Institute of Computer Science
Department of Mathematics and Computer Science

Freie Universität Berlin

Algorithms and Programming IV

Communication Paradigms in Distributed Systems

Summer Term 2021 | 11.06.2021

Barry Linnert

Our topics today

Architectural Pattern

Layered Architecture

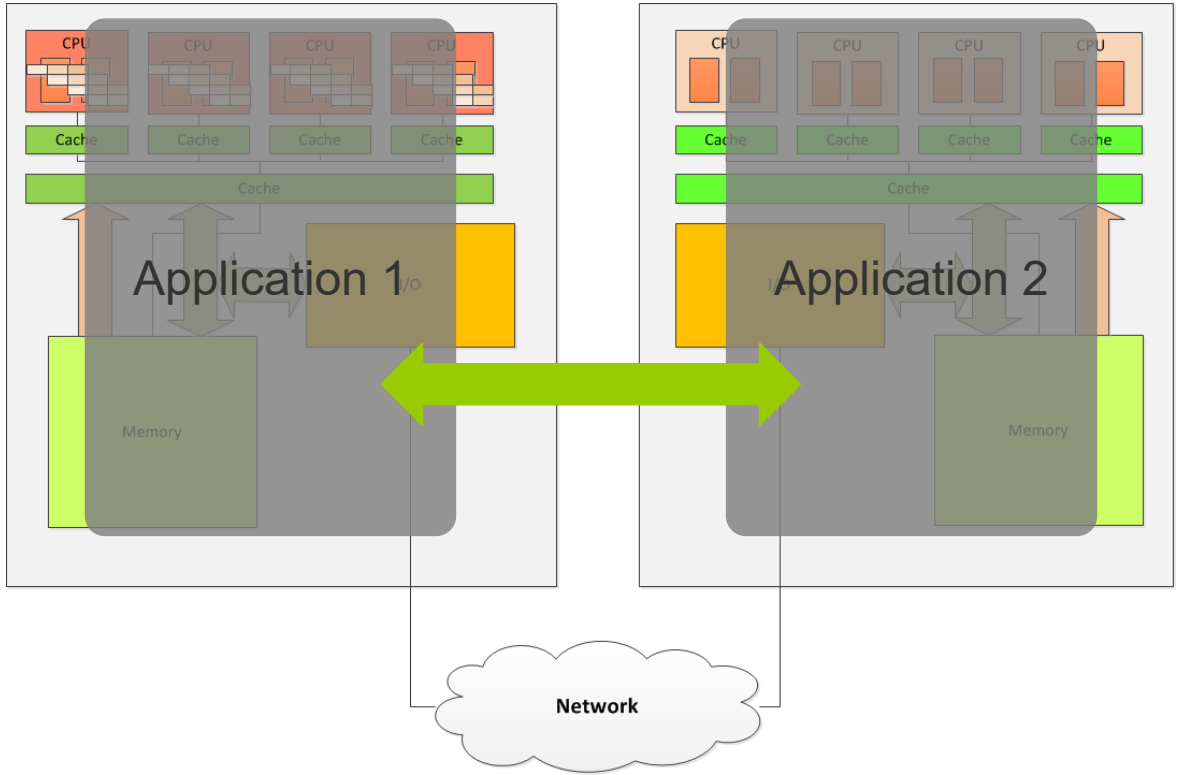
Tiered Architecture

Communication Paradigms

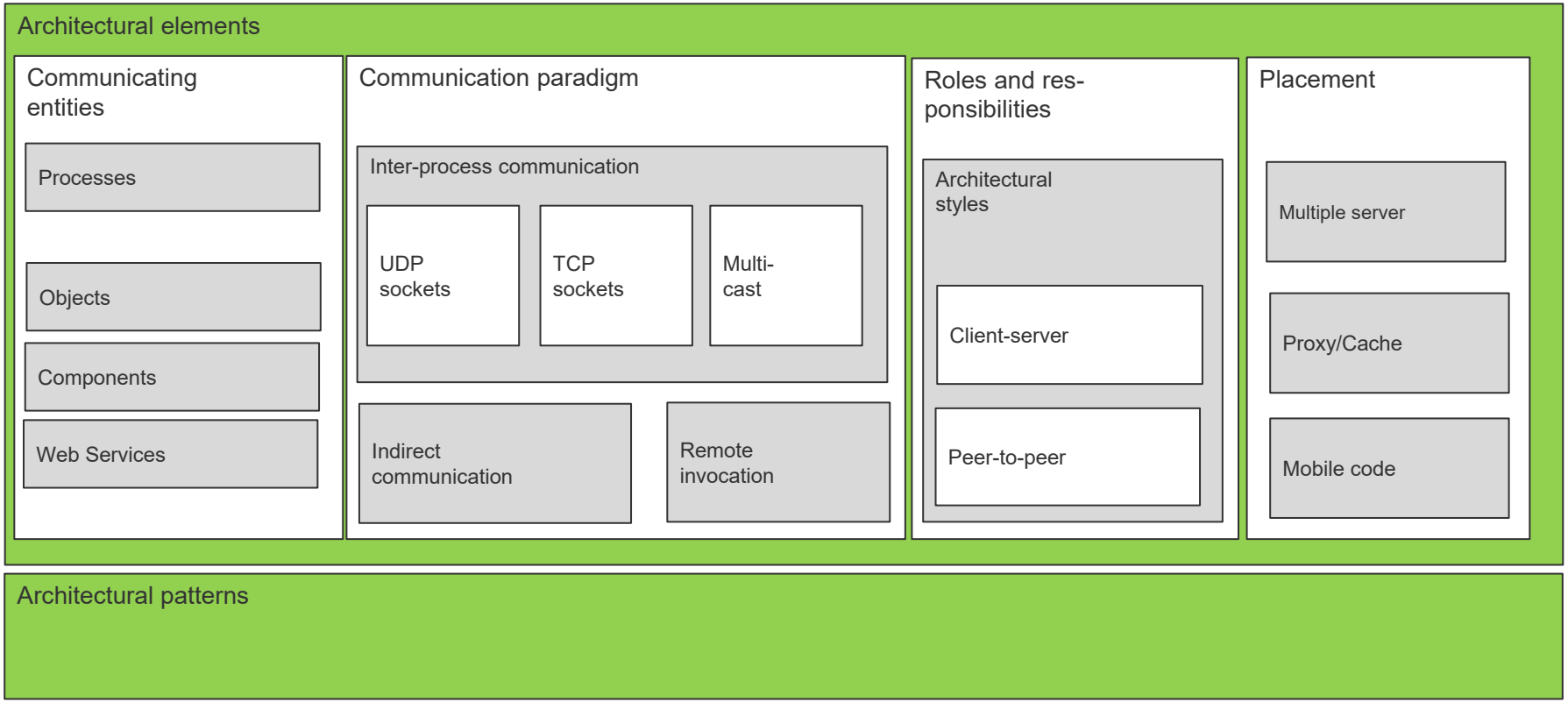
Interprocess Communication

- API for Internet Protocols
- UDP Datagram Communication
- TCP Stream Communication

