf = baos.toByteArray(); // buf contains written data
```

More info about DataInput/OutputStream in a few slides…

ByteArrayOutputStream and ByteArrayInputStream

How to read data from a byte stream:

```
byte[] buf = baos.toByteArray();

ByteArrayInputStream bais = new ByteArrayInputStream(buf);
DataInputStream dis = new DataInputStream(bais);
int value = dis.readInt(); // read an int
String text = dis.readUTF(); // read a string
dis.close(); // !!! close stream
```

Note that read and write operations must be symmetrical: (for example) trying to read a String while the array contains an integer will raise an exception or cause undefined behaviour.

DATAGRAM SOCKET OPTIONS

Socket options allow to finely tune sockets behaviour.

receive is blocking (synchronous wait)

`SetSoTimeout (int timeout) throws …`

This options defines a **timeout** in msec, after which the function terminates and throws an exception

If timeout is set to 0, it is disabled (i.e. timeout is equal to infinity)

There are many other (advanced) options.

STREAM SOCKET

A STREAM socket is an end point of a virtual communication channel, established before any data exchange

at-most once semantic
Communication point-to-point between Client and Server, **bi-directional, reliable, data** (byte) delivered **in order at most once** (FIFO, like Unix pipes)
If data is lost??? We can't say much**…**

