Intro to Distributed Systems. Networking

TCP Sockets in Java

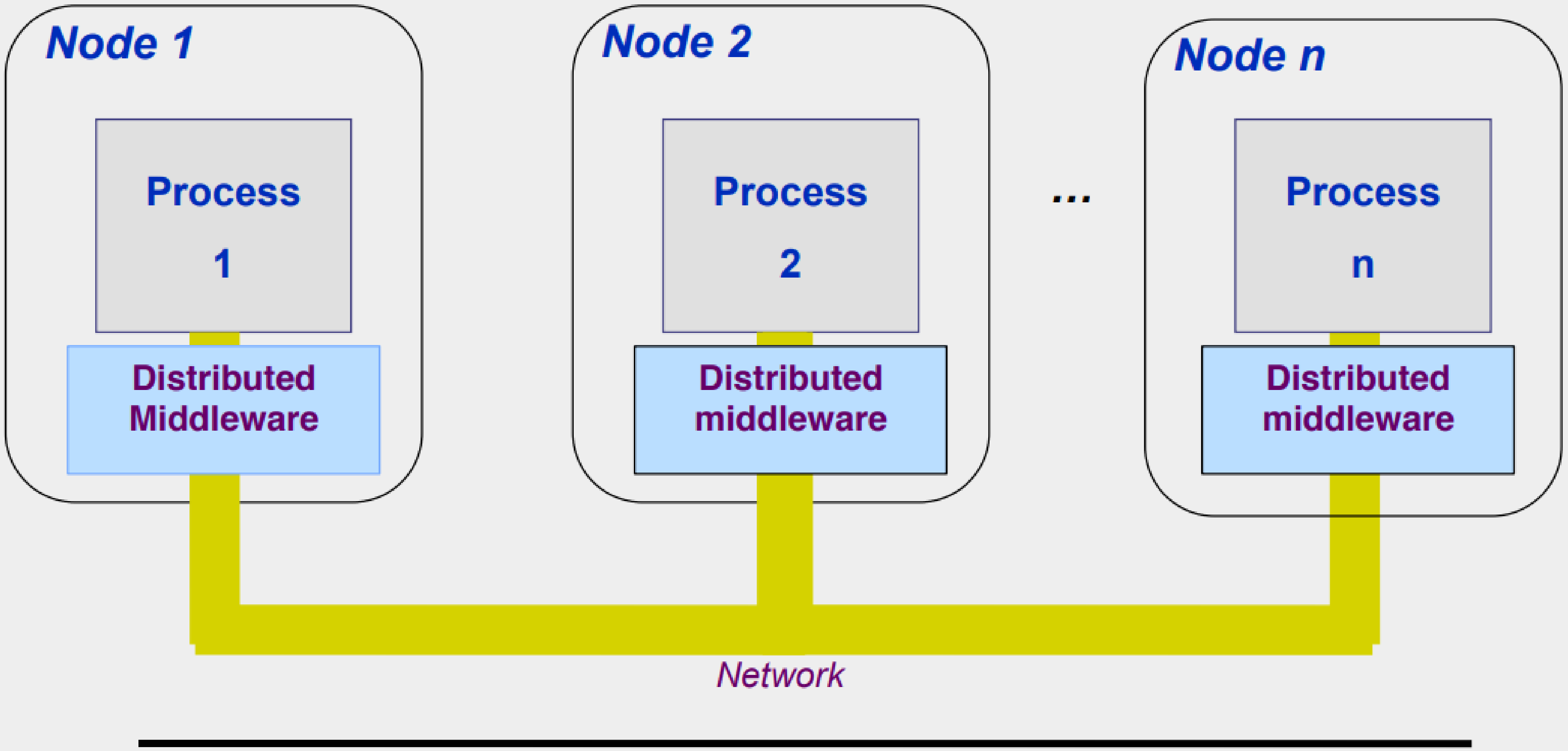
Bibliography

- https://docs.oracle.com/javase/tutorial/networking/sockets/index.html

Distributed Systems

- *“A distributed system is one where you can't get your work done because some machine you've never heard of is broken.“*
  ([Leslie Lamport](), 2013 Turing Award for his seminal work in distributed systems)
- A distributed system is made of:
- A set of processes running in separate address spaces (either on the same machine or on distributed ones)
  - Communicating through a network: Classically components communicate through a distributed middleware providing communication facilities
  - Collaborating to a common goal

Distributed System



Distributed System

- A very broad definition: A set of autonomous processes communicating among themselves to perform a task
- Advantages:
  - Resource Sharing
  - Higher Performance
  - Fault Tolerance
  - Scalability
- Issues:
  - Un-reliability of communication
  - Lack of global knowledge
  - Lack of synchronization and causal ordering
  - Concurrency control
  - Failure and recovery

Topics in Distributed Systems

- Networking – Communication Basics
- Distributed Systems Architectures
- Distributed algorithms

Topics in Distributed Systems

- Networking – Communication Basics
  - enabling communication between devices.
  - needs protocols that define how data is formatted, transmitted, received, and interpreted.

Topics in Distributed Systems

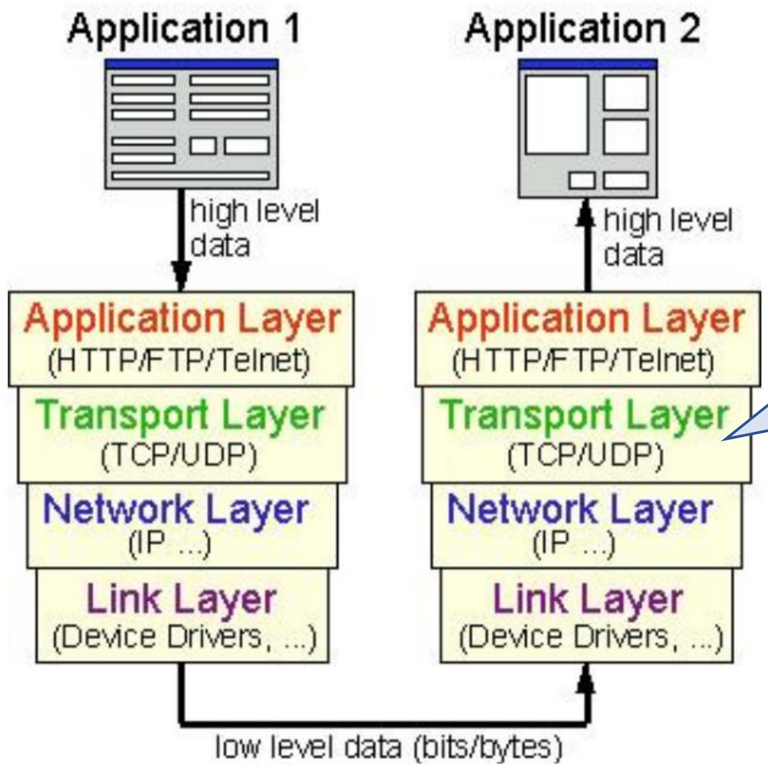
- Distributed Systems Architectures
  - How multiple computers (nodes) work together; How are tasks spread across many computers.
  - Communication Protocols and Middleware:
    - Software that helps manage communication between nodes, providing services like message queueing, remote procedure calls, distributed transactions.
    - It includes application servers, messaging and similar tools that support application development and delivery.
    - Known example for this class: MPI middleware