Recap: Distributed System Model



Recap: Architectural Model

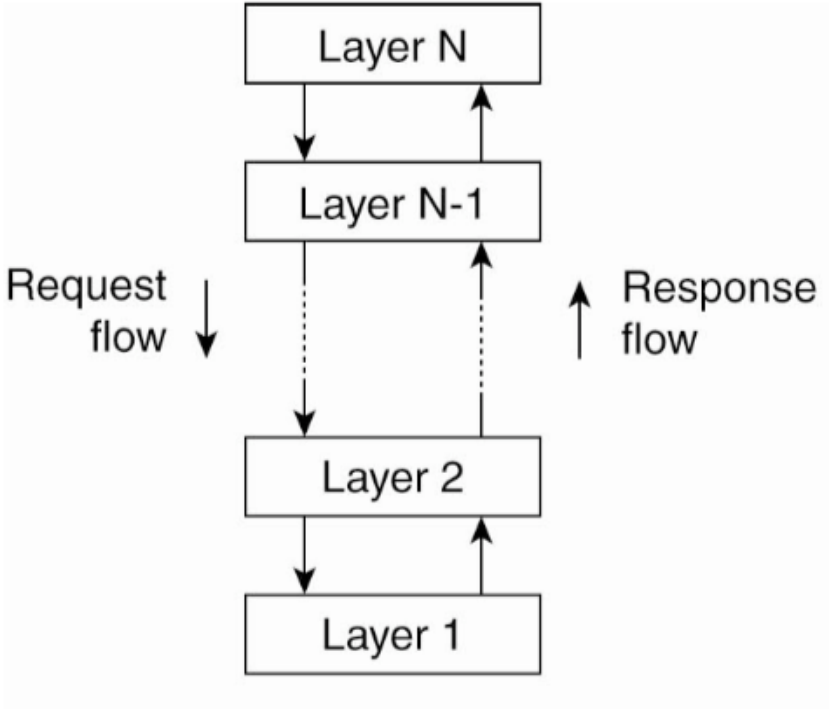


Freie Universität Berlin

Architectural models

ARCHITECTURAL PATTERNS

Layered Architecture



Applications, Services

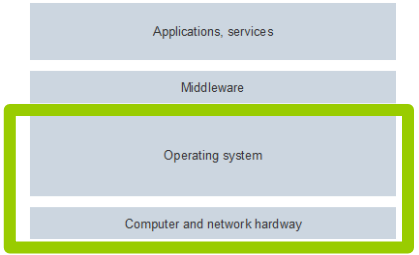
Middleware

Operating system

Computer and network hardware

Tanenbaum & van Steen. Distributed Systems. Principles and Paradigms. 2007.

Layered Architecture: Platform



Lowest level hardware and software layers for distributed systems and applications.

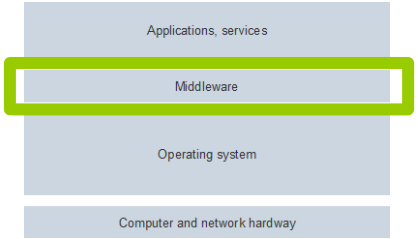
Characteristics

- Provide services to the layers above them.
- Implemented independently in each computer.
- Bringing the system's programming interface up to a level that facilitates communication and coordination between processes.

Examples

- x86/Windows, Intel x86/Solaris, PowerPC/Mac OS X, Intel x86/Linux

Layered Architecture: Middleware



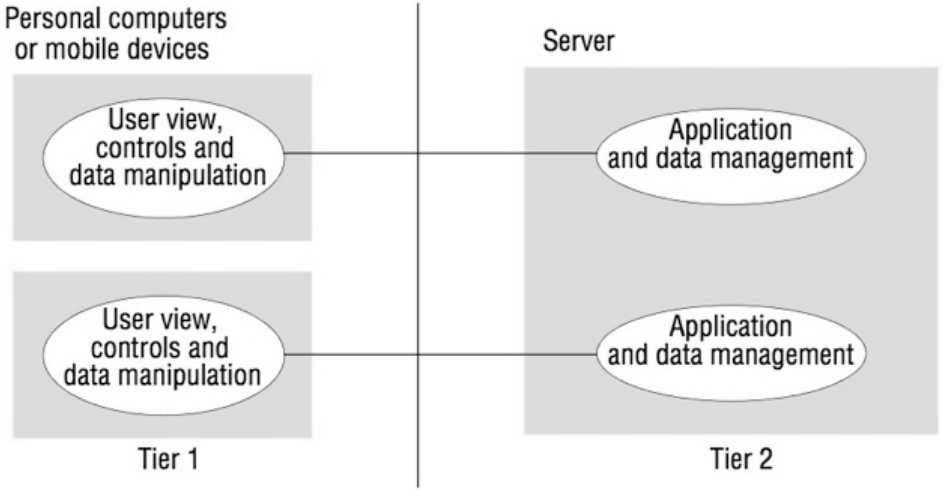
It is a layer of software whose purpose is to mask heterogeneity and to provide a convenient programming model to application programmers.