Connection between Client process and Server process is unambiguously identified by **a unique quadruplet and a protocol**

<Client IP address; Client port; Server IP address; Server port>

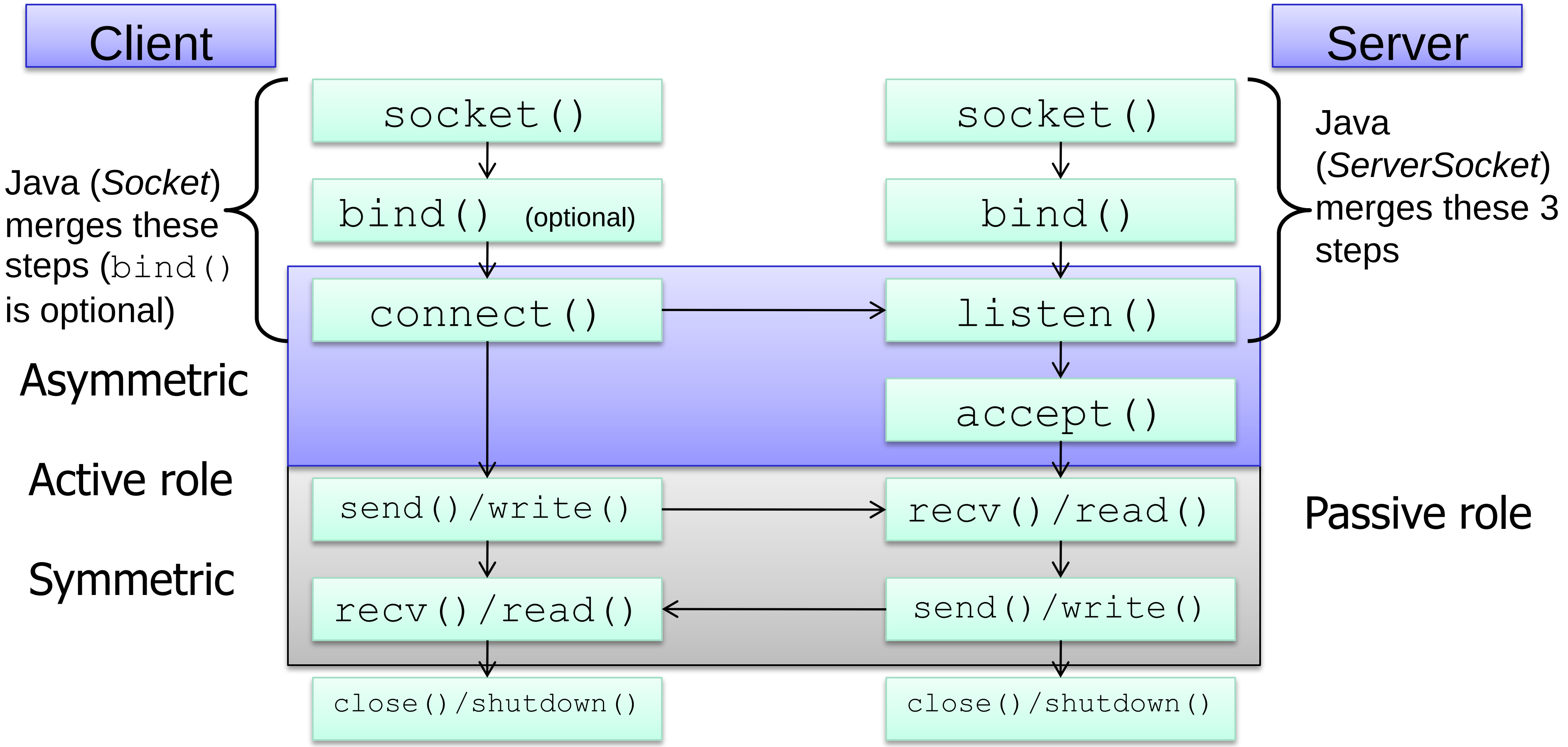
STREAM sockets use TCP (+ IP) as communication protocol

- TCP is the transport protocol and provides the port abstraction;
- IP is the network protocol and provides identifiers for each node

Stream communication between Client and Server needs a connection setup and is asymmetric.

STREAM SOCKET: C-to-Java primitives mapping

Stream sockets create a reliable communication flow between a Server and a Client.

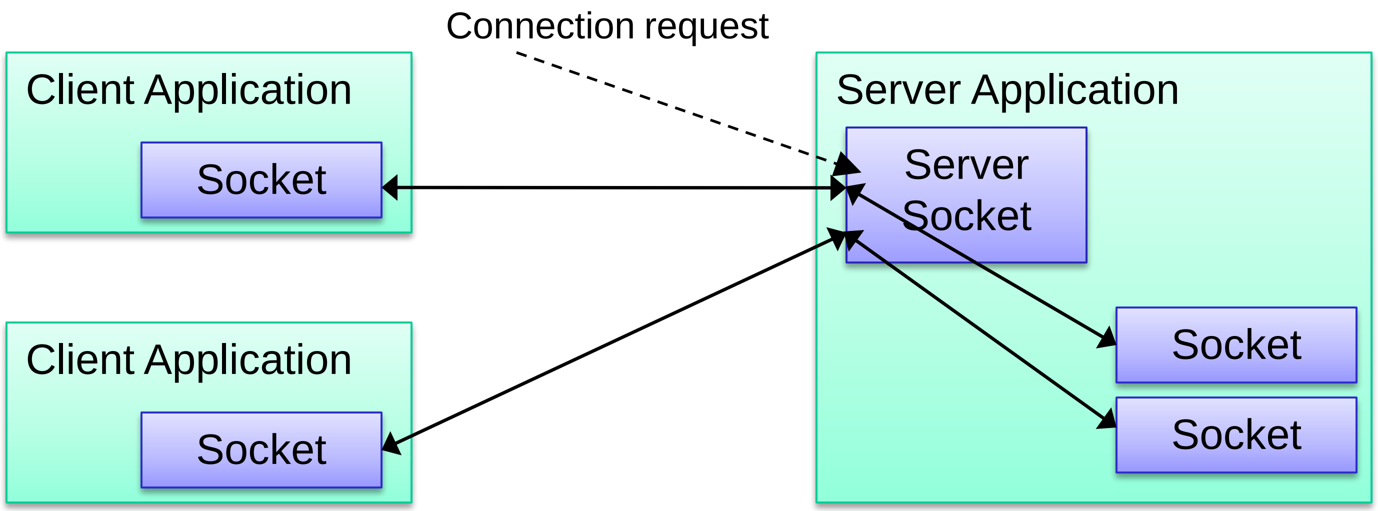


STREAM SOCKET

Java uses two different socket types for different roles: one for both Client and Server processes and one only for Server process.

Different classes for Client and Server

`java.net.Socket` and `java.net.ServerSocket`



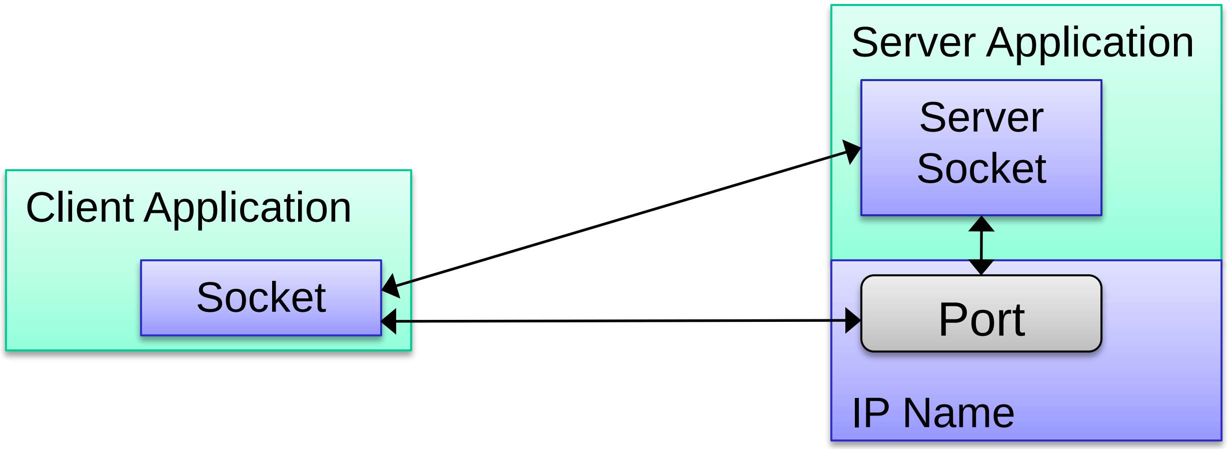
Whenever possible, Java **hides implementation details**, for example by **wrapping many setup steps in constructors**.

STREAM SOCKET: CLIENT

`Socket` class represents an “active”, connected stream socket (TCP) between a Client and a Server

Socket constructors build the socket, bind it to a local port, and connect it to a port on a remote host running the server.

The resulting connection is bi-directional (full duplex)



Creating a socket causes atomically its connection to the server (which must exist).

(Unix API are more complex and complete, see: ***socket, bind, connect****).*

STREAM CLIENT: CONSTRUCTORS

`public Socket(InetAddress remoteHost, int remotePort) throws IOException;`

Builds a client socket stream and connects it to the specified host on the specified port (Unix equivalent: socket, bind, connect)

`public Socket (String remoteHost, int remotePort)throws…`

Builds a client socket stream and connects it to the specified port of the host whose logic name is `remoteHost`

`public Socket(InetAddress remoteHost, int remotePort, InetAddress localHost, int localPort)throws IOException;`

Builds a client socket stream, binds it to a local port (if `localPort` is 0, it is chosen automatically) and connects it to a port of the remote host.

Building a socket atomically causes socket connection to the server (or throws an exception).

CLIENT STREAM: MANAGEMENT

OPEN implicit by the constructor

creating successfully a stream socket implies establishing a bi-directional byte-oriented communication flow (stream) between two processes and requires resources allocation at the two end points.

CLOSE needed to free system resources

Connections are resources: defining and building them is not for free; they must be managed, kept alive, and then freed

Usually: keep only needed connections, close unneeded connections, limit multiple connections

close() method **closes the socket** and disconnects Client from Server