Topics in Distributed Systems

- Distributed algorithms:
  - separate parts of the algorithm are run simultaneously on *independent* nodes, and having *limited information* about what the other parts of the algorithm are doing.
  - Challenges: coordinating the behavior of the independent parts of the algorithm in the possibility of ***processor failures*** **and** ***unreliable communications links***.
    - Distributed algo problems: Consensus, Leader election, Flooding
    - The Byzantine Generals Problem: a game theory problem, which describes the difficulty decentralized parties have in arriving at consensus without relying on a trusted central party.
      - In a network where no member can verify the identity of other members, how can members collectively agree on a certain truth?
      - Applies to *descentralized* systems
      - Blockchain
    - Protocols for solving consensus: Paxos, Raft
    - Apache Zookeeper: an open-source coordination and synchronization service for distributed application.
      - Leader election algorithm
      - Initially developed at Yahoo
      - Used by: Hadoop, Kafka; Use the protocol: Meta, Netflix, Twitter

Intro Networking

Network Protocol Layers



TCP: connection based protocol, provides a reliable point-to-point communication channel for applications.
UDP: connectionless, datagram.
Faster but not reliable

IP (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 (example: 193.226.12.16)
  - 16 bytes(IPV6)
  - IP addresses are location dependent

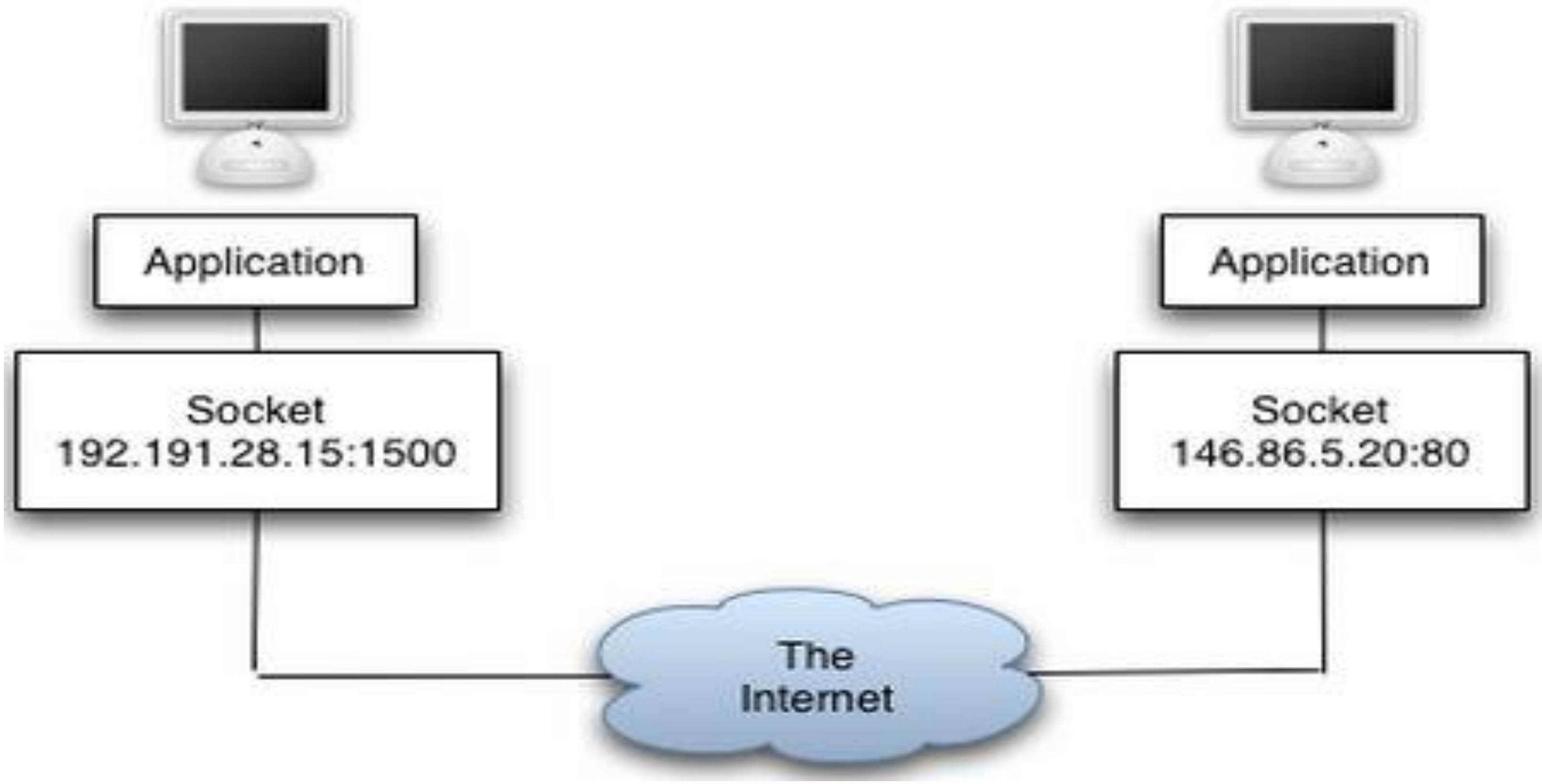
DNS (Domain Name System)

- IP Address resolution
  - manages the translation between a host name and its IP address
- DNS design
  - DNS is a fairly complex world-wide distributed system in itself
  - Allows name aliases (multiple names for an address)
  - 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
- Example: staff.cs.upt.ro 193.226.12.16
- localhost 127.0.0.1 = the loopback IP address

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**
- One process can use several ports at a time
- A computer usually has a single physical connection to the network (IP address), but different applications can open their own communication channels using different ports on the same physical network connection.

Communication channels



Sockets

- A communication channel is defined by:
  - 2 communication endpoints (Sockets)
  - the protocol
- A communication endpoint (Socket):
  - Address: has 2 components: host (IP address) and port
- Sockets allow you to exchange information between processes on the same machine or across a network
- Socket APIs are offered by the operating system

Socket Protocols

- A *protocol* is a standard set of rules for transferring data
- Sockets are classified according to communication properties. Processes usually communicate between sockets of the same type.
- Type of sockets: describes the semantics of communications using that socket. The socket type determines the socket communication properties such as reliability, ordering, and prevention of duplication of messages.

- Stream sockets. TCP protocol
  - Stream oriented: components exchange streams of bytes
  - Lossless: 0 bytes lost
  - Ordered: 0 bytes reordered
  - Connection-oriented
- Datagram sockets. UDP protocol
  - Packet based: components exchange messages
  - Not reliable, not sequenced: packets may be lost or reordered
  - Efficient

Applications over TCP and UPD

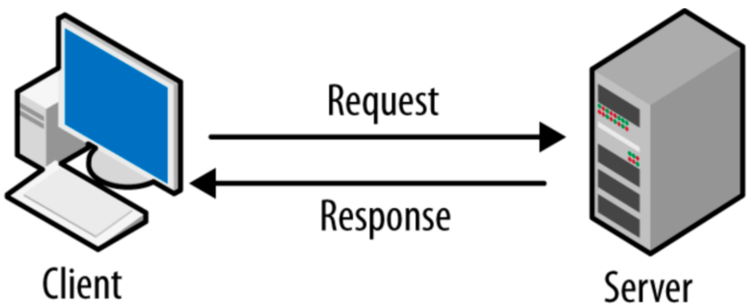
- TCP
  - Applications that do not support loss or reordering
  - Transferring files (ftp)
  - Downloading web pages
  - ..
- UDP
  - Applications requiring high bandwidth & accepting loss or reordering
  - Transmission of video/sound in real time
    - Ex: VoIP (Skype)
    - Out of sequence or incomplete frames are just dropped