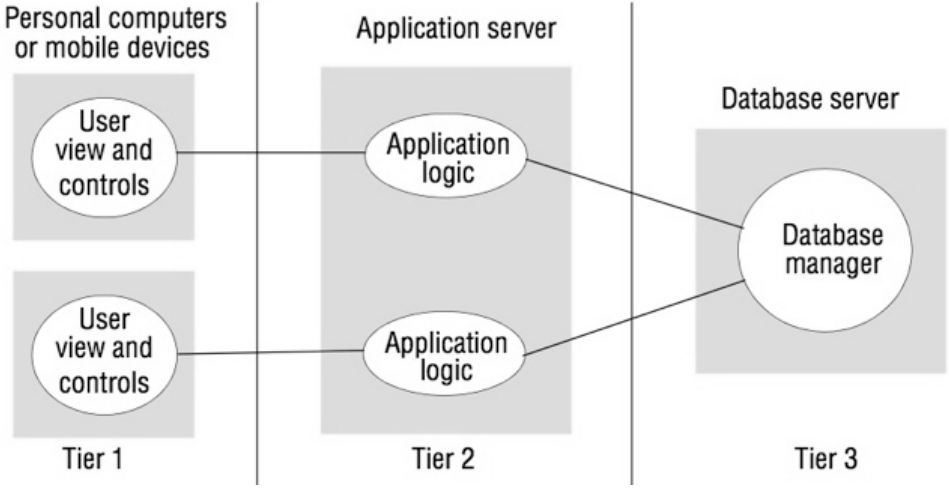
It is concerned with providing useful building blocks for the construction of software components that can work with one another.

Many distributed applications rely entirely on services provided by middleware to support their needs for communication and data sharing.