```
public synchronized void close() throws SocketException;
```

CLIENT STREAM: SUPPORT

To get additional information about socket

```
public InetAddress getInetAddress();   // remote
```

Returns remote host’s IP address

```
public InetAddress getLocalAddress(); // local
```

Returns local IP address

```
public int getPort(); // remote port
```

Returns remote port

```
public int getLocalPort(); // local
```

Returns local port

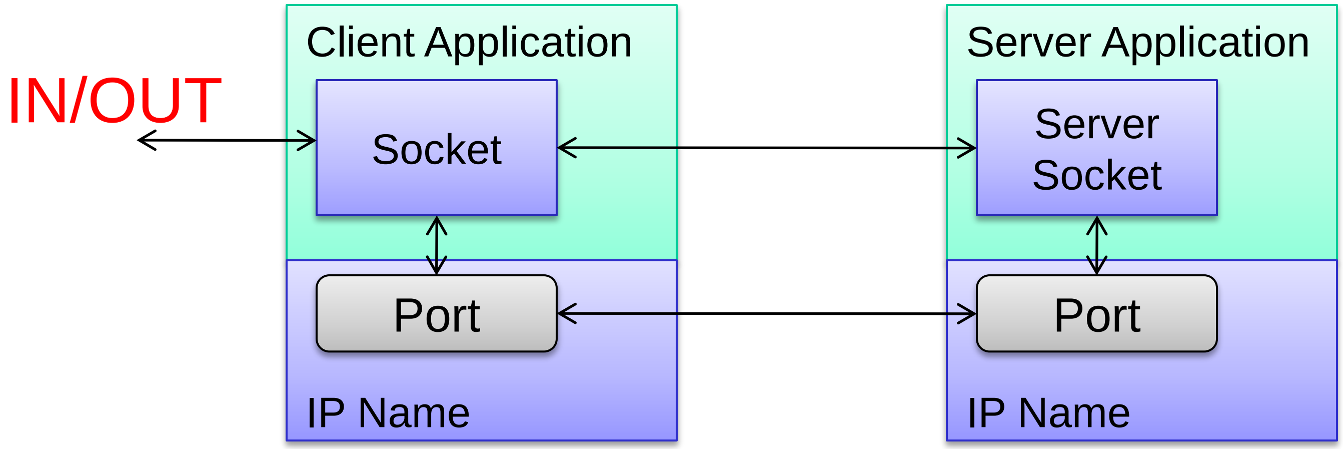
Example:

```
int port = socket.getPort();
```

Additional information about a socket are available runtime.

STREAM: SUPPORT RESOURCES

After building a connection we can send and receive data.
Read/write from/to socket (**IN/OUT**)



JAVA COMMUNICATION STREAM

Read and write from/to a socket after getting stream resources from the socket

```
public InputStream getInputStream()
public OutputStream getOutputStream()
```

These methods return a **stream object** to wrap the communication channel (`InputStream` and `OutputStream` classes)

- A stream can only send/receive raw **bytes**, **without any additional formatting information**
- TCP delivers bytes ordered and without duplicates (a byte is available on the receiving site only if all preceding bytes already arrived); bytes are delivered at most once
- If there is an error? No error information…

A different Java stream can wrap a stream socket to provide high level formatting functions (e.g. `DataInputStream`)

DataOutputStream and DataInputStream

DataOutputStream and DataInputStream provide methods to **send** and **receive primitive Java types**.

Typical usage: define a Client/Server protocols **based on Java** (exchanging Java object instances): **in this class** we use them to create **Java C/S applications (e.g. to exchange int and Strings…)**

Main methods to write/read primitive data:

	DataOutputStream	DataInputStream
String	void writeUTF(String str)	String readUTF()
char	void writeChar(int v)	char readChar()
int	void writeInt(int v)	int readInt()
float	void writeFloat(float v)	float readFloat()
...	...	...

UTF Unified Transformation Format (for strings…)

FileOutputStream and FileInputStream

Java uses streams to abstract file access: **FileOutputStream** and **FileInputStream**, which are specializations respectively of OutputStream and InputStream (generic Output and Input streams)

FileOutputStream inherits from OutputStream:

```
void write(int b) throws IOException
```

Writes the specified byte to this file output stream.

```
void write(byte[] b) throws IOException
```

Writes b.length bytes from the specified byte array to this file output stream.

FileInputStream inherits from InputStream:

```
int read() throws IOException
```

Reads a byte. Returns the next byte of data, or -1 if the end of the file is reached.