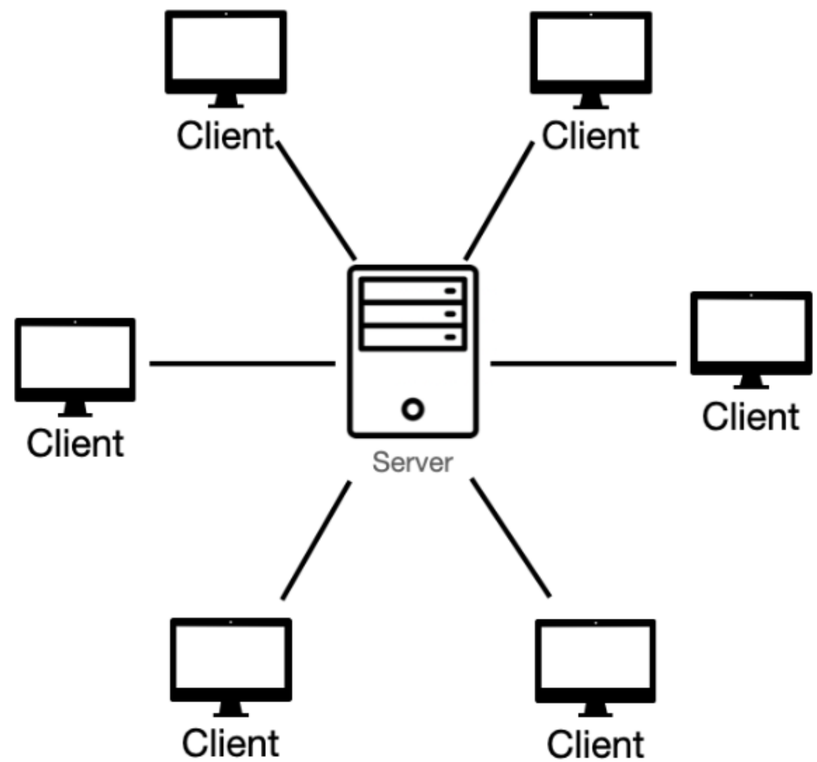
Types of Distributed Architectures

- Client-Server
- Peer-to-Peer

Basic Client-Server Architecture

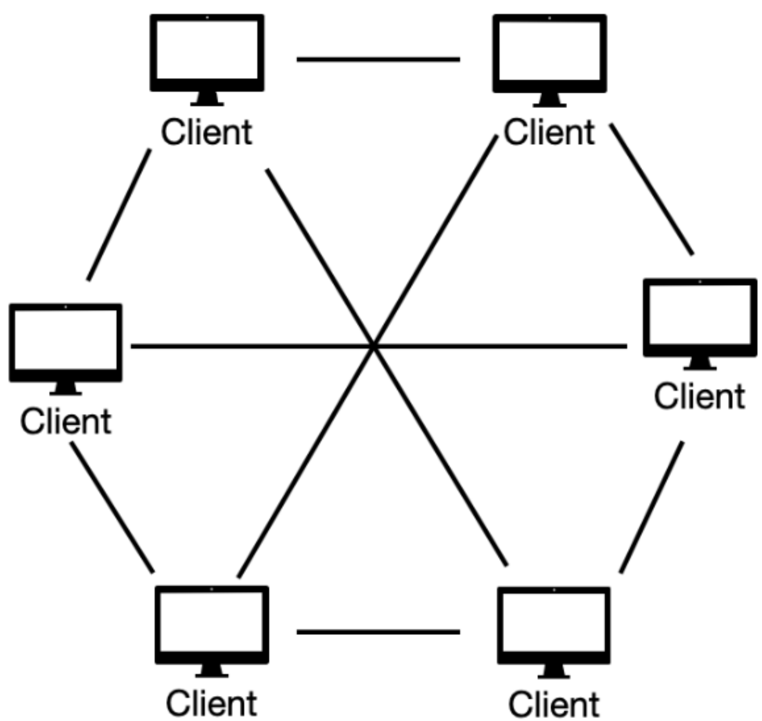
- Clients use some services provided by a server
- The services conform to a contract (an interface)





Peer-to-Peer Architectures

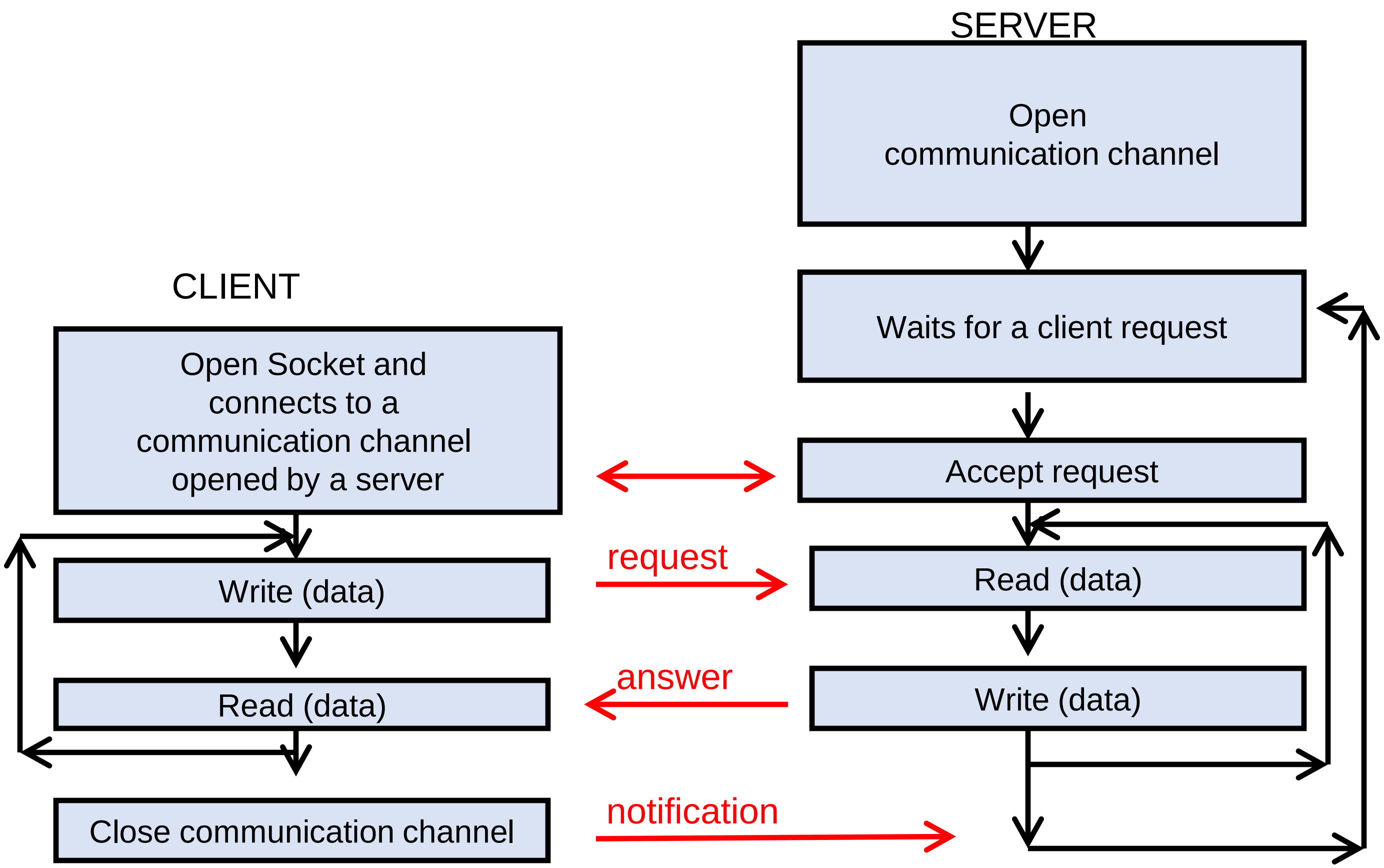
- Decentralized architecture (no central server)
- Software components play the role of both client and server



