Distributed Systems

COMP 212

Lecture 12

Othon Michail

Communication

Interprocess Communication

- The “heart” of every distributed system.
- Question: how do processes on different machines exchange information?
- Answer: with difficulty … ☹
- Message passing harder than shared memory
  - But shared memory in general not available in DSs
- Established computer network facilities are too primitive, resulting in DSs that are too difficult to develop
  - Facilities not expected to change dramatically soon

Client-Server Model

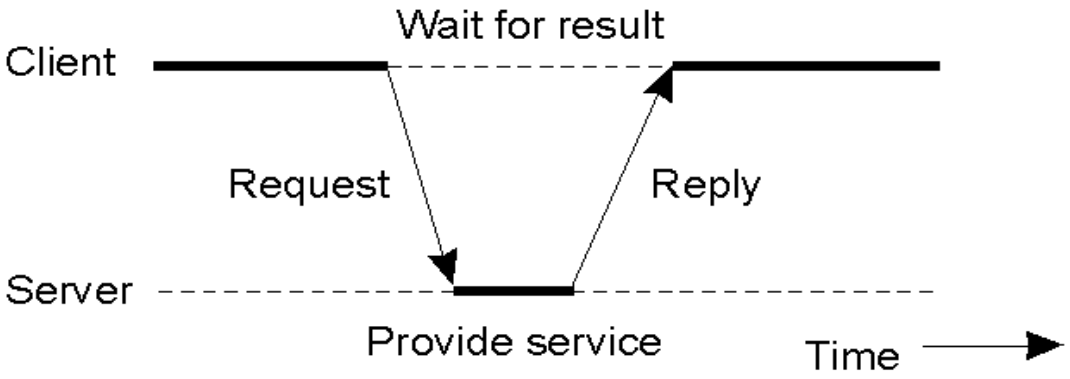
Participants are divided into

- Servers: implementing a specific service
- Clients: requesting a service from a server by sending it a request and subsequent waiting for the server's reply