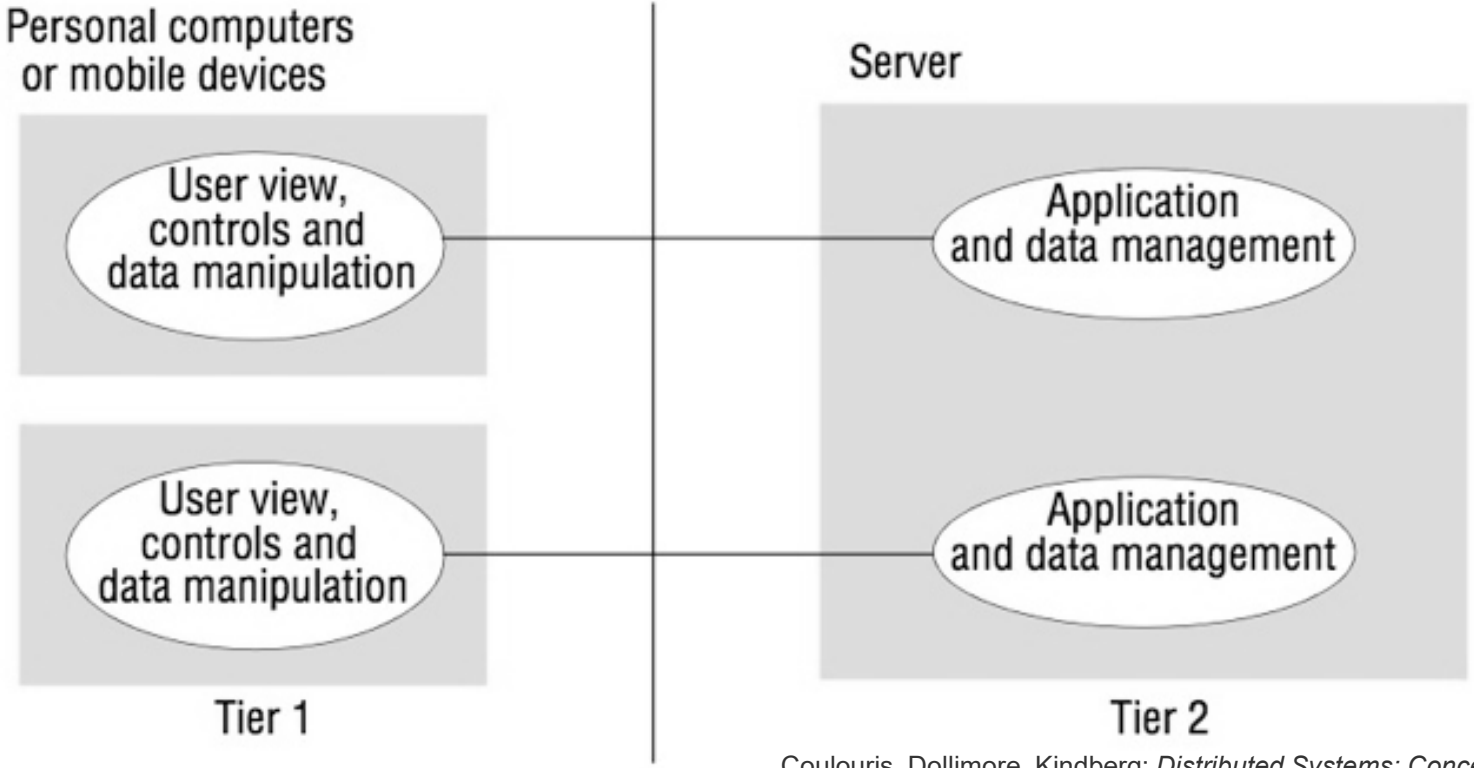
Tiered Architecture

- Tiered architectures are complementary to layering.
- Layering (logical decoupling) deals with the vertical organization of services into layers of abstraction.
- Tiering (physical decoupling) is a technique to organize functionality of a given layer and place this functionality into appropriate servers and, as a secondary consideration, on to physical nodes.