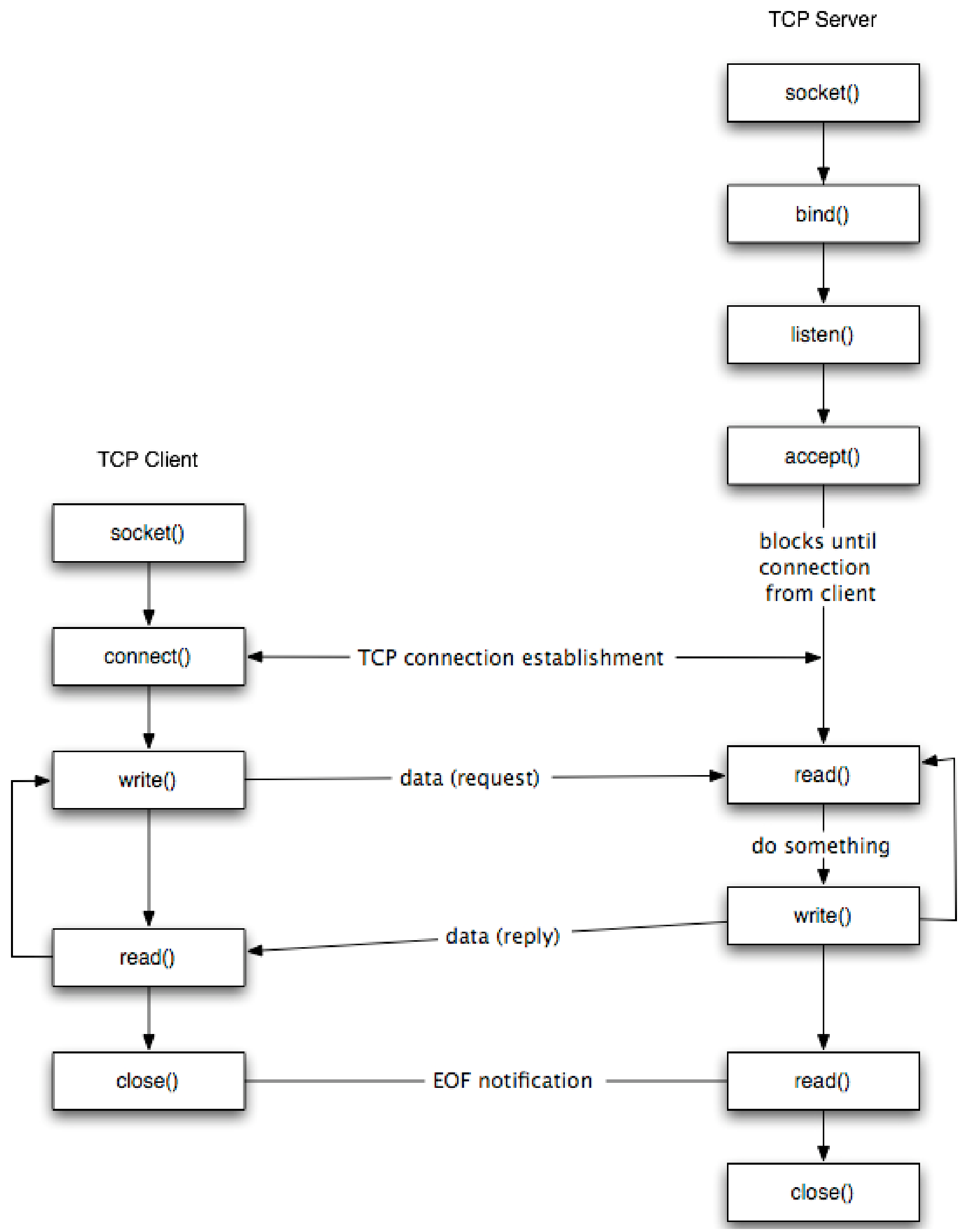
Interaction pattern for network applications

- Client-Server:
  - A server is an application that provides a "service" to various clients who request the service
  - A server must have an addressable communication endpoint (a server socket)
  - Clients have simple ephemeral communication endpoints (sockets)
- Peer-to-Peer:
  - Participants are equal, all can be clients or servers at any time