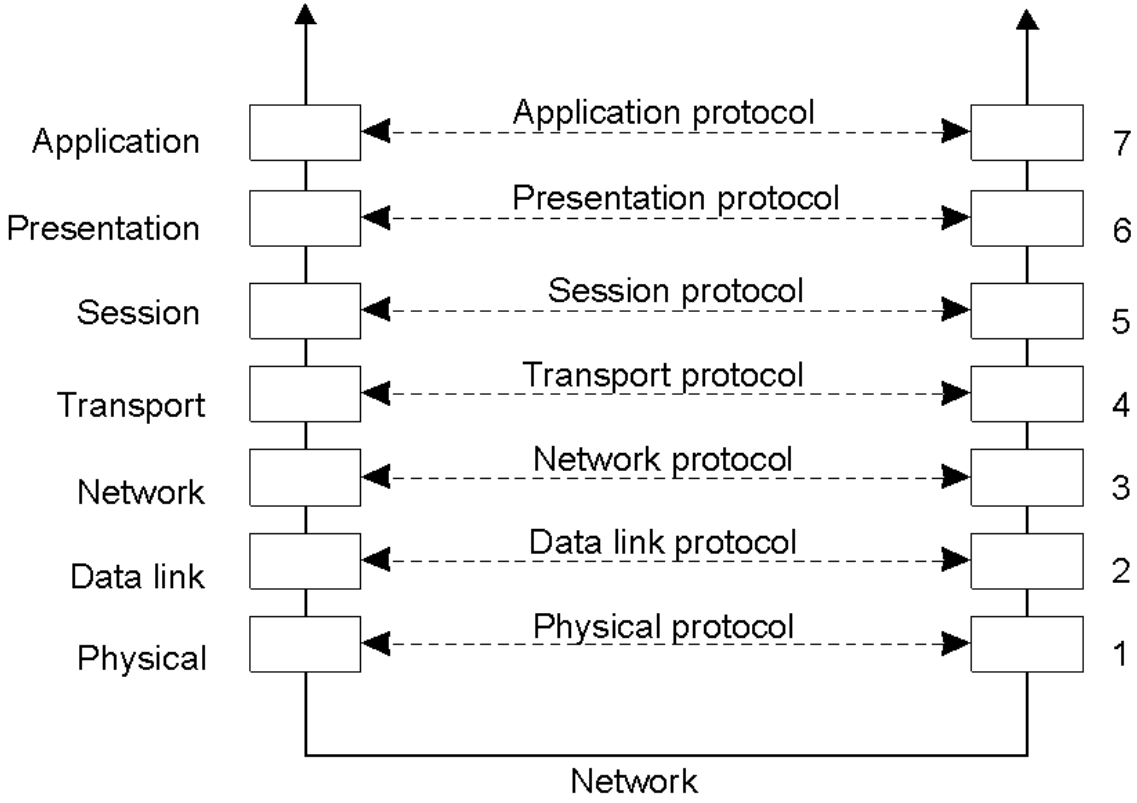
Distributed across different machines

  - possibly overlapping

Follow a request-reply behavior



Layered Protocols



Layers, interfaces, and protocols in the OSI model

Physical Layer

- Concerned with transmitting 0s and 1s
- Specifies the mechanical, electrical and optical behaviour of the plugs and (hardware) sockets of a physical interface to the network
- Implementations of the physical layer perform functions in order to signal, modulate, and synchronize the transmission of bits over the physical network
- It may perform error detection by monitoring the quality of the electrical or optical signals received

Data Link Layer

- Basic task: error correction
  - Groups bits into frames
  - Puts a special bit pattern at the start and end of each frame
  - Also appends a checksum
  - Receiver can check correctness by recomputing the checksum and comparing to the checksum received
- Responsible for transmission of frames between nodes that are directly connected by a physical link
  - Nodes may be routers, switches, or computers
- It maps the physical connection provided by the physical layer into a layer that can transmit frames from one node to another in a relatively error-free mode
- Would be sufficient for a LAN but not for a WAN
  - Does not provide a mechanism for locating the receiver

Network Layer

- Provides a routing mechanism
- Routing: How to choose the best path from one machine to another
  - The shortest route is not always the best
  - What really matters is the delay (as with road traffic)
- Data link layer: node-to-node connection
- Network layer: routing messages along multiple nodes in the network
  - It implements connections between arbitrary nodes of the network
- IP (Internet Protocol)
  - Part of the Internet protocol suite
  - Each IP packet (i.e., message) is routed to its destination independent of all others
  - No internal path is selected and remembered

Transport Layer

- Provides an interface between the underlying network and the application developer
- "Turns" the network into something that developers can use
- The Application layer expects from the Transport layer to "eventually" deliver a message without loss
  - Provides a reliable connection service

1. Breaks message into pieces, small enough for transmission
2. Assigns a number to each and
3. Sends them all
4. Packets may not arrive at the correct order
   - Puts everything back in order before delivering to the application

- TCP (Transmission Control Protocol)
  - The transport protocol of the Internet
  - TCP/IP is the de facto standard for network communication

Application Layer

- Contains all applications that do not qualify as members of the other layers
  - Mainly application-specific protocols
  - i.e., designed to meet the communication requirements of specific applications
  - Often defining the interface to a service
- Examples:
  - HTTP (HyperText Transfer Protocol)
    - Handles the transfer of Web pages
    - Implemented by Web browsers and Web servers
  - FTP (File Transfer Protocol)
- The Session and Presentation layers are obsolete
  - Have been replaced by a Middleware layer
    - contains application-independent protocols (general purpose)