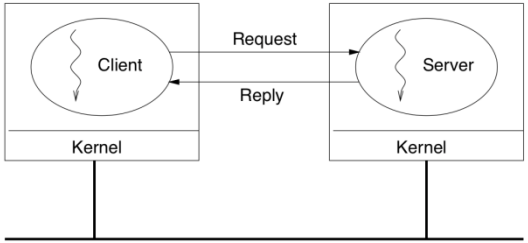


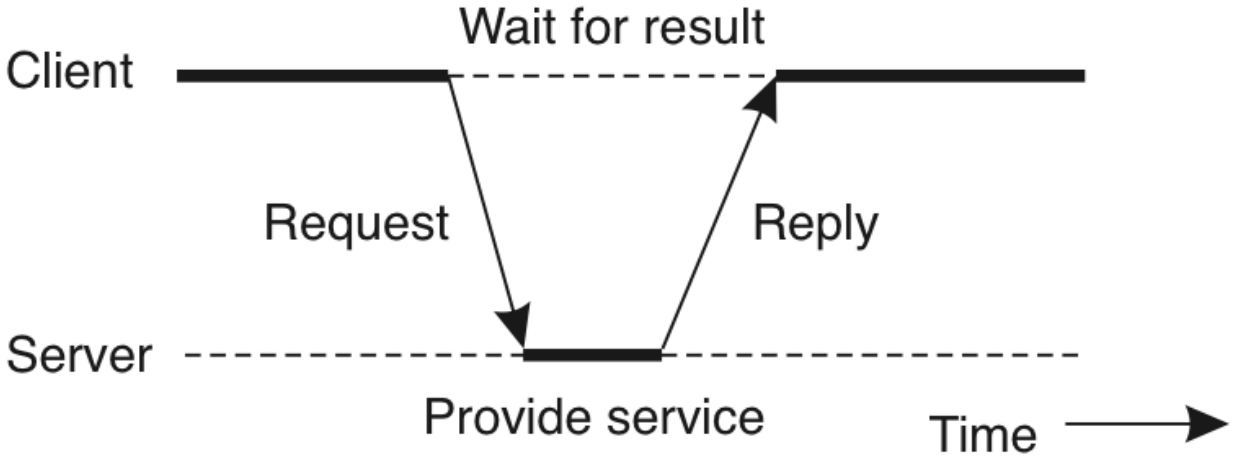
Two-tier Architecture



Coulouris, Dollimore, Kindberg: *Distributed Systems: Concepts and Design*. 2011.