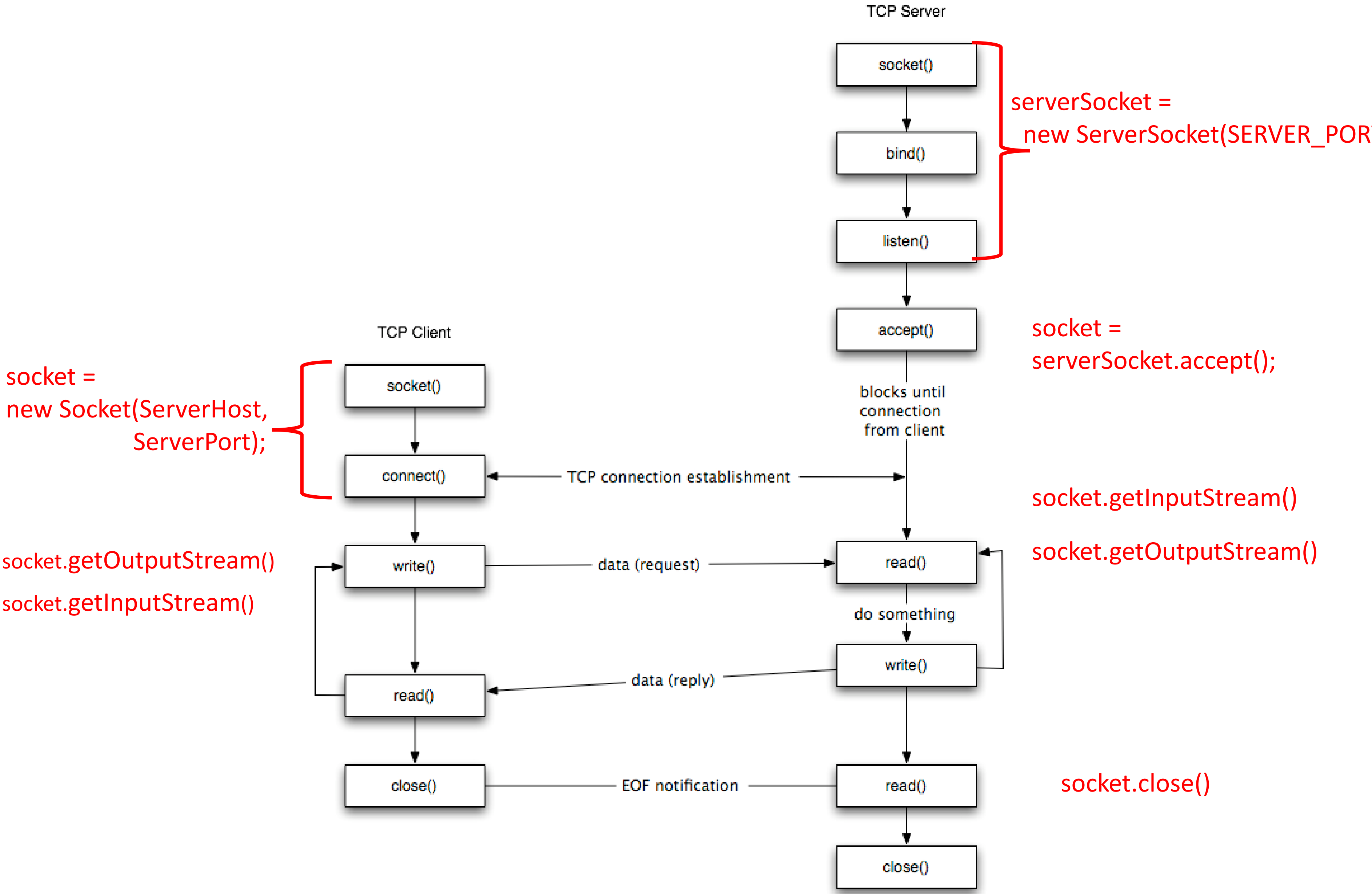
Typical Client-Server Interaction



The API for TCP Sockets in the Unix OS



The Socket API in Java



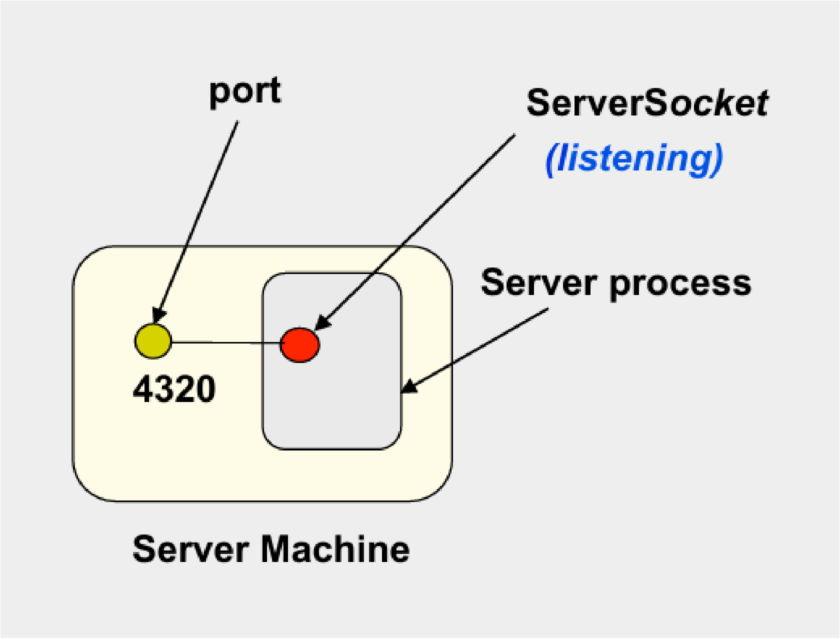
TCP Sockets in Java

TCP Sockets in Java

- The java.net package
- Class Socket: implements one side of a two-way connection between your Java program and another program on the network.
- 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)
- Class ServerSocket: implements a socket that can *bind* to an address, *listen for* and *accept* connections
  - Configured with a backlog (maximum number of queued connection requests, to avoid queuing too much connection requests)
- Other utility classes (InetAddress, SocketAddress, ..

Step 1- Server side:

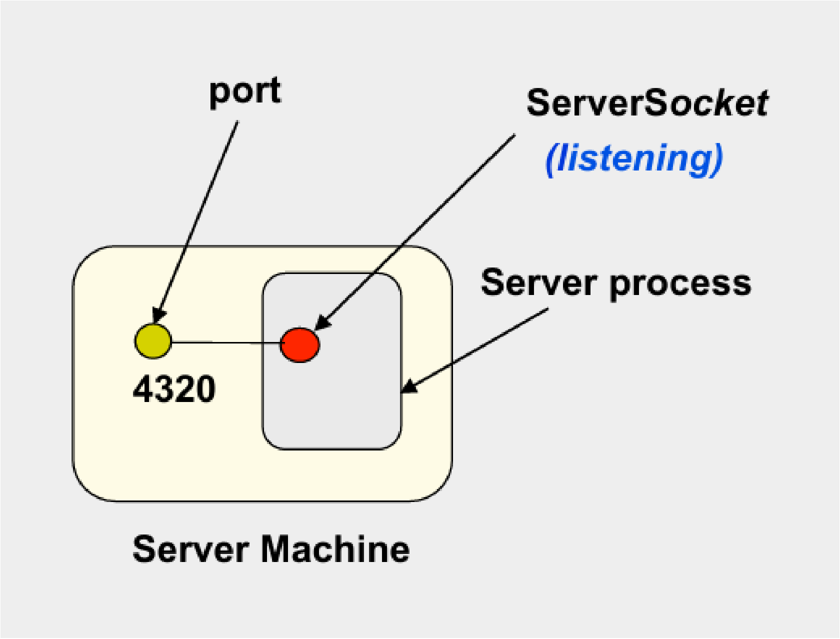
- The port of the server is decided (example: 4320)
- Create a ServerSocket on the desired port to listen to connection requests
- `ServerSocket listenSoc = new ServerSocket(port, backlog);`



https://lig-membres.imag.fr/boyer/html/Documents/cours/AR/Sockets.pdf

Step 2- Server side:

- Loop: wait for a connection request
- `Socket soc = listenSoc.accept(); //blocking call`

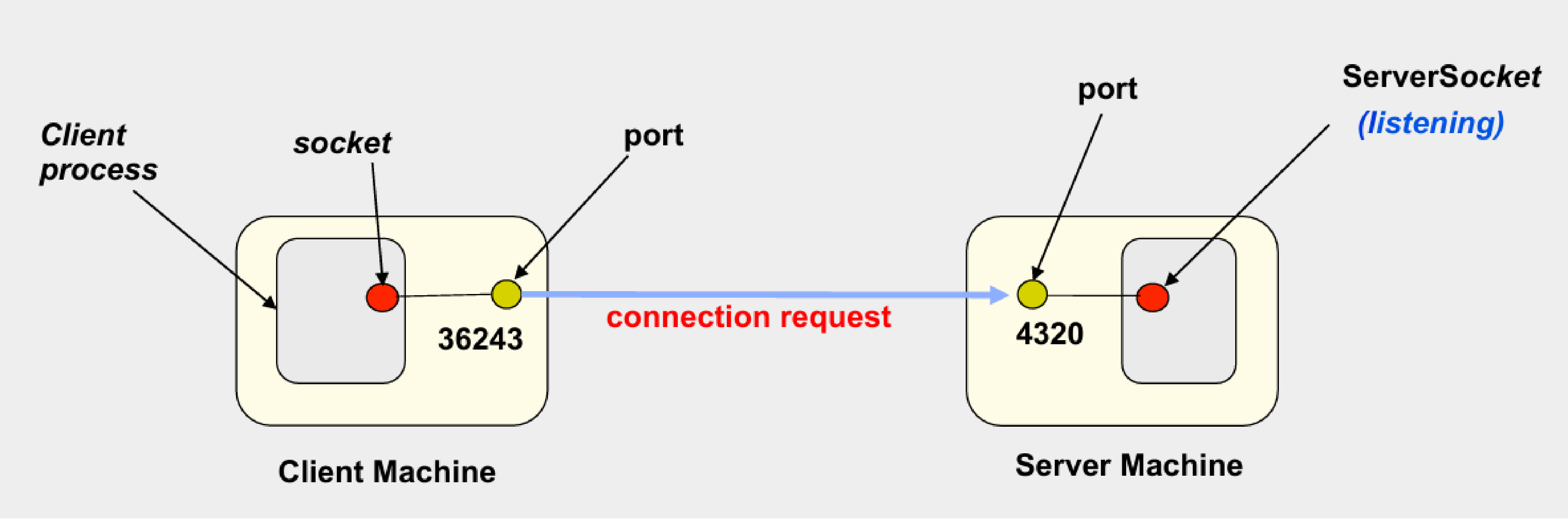


https://lig-membres.imag.fr/boyer/html/Documents/cours/AR/Sockets.pdf

Step 3 - Client side:

- Create a Socket to connect to the server socket (automatically allocates a port & sends a connection request to the server)

```
Socket soc = new Socket(serverHost, serverPort);
```