Connection (-less) Transport Layer

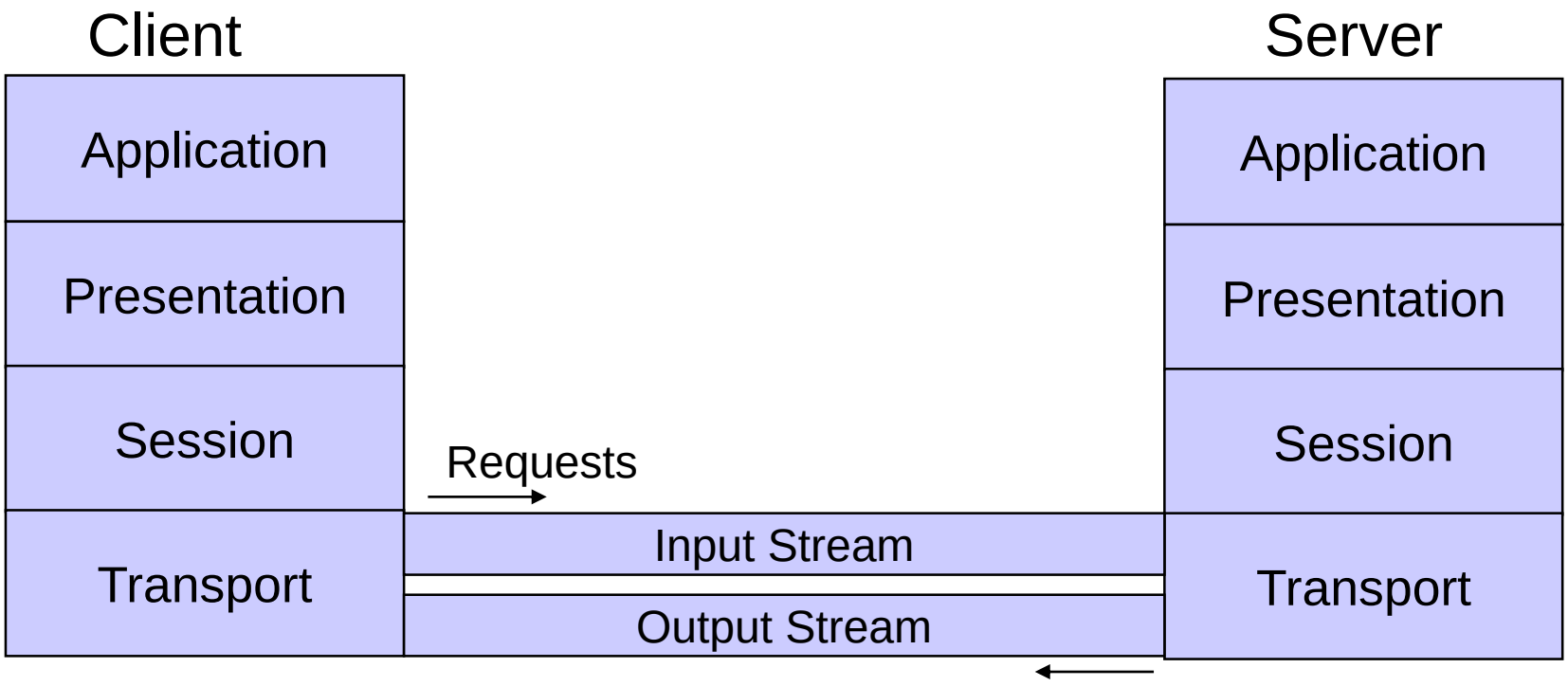
- Connection-oriented transmission protocols
  - Provide operations to:
    - Open a connection
    - Close a connection
    - Write data into such a connection
    - Read data from such a connection
- Connection-less transmission protocols
  - Provide the ability of sending fixed length messages between distributed hosts
  - These messages are referred to as datagrams