```
int read(byte[] b) throws IOException
```

Reads up to b.length bytes of data from this input stream into an array of bytes. Returns the total number of bytes read into the buffer, or **-1 if the end of file is reached**.

If needed, a DataOutputStream can wrap a FileOutputStream and a DataInputStream can wrap a FileInputStream.

FileOutputStream and FileInputStream: example

```
FileOutputStream fos = new FileOutputStream("result.txt");
        // open the file result.dat in the current directory
byte[] buf = {'t', 'e', 's', 't', '1', '2', '3'}; // test data
fos.write(buf); // write the byte array to the fle
fos.flush(); // !!! Flush writings
fos.close(); // !!! close the file

FileInputStream fis = new FileInputStream("result.txt");
byte[] buf2 = new byte[3]; // buffer to read data from file
int numread;
while ((numread = fis.read(buf2)) > 0)
                              // repeat reads until we reach EOF
{
  for (int i = 0; i < numread; i++)
    System.out.println(buf2[i] + " ");
    // print each byte read from the file as an integer value
}
fis.close(); // !!! Close the file
```

STREAM SERVER: ARCHITECTURE

Server side we use the class **`java.net.ServerSocket`**, a `ServerSocket` is a special socket that can only accept connection requests from many Clients

- many pending requests at the same time and
- many ready connections at the same time

  Need to define the length of the queue of incoming connection that wait to be accepted by the server.

  Java ServerSocket constructor wraps many simpler steps visible when using UNIX sockets, like `socket`, `bind`, `listen`.

Connection established when the Server process allows it (**`accept`**)

  `accept` (server side) returns a common **`Socket`** object valid for the connection request just accepted.

SERVERSOCKET: CONSTRUCTORS

Socket server side:

```
public ServerSocket(int localPort)
                        throws IOException, BindException;
```

Constructs a socket listening on the specified port

```
public ServerSocket(int localPort,int count)
```

Constructs a socket listening on the specified port having a connection queue of length `count`

Server is “passive”: it creates a connection queue and waits for clients

Server becomes active when it explicitly accepts a connection.

Queued connection requests are not accepted automatically, server uses a specific API to express its will to accept a connection.

SERVERSOCKET: ACCEPT

Server **waits for connection requests** calling **`accept()`**

`public Socket accept() throws IOException;`

`accept` **blocks the Server** until at least one connection request is pending

 `accept` returns a **`Socket`** instance which allows communication between Client and Server

For new incoming connection requests, **accept creates a new socket for the transport connection already created with the Client: the new Socket** returned by accept is the real stream

 accept **blocks the caller**, until a new connection request arrives

- If there are no pending connection requests, the server blocks
- If there is at least one pending connection request, accept unblocks and creates the connection (connection request is removed from incoming connection requests queue)

STREAM SERVER: SUPPORT

Data transmission is done the same way by both Client and Server

Connection endpoints are completely homogeneous
(property derived from TCP)

Information about already connected socket:

```
public InetAddress getInetAddress();  // remote
```

Remote address

```
public InetAddress getLocalAddress(); // local
```

Local end point address

```
public int getPort(); // remote port
```

Remote end point port

```
public int getLocalPort(); // local
```

Local end point address

PARALLEL SERVER

A **parallel server, …**

After accepting a connection, the server **spawns a new activity (Thread) that manages the service**, inherits the new connection, and closes it at the end of operations

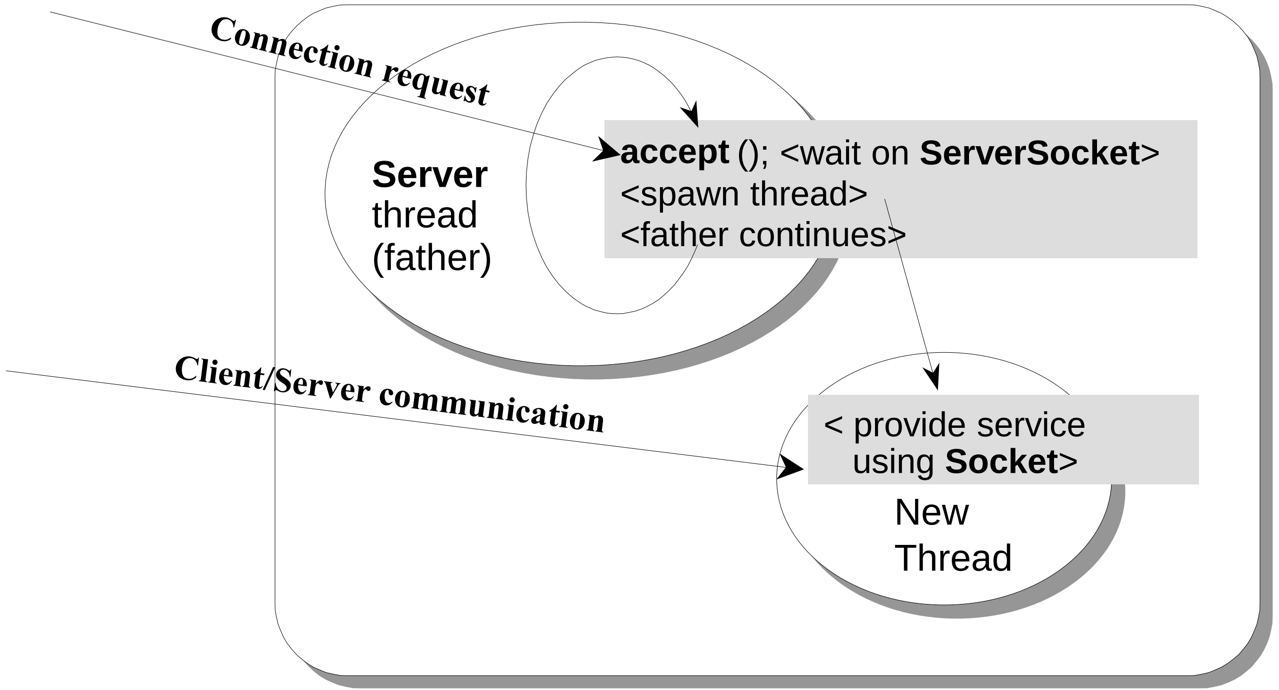
The **main server immediately waits** for new connection requests.

PARALLEL

MULTIPROCESS

CONNECTED

SERVER


Connection request
Server thread (father)
accept (); <wait on ServerSocket>
<spawn thread>
<father continues>
Client/Server communication
< provide service using Socket>
New Thread

Example C/S STREAM PROTOCOL

Remote CoPy (RCP) is a distributed Client/Server application that copies a file from C to S

- Design both the Client and Server applications.

Client application must be started as:

rcp_client   servernode serverport filename destfilename

- **servernode and serverport** identify the **Server, filename** is the filename of a file existent and readable in the Client filesystem.
- The Client process must send the file **filename** to the Server, that must write it into the current directory as **destfilename**.

Connection oriented: Client wants to transfer a lot of data, each byte must be received in order and at-most-once.

The Client/Server connection is used for both **coordination** and **data transfer**.

Example: RCP CLIENT / 1

RCP Client – code snippets... UTF: standard text representation format (use of write/readUTF to write and read Strings)

```
try {
rcpSocket = new Socket(host, port);
outSocket = new DataOutputStream(rcpSocket.getOutputStream());
inSocket = new DataInputStream(rcpSocket.getInputStream());
outSocket.writeUTF(destFilename);
String resp = inSocket.readUTF();
System.out.println(resp);
if (response.equalsIgnoreCase("MSGSrv: waitingForFile")) {
// send file
```

Example: RCP CLIENT / 2

```
sourceFile = new FileInputStream(localFName);
byte[] buf = new byte[1000]; // buffer to read a fixed portion
                             // of the file
int numbytes;
while ((numbytes = sourceFile.read(buf)) > 0) {
            // read up to buf.length bytes and put the number
            // of read bytes in numbytes
            // Repeat reading until we consume the whole file

    outSocket.write(buf, 0, numbytes); // send read bytes
}
rcpSocket.close(); // close socket

catch (IOException e) { ... }
```

RCP: SEQUENTIAL SERVER / 1

```
try { // ...
ss = new ServerSocket(serverPort, 5);
System.out.println("Listening on port " + ss.getLocalPort());
while (true) {  // endless loop
  Socket s = ss.accept();  // wait for a connection
  System.out.println("Connection "+ s);
  inSock = new DataInputStream(s.getInputStream());
  outSock = new DataOutputStream(s.getOutputStream());
  String filename = inSock.readUTF();
  System.out.println("Recevied file request: " + filename);
  outSock.writeUTF("MSGSrv: waitingForFile");
```

RCP: SEQUENTIAL SERVER / 2