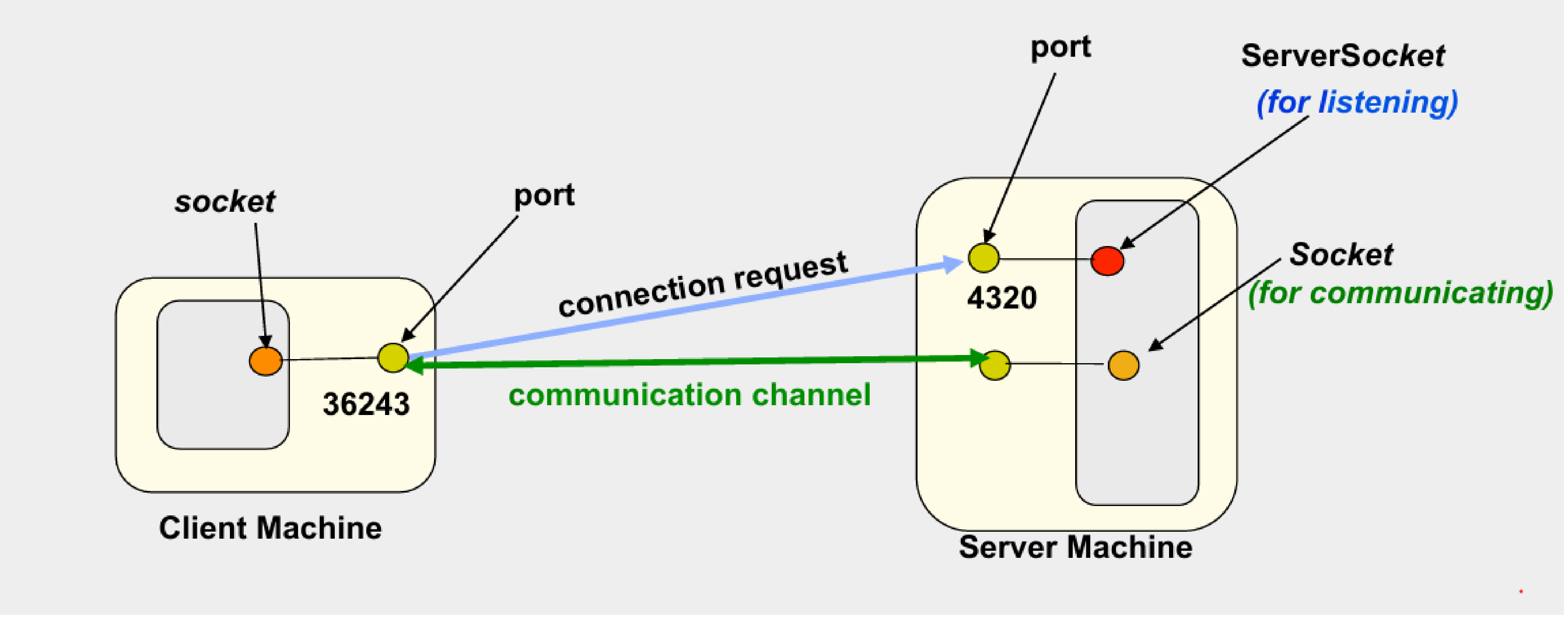


https://lig-membres.imag.fr/boyer/html/Documents/cours/AR/Sockets.pdf

Step 4- Server side:

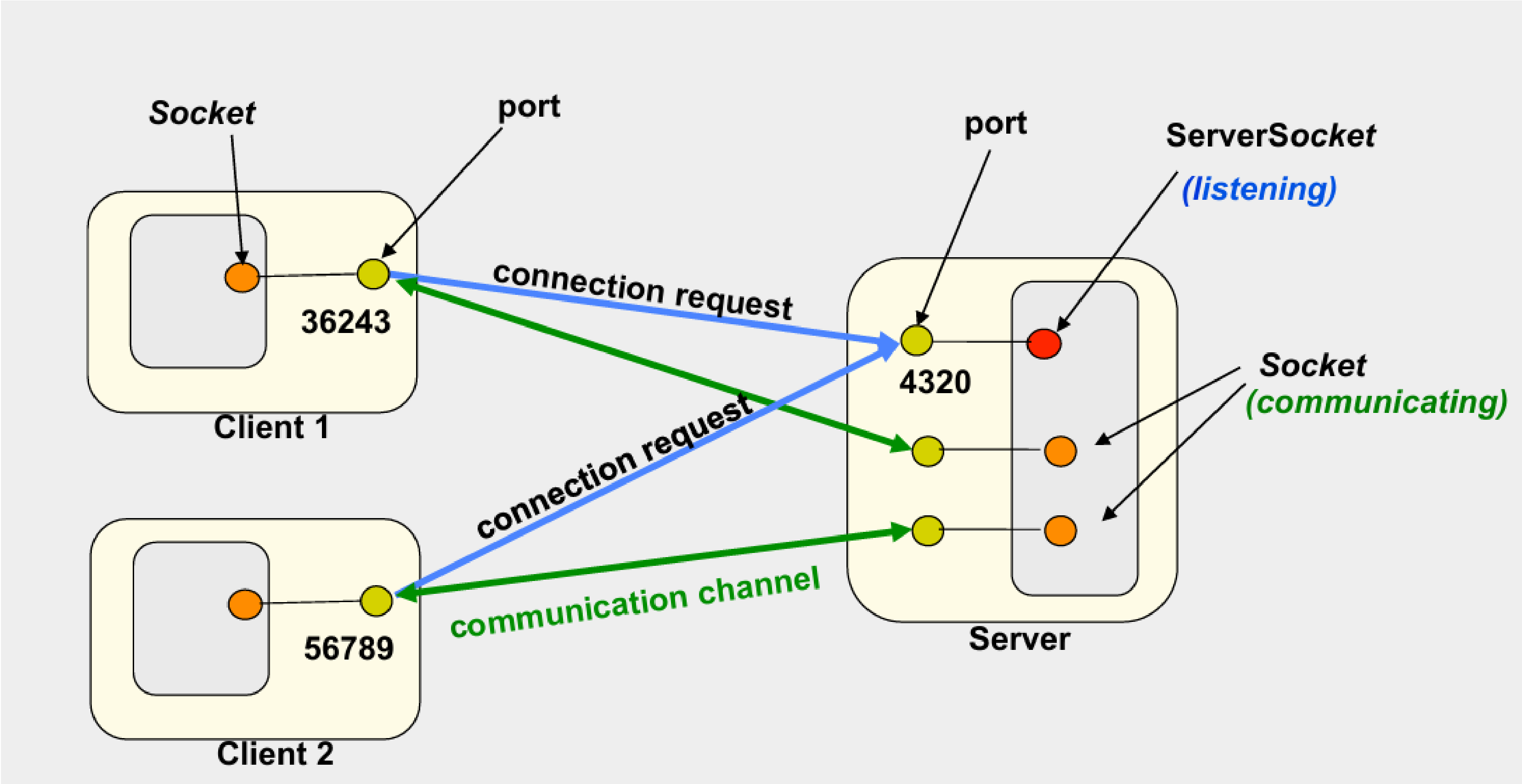
- Accept the connection from the client (a port/Socket is automatically allocated to communicate with the client)

```
Socket soc = listenSoc.accept();
```



https://lig-membres.imag.fr/boyer/html/Documents/cours/AR/Sockets.pdf

Several Clients connected to a Server



https://lig-membres.imag.fr/boyer/html/Documents/cours/AR/Sockets.pdf

Typical TCP Server in Java

```
int port = 4320; // Example Port for the server to listen on
int backlog = 3; // Maximum number of clients to wait for in the backlog queue

ServerSocket listenSoc = new ServerSocket(port, backlog);

// Server loop to handle incoming connections
while (true) {

  // Wait for a connection request
   Socket soc = listenSoc.accept(); // blocking call

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

   // Close the connection with the client
   soc.close();
}
```