Internet Transport Protocols

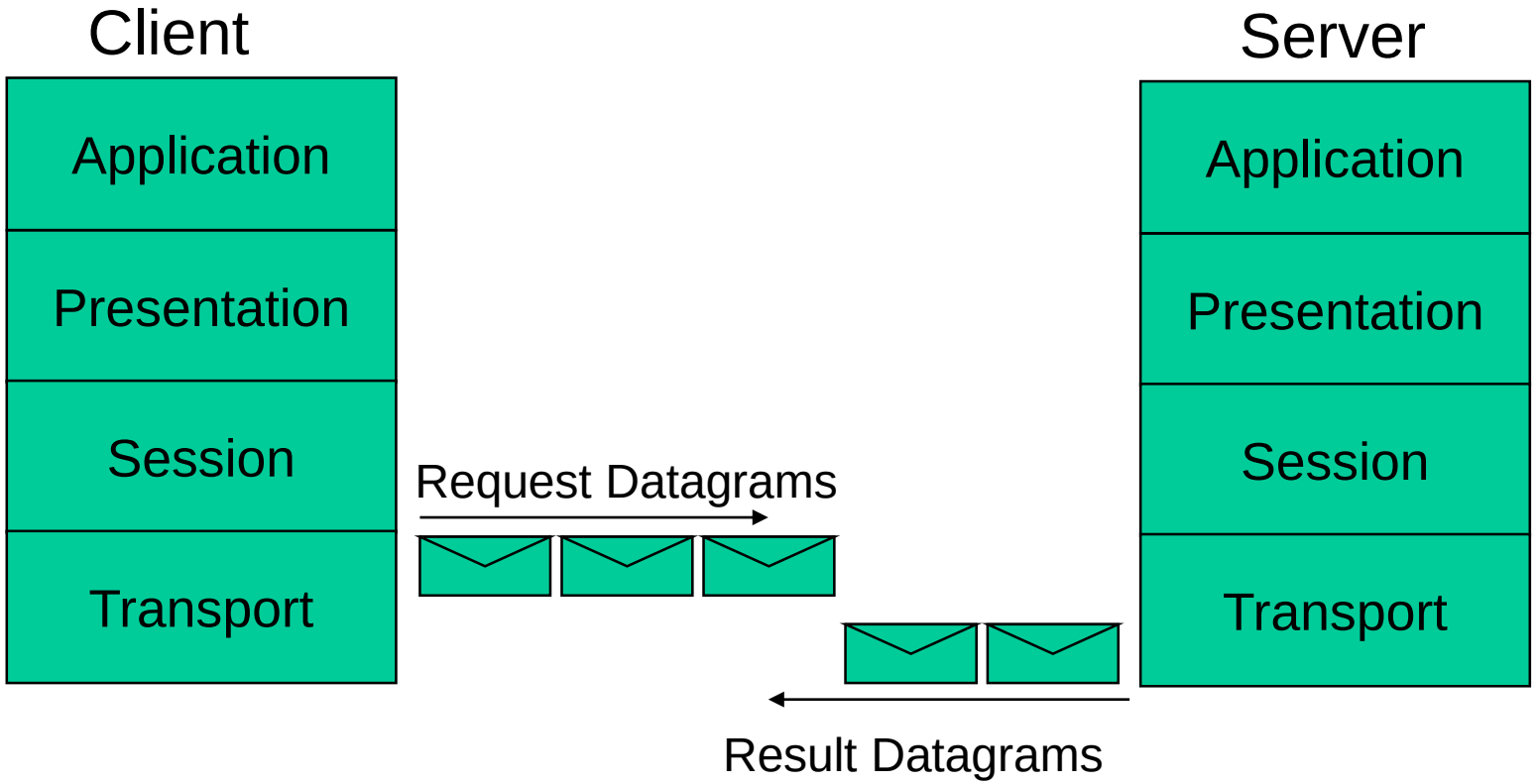
- TCP (Transmission Control Protocol)
  - Connection-oriented
  - Reliable but slow (introduces overhead to guarantee delivery)
  - Applications: HTTP, POP, IMAP
- UDP (User Datagram Protocol)
  - Connection-less
  - Very fast but unreliable (does not guarantee delivery)
  - Applications: Domain Name System (DNS), Voice and Video streaming, VoIP
- Choice of protocol depends on whether
  - reliability or
  - performance