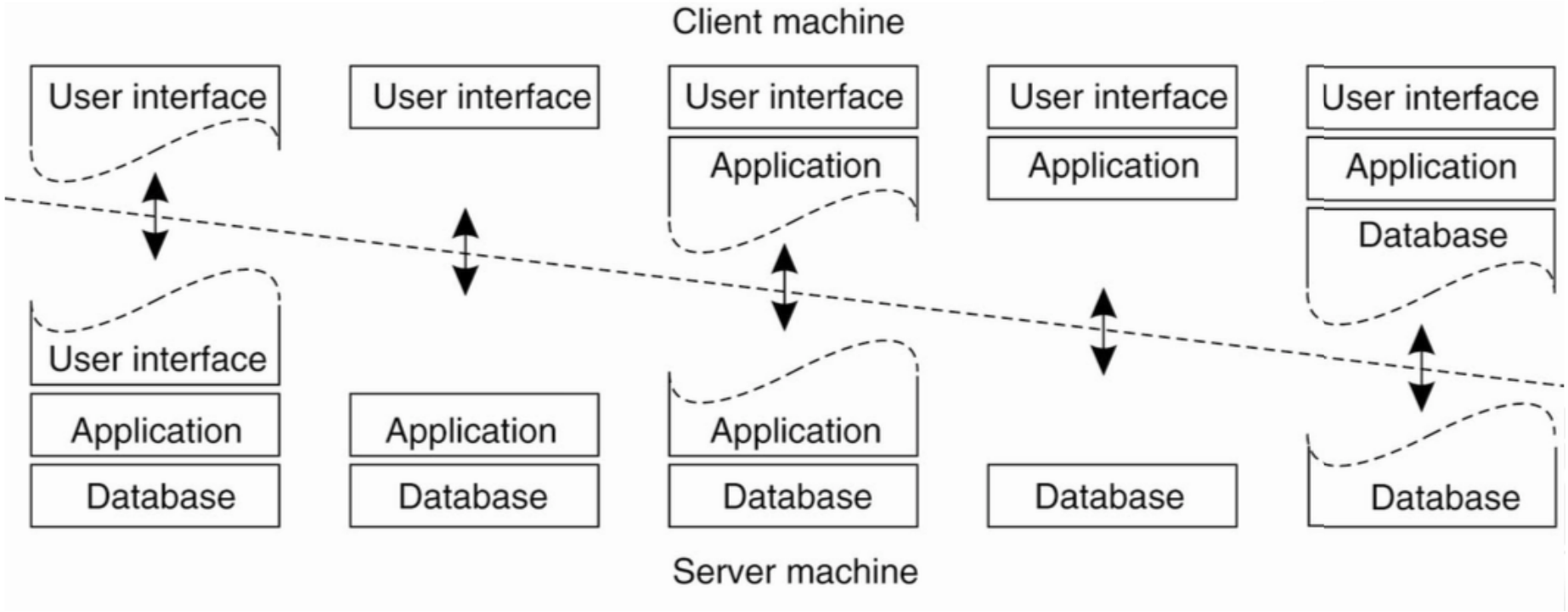
Client-Server





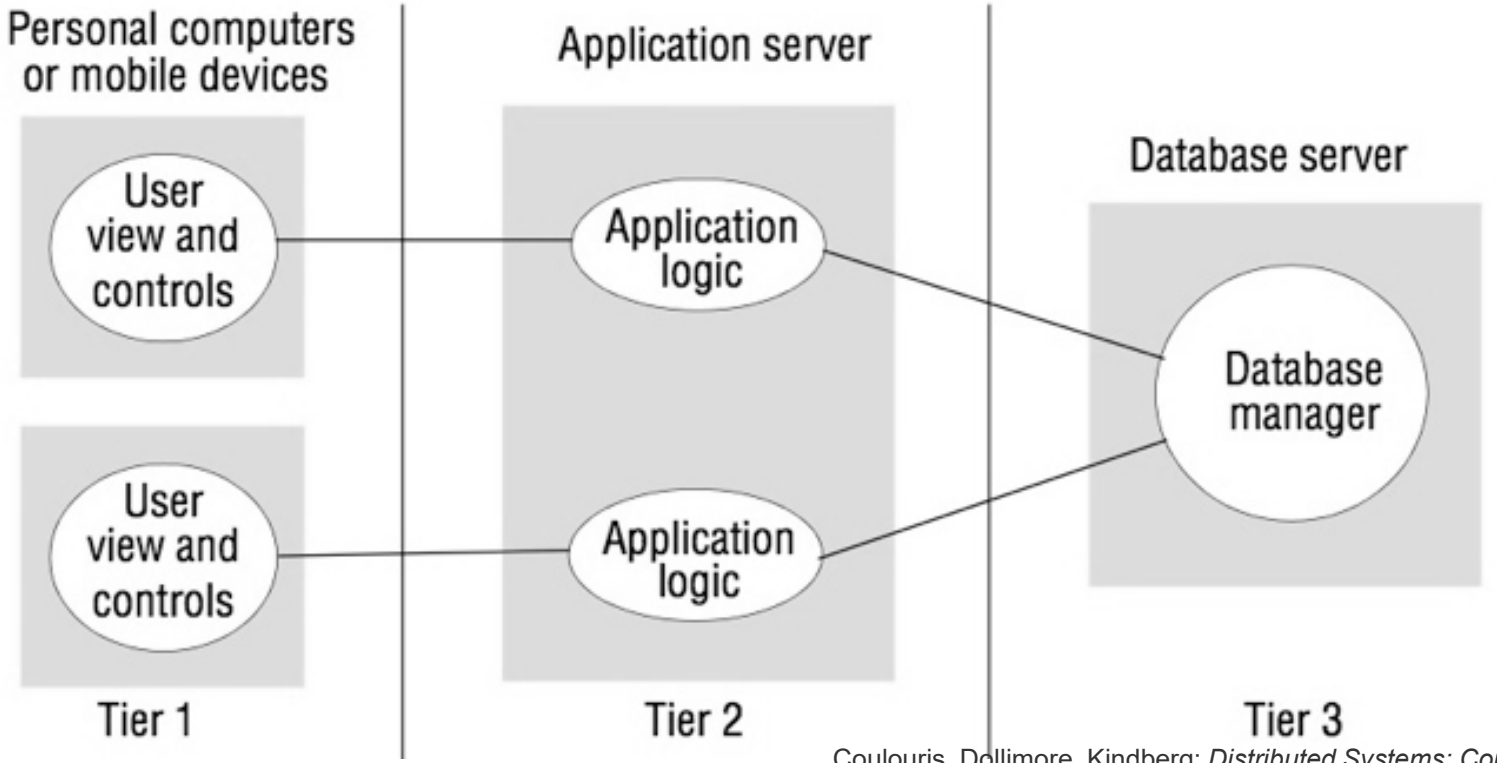
Tanenbaum & van Steen. Distributed Systems. Principles and Paradigms. 2007.

Alternative Client-Server Organisations