Typical TCP Client in Java

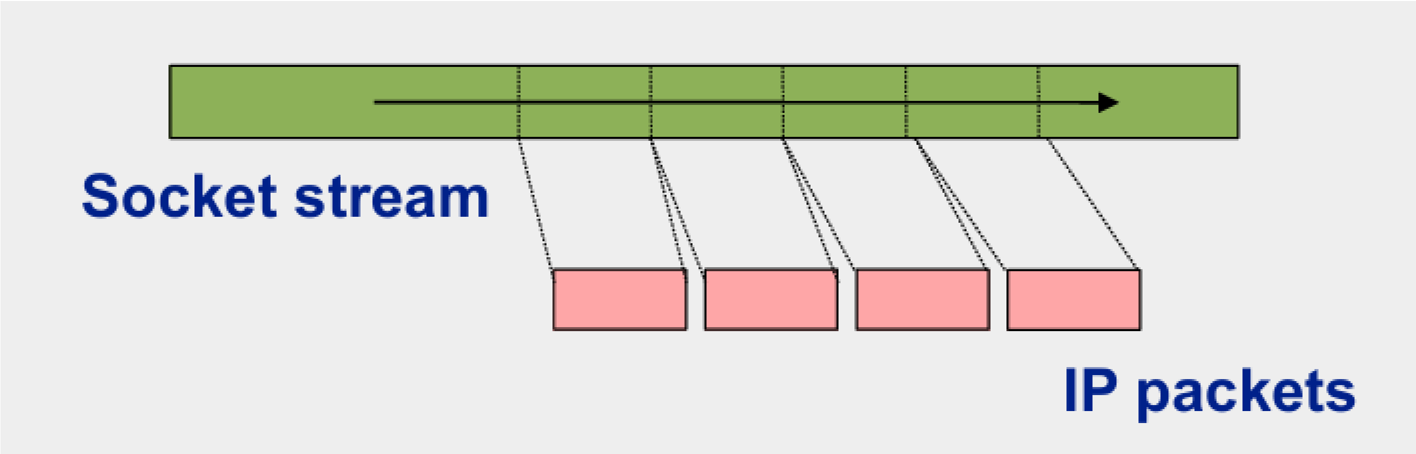
```
String serverHost= “bla.com”; //Host where the server is located
 int serverPort = 4320; // Port where the server listens

// connect to the Server
 Socket soc = new Socket(serverHost, serverPort);

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

Sending Bytes over a TCP Socket

- The receiver must be able to check that all bytes have been received:
  - Send the length of data as prefix (or use a marker at the end of data)
  - When writing the length, it must be endianness proof



Sending Bytes over a TCP Socket

- Over the socket we send BYTES, but these bytes come from an encoding
- Both parties must be in sync with the encoding scheme: sending and reading data must use the same encoding scheme
- Supported encodings: link doc
- UTF8 or UTF16

Sending Bytes over a TCP Socket in Java

- InputStream / OutputStream can be wrapped in upper streams
- DataInputStream / DataOutputStream
  - allows to read Java primitive types, lines of text, or bytes
  - readInt(), readChar(), readDouble(), readLine(), read()
- ObjectInputStream / ObjectOutputStream
  - Reads/writes serializable Java objects

Examples: Send Length of Bytes and Set String Encoding

```
// Participant 1
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);

// Participant 2
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");
```

Example: Use 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();  // "\n" is used as the end-mark in readLine()
    bw.write(line);
    bw.newLine();
    bw.close();
 }
```

TCP Echo Service

- The Echo protocol: https://datatracker.ietf.org/doc/html/rfc862
  - A very useful debugging and measurement tool is an echo service. An echo service simply sends back to the originating source any data it receives.
- Echo protocol established by RFC 862:
  - One echo service is defined as a connection based application on TCP. A server listens for TCP connections on TCP port 7. Once a connection is established any data received is sent back. This continues until the calling user terminates the connection.
- See also https://docs.oracle.com/javase/tutorial/networking/sockets/readingWriting.html

```java
public class EchoServer {
    public static void main(String[] args) throws IOException {

        if (args.length != 1) {
            System.err.println("Usage: java EchoServer <port number>");
            System.exit(1);
        }

        int portNumber = Integer.parseInt(args[0]);

        try (ServerSocket serverSocket = new ServerSocket(portNumber)) {
            System.out.println("Server started on port " + portNumber);

            // Loop to handle multiple client connections sequentially
            while (true) {
                // Accept a new client connection
                try (Socket clientSocket = serverSocket.accept();
                     PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
                     BufferedReader in = new BufferedReader(new
                                                   InputStreamReader(clientSocket.getInputStream()))) {

                    System.out.println("New client connected: " + clientSocket.getInetAddress());
```

Continues on next slide ...

Continues from previous slide ...

```
            // a client has connected

             String inputLine;
             // Read input from the client and echo it back
             while ((inputLine = in.readLine()) != null) {
                 System.out.println("Received: " + inputLine);
                 out.println(inputLine);
             }
             // Once the client is done, the connection is closed and the loop continues
             //The Java runtime automatically closes the readers and the socket
            // because they were created in the try-with-resources statement!
            // The Java runtime closes these resources in reverse order that they were created.

         } catch (IOException e) {
             System.out.println("Error handling client: " + e.getMessage());
         }
     }
   } catch (IOException e) {
       System.out.println("Error: " + e.getMessage());
   }
 }
}
```

```java
public class EchoClient {
    public static void main(String[] args) throws IOException {

        if (args.length != 2) {
            System.err.println(
                    "Usage: java EchoClient <host name> <port number>");
            System.exit(1);
        }

        String hostName = args[0];
        int portNumber = Integer.parseInt(args[1]);

        try (
            Socket echoSocket = new Socket(hostName, portNumber);
            PrintWriter out =
                new PrintWriter(echoSocket.getOutputStream(), true);
            BufferedReader in =
                new BufferedReader(
                    new InputStreamReader(echoSocket.getInputStream()));
            BufferedReader stdIn =
                new BufferedReader(
                    new InputStreamReader(System.in))
        ) {
```

Continues on next slide ...

Continues from previous slide ...

```
        ) {
            String userInput;
            while ((userInput = stdIn.readLine()) != null) {
                out.println(userInput);
                System.out.println("echo: " + in.readLine());
            }
          echoSocket.close();

        } catch (UnknownHostException e) {
            System.err.println("Don't know about host " + hostName);
            System.exit(1);
        } catch (IOException e) {
            System.err.println("Couldn't get I/O for the connection to " +
                    hostName);
            System.exit(1);
        }
    }
}
```

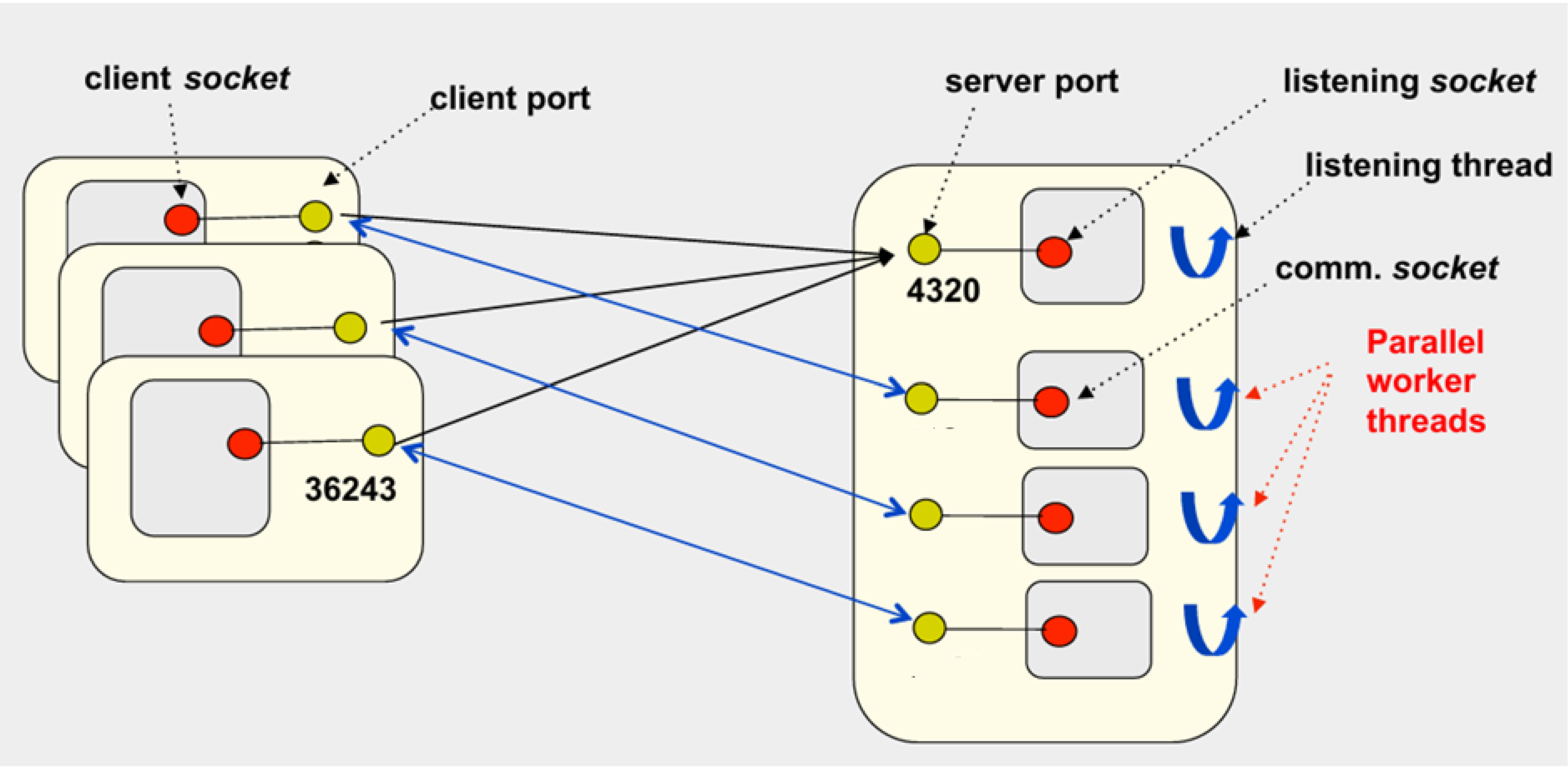
Source Code

- https://staff.cs.upt.ro/~ioana/apd/java/EchoServer.java
- https://staff.cs.upt.ro/~ioana/apd/java/EchoClient.java

Server design

- 2 types of servers:
  - Sequential server: a single thread processes incoming requests in sequence. Server cannot accept new requests until the current request is not finished.
  - Concurrent (parallel) server: uses multiple threads to process incoming requests. Each request is processed by a thread.

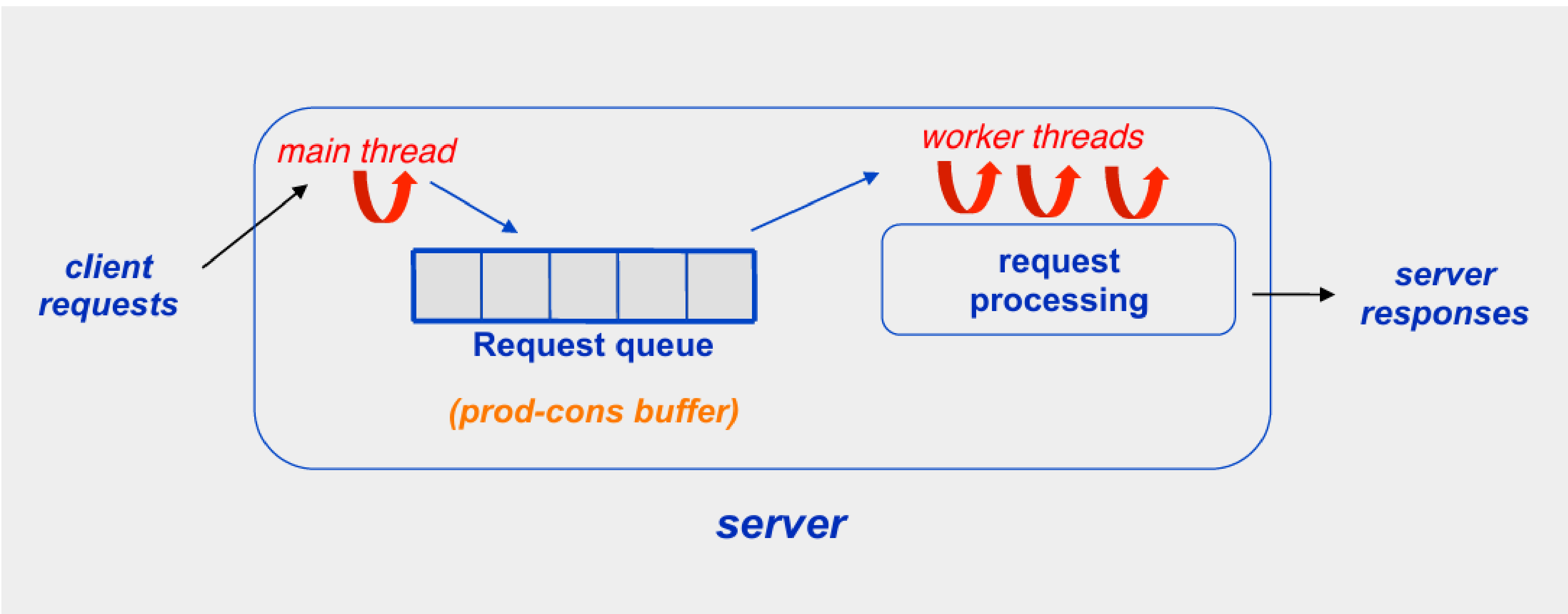
Multithreaded TCP Server



Basic design multithreaded server

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

Thread-Pool in 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 {
 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
  }
 }
```

```java
public class EchoConcurrentServer {
    public static void main(String[] args) throws IOException {

        if (args.length != 1) {
            System.err.println("Usage: java EchoConcurrentServer <port number>");
            System.exit(1);
        }

        int portNumber = Integer.parseInt(args[0]);

        // Create a thread pool with a fixed number of threads
        ExecutorService threadPool = Executors.newFixedThreadPool(10);

        try (ServerSocket serverSocket = new ServerSocket(portNumber)) {
            System.out.println("Server started on port " + portNumber);
```

Continues on next slide ...

Continues from previous slide ...

```
    // Loop to accept multiple client connections concurrently
    while (true) {
        // Accept a new client connection
        try {
            Socket clientSocket = serverSocket.accept();
            System.out.println("New client connected: " +
                                                        clientSocket.getInetAddress());

            // Submit a task to handle the client in the thread pool
            threadPool.submit(new ClientHandler(clientSocket));
        } catch (IOException e) {
            System.out.println("Error accepting client: " + e.getMessage());
        }
    }
} catch (IOException e) {
    System.out.println("Error: " + e.getMessage());
}
}
```

Continues from previous slide …

```
private static class ClientHandler implements Runnable {
    private final Socket clientSocket;

    public ClientHandler(Socket clientSocket) {
        this.clientSocket = clientSocket;
    }

    public void run() {
        try (PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
             BufferedReader in = new BufferedReader(new
                                InputStreamReader(clientSocket.getInputStream()))) {

            String inputLine;
            // Read input from the client and echo it back
            while ((inputLine = in.readLine()) != null) {
                System.out.println("Received: " + inputLine);
                out.println(inputLine);
            }
        } catch (IOException e) {
            System.out.println("Error handling client: " + e.getMessage());
        }
    }
}
```

Source Code

- https://staff.cs.upt.ro/~ioana/apd/java/EchoConcurrentServer.java

Case study: Implementation of a MPI middleware

MPI architecture