  is the major concern for a given application

TCP - Graphical Representation



UDP - Graphical Representation



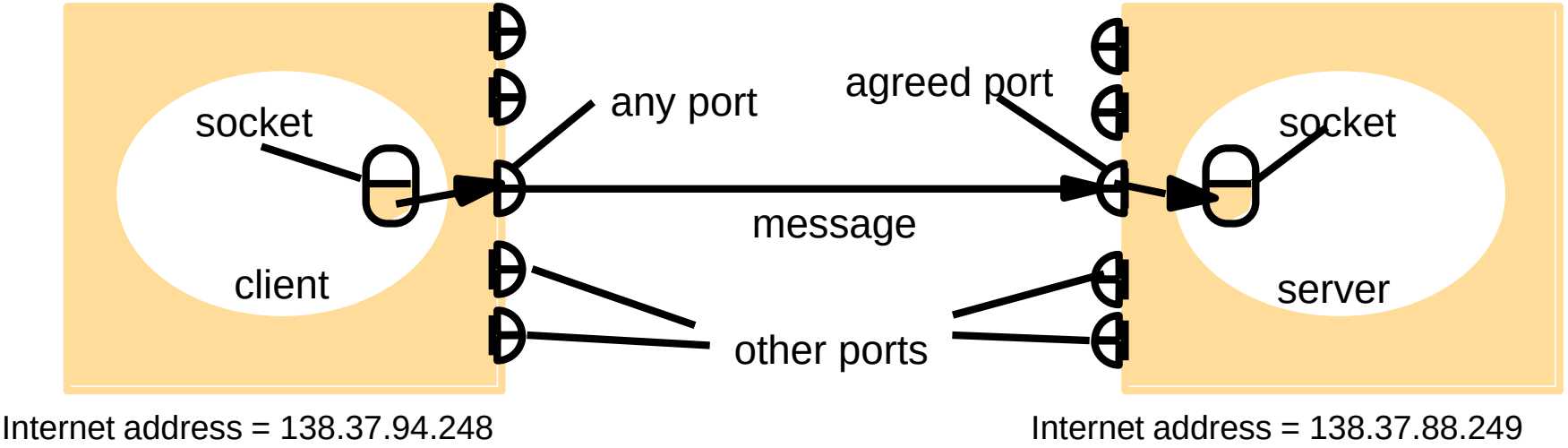
- We do not study UDP in detail here

Communication at Transport Level

- Provided by all networks
- Practically, lowest-level protocol available to application developer

Sockets and Ports

- Reliable point-to-point communication provided by TCP
  - Processes bind sockets to a connection and read/write to them



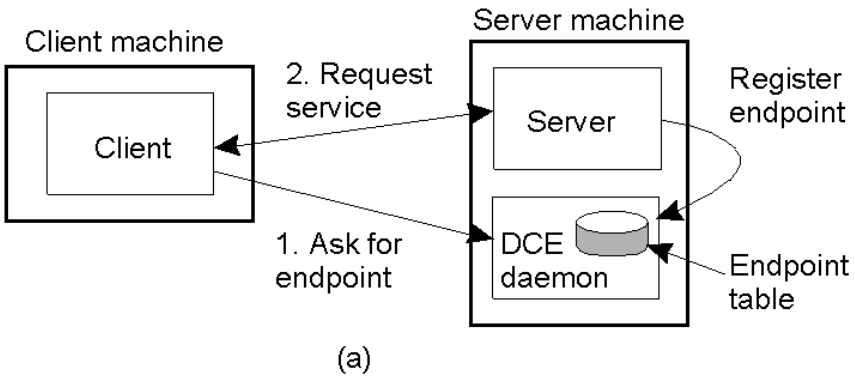
- A socket is one endpoint of a two-way communication link between two programs running on the network
- Is bound to a port number (logical construct) so that the TCP layer can identify the application that data is destined to be sent

Problem: Identifying "end-points"?

- How do clients know which end-point (or port) to contact a server at? How do they "bind" to a server?

1. Statically assigned end-points
   - e.g., TCP port 21 for telnet, TCP port 80 for HTTP, …
2. Dynamically assigned end-points
   - Special daemon on each machine that runs servers
   - A client first contacts the daemon
3. "Super-server"
   - Listens to several ports, i.e., provides several services
   - Starts another process/thread, e.g., inetd on UNIX

Servers: Binding to End-Points



a) Client-to-server binding using a daemon (DCE)

b) Client-to-server binding using a super-server (inetd on UNIX)

Sockets in Java

Sockets in Java

- Facilities for simple network programming in Java can be found in `Java.net` package