```
FileOutputStream fileOut = new FileOutputStream(filename);
  byte[] buf = new byte[200];
  int numbytes;
  while ((numbytes = inSock.read(buf)) > 0) // same as before
    fileOut.write(buf, 0, numbytes);
  s.close();
  fileOut.flush(); fileOut.close(); // ! flush output file and
close it
} catch (IOException e) { ... }
```

RCP: PARALLEL SERVER

```
… try {…
rcpSocket = new ServerSocket(port);
System.out.println("Listening port:"+rcpSocket.getLocalPort());
while(true)
{ rcpSocketConn = rcpSocket.accept();
  serviceThread = new ServiceRcp (rcpSocketConn);
  serviceThread.start();
} }
catch (IOException e) {System.err.println(e);}
```

Create a new process for each accepted connection.

Calling **close** on a socket closes it (if many threads have a reference to the same socket and any of them calls close, the socket is closed for all of them)

??? How many opened socket per process?

RCP: SERVER THREAD / 1

```
public class ServiceRcp extends Thread { ...
ServiceRcp(Socket incoming){rcpSocketSrv = incoming;}
public void run() {
System.out.println("thread #" + Thread.currentThread());
System.out.println("Connection: " + rcpSocketSrv);
try
{outSocket= new DataOutputStream
                (rcpSocketSrv.getOutputStream());
 inSocket = new DataInputStream(rcpSocketSrv.getInputStream());
 fileName = inSocket.readUTF();
 String filename = inSock.readUTF();
```

RCP: SERVER THREAD / 2

```
  System.out.println("Recevied file request: " + filename);
  outSock.writeUTF("MSGSrv: waitingForFile");
  fileOut = new FileOutputStream(filename);
  byte[] buf = new byte[200];
  int numbytes;
  while ((numbytes = inSock.read(buf)) > 0) { // same as before
    fileOut.write(buf, 0, numbytes); }
  /* free resources*/
  rcpSocketSrv.close();
  fileOut.flush();
  fileOut.close(); // ! flush output file and close it
  } catch (IOException e) { ... }
  System.out.println("End of service #" +
                 Thread.currentThread());
} catch (IOException e)
 { System.err.println(e); exit(1);} …
```

SOCKET CLOSE

A Java socket requires application level resources, in addition it requires other operating system level resources, which are allocated to it until the application calls `socket.close()`

It is crucial to close socket to let the system know **that it can free the resource associated to the socket itself**

- **After closing a socket, some resources remain allocated for some time** (depending on importance of pending operations, the input buffer is not saved)
  - For **closed connected sockets**, the output buffer is saved to keep sending data to peers
- A peer can detect a closed socket thanks to exception, functions and events, that are notified if it tries to read or write data from/to the socket

Any peer can close the socket.
Socket close impacts the peer.

GRACEFUL SOCKET CLOSE

Closing a socket means closing two communication flows

Each peer is more responsible for one of the two flows: the read operations of the other peer depend on its output stream.

There are methods to close only a flow of a connection:
`shutdownInput() and shutdownOutput();`

Usually Java applications use `shutdownOutput()`

If a peer shuts down its output flow, the output buffer is saved in memory to send remaining data to the other peer, the input memory depends on the actions of the other peer.

SOCKET OPTIONS

Socket options allow to fine tune sockets behaviour

Socket options for stream sockets:

`SetSoLinger` `(boolean` **`on`**`, int` **`linger`**`)`

After closing a socket, the system tries to deliver packets still in the output queue. This options discards such packets after **`linger`** seconds.

`SetTcpNoDelay` `(boolean` **`on`**`) throws ...`

Output data is sent **as soon as possible, without buffering**

`SetKeepAlive` `(boolean` **`on`**`) throws...`

Enables and disables the **keepalive** option

All supported options are declared in the `SocketOptions` interface, with provides both get and set methods.