- Main classes:
  - Socket
  - ServerSocket

The Socket Class

- Provides a number of constructors which enable the programmer to create a socket to a remote computer
  - Usually for programming the client side of the connection
- The simplest constructor has two arguments:
  - Hostname (a String: name of the computer using the DNS convention)
  - port number (an integer from 0 to $2^{16}-1 = 65535$)
- Example:

```
Socket mySock = new Socket("www.csc.liv.ac.uk", 80);
```

Reading from a Socket (1)

- The method getInputStream will obtain an InputStream object attached to the socket which can then be used to read data.
- Example:

```
String hostName = args[0]; // e.g. "www.csc.liv.ac.uk"
int portNumber = Integer.parseInt(args[1]); // e.g. 8000

Socket myClientSocket = new Socket(hostName, portNumber);
InputStream myInputStream = myClientSocket.getInputStream();
```

Reading from a Socket (2)

- Also we can attach a buffer to the InputStream object which can contain data until enough can be sent in an efficient way
- Example:

```
Socket myClientSocket = new Socket(hostName, portNumber);
InputStream myInputStream = myClientSocket.getInputStream();
BufferedReader input = new BufferedReader(new
   InputStreamReader(myClientSocket));
String lineRead = input.readLine();
```

Writing to a Socket

- The method getOutputStream can be used to write data to the socket.
- Example:

```
Socket myClientSocket = new Socket("localhost", 2048);
OutputStream myOutputStream =
    myClientSocket.getOutputStream();
PrintWriter output = new PrintWriter(myOutputStream, true);
```

Connection is Bi-directional

```
Socket aSocket = new Socket("www.csc.liv.ac.uk", 80)

//Set up the streams associated with the socket
    BufferedReader input = new BufferedReader(
            new InputStreamReader(aSocket.getInputStream()));
    PrintWriter output = new PrintWriter(
           aSocket.getOutputStream(), true);

//Send message to server
    output.println("Hello");

//Get reply from server
    String serverReply = input.readLine();
    if (serverReply.equals("Hello")) {
        ...  //Process the reply
    } else {
        ...  //Carry out some error process
    }
```

The Server Socket

- Provides a number of constructors which enable the programmer to create a socket on a server
- The simplest is:

```
ServerSoket(int port)
```

- This creates a socket on the given port (this machine)

Accepting Connections

- The most important method: accept
- This method suspends the server until a client attempts to connect to it
- The method “accept” returns the Socket object which can be used to establish a connection with the client that has connected in

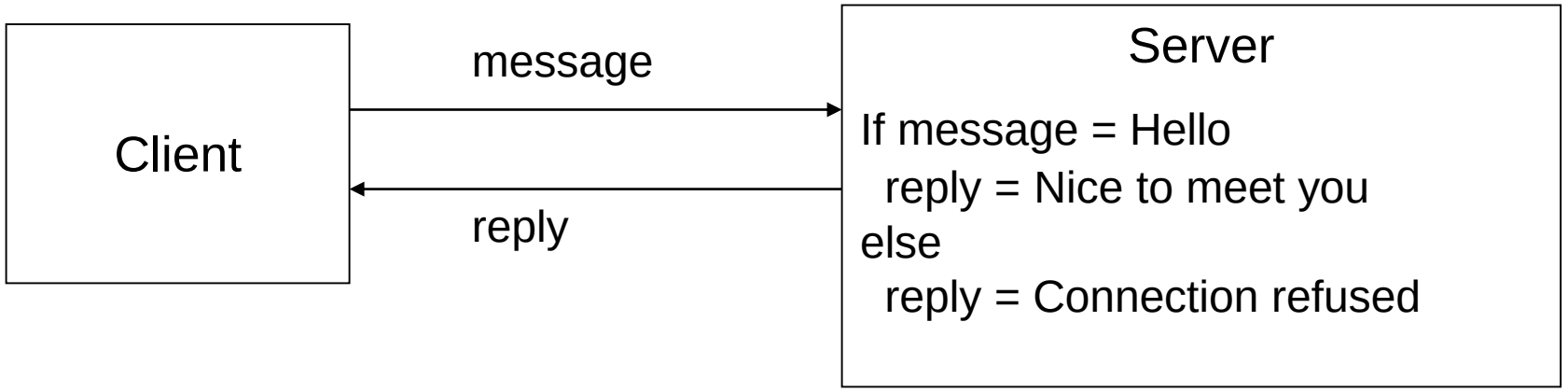
Server Socket I/O

```
ServerSocket serverSocket = new ServerSocket(2500);

// Wait for a connection
   Socket clientSocket = serverSocket.accept();
// Set up the streams associated with the socket
   BufferedReader in = new BufferedReader(new
         InputStreamReader(clientSocket.getInputStream()));
   PrintWriter out = new
         PrintWriter(clientSocket.getOutputStream(),true);

// Read string from client
   String inputLine = in.readLine();
   if (inputLine.equals("Hello")) {
         out.println("Connection established");
   }
```

A Trivial Communication Protocol



- The server waits for a message from a client.
- If the string is “Hello”, then “Nice to meet you” is sent to the client, otherwise “Connection refused” is sent to the client.

A Complete Client (1)

```
import java.net.*;
import java.io.*;

public class HelloClient {
    public static void main(String[] args) throws IOException {

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

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

        try (
            Socket myClientSocket = new Socket(hostName, portNumber);
            PrintWriter output = new
    PrintWriter(myClientSocket.getOutputStream(),true);
            BufferedReader input = new BufferedReader(new
    InputStreamReader(myClientSocket.getInputStream()));
            BufferedReader stdIn = new BufferedReader(new
    InputStreamReader(System.in));
        ) {
            String userInput;
            System.out.println(input.readLine()); // reads the first message from the
server and prints it
```

A Complete Client (2)

```
            System.out.println("Say something to the server (and then press enter): ");
            userInput = stdIn.readLine(); // reads user's input
            output.println(userInput); // user's input transmitted to server
            System.out.println(input.readLine()); // reads server's ack and prints it
            System.out.println("-----------End of communication-----------");
            System.out.println("\nCommunication with server " + hostName + " was
successful! Now closing...");

        } 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);
            e.printStackTrace();
            System.exit(1);
        }
    }
}
```

Its Corresponding Server (1)

```
import java.net.*;
import java.io.*;

public class HelloServer {
    public static void main(String[] args) throws IOException {

        int portNumber;

        if (args.length < 1) {
            System.out.println("Warning: You have provided no arguments\nTrying to connect
to the default port 8000...");
            portNumber = 8000;
        } else if (args.length == 1) {
            portNumber = Integer.parseInt(args[0]);
        } else {
            System.out.println("Warning: You have provided > 1 arguments\nTrying with the
first argument to connect to a port...");
            portNumber = Integer.parseInt(args[0]);
        }

        while(true){ //in order to serve multiple clients but sequentially, one after the other
            try (
                ServerSocket myServerSocket = new ServerSocket(portNumber);
                Socket aClientSocket = myServerSocket.accept();
                PrintWriter output = new PrintWriter(aClientSocket.getOutputStream(),true);
                BufferedReader input = new BufferedReader(new
```

Its Corresponding Server (2)

```
    InputStreamReader(aClientSocket.getInputStream()));
            ) {

                System.out.println("Connection established with a new client with IP address:
" + aClientSocket.getInetAddress() + "\n");
                output.println("Server says: Hello Client " + aClientSocket.getInetAddress() +
". This is server " + myServerSocket.getInetAddress() +
                " speaking. Our connection has been successfully established!");

                String inputLine = input.readLine();
                System.out.println("Received a new message from client " +
aClientSocket.getInetAddress());
                System.out.println("Client says: " + inputLine);
                output.println("Server says: Your message has been successfully received!
Bye...");
                System.out.println("Connection with client " + aClientSocket.getInetAddress()
+ " is now closing...\n");

            } catch (IOException e) {
                System.out.println("Exception caught when trying to listen on port "
                    + portNumber + " or listening for a connection");
                System.out.println(e.getMessage());
            }
        }
    }
}
```

Summary

- Communication is indispensable for distributed systems
- Layered approach
- Socket-based (transport layer) communication is available in all modern computer systems
  - Hard to develop large-scale applications at this level of abstraction
  - Higher-level development methods and APIs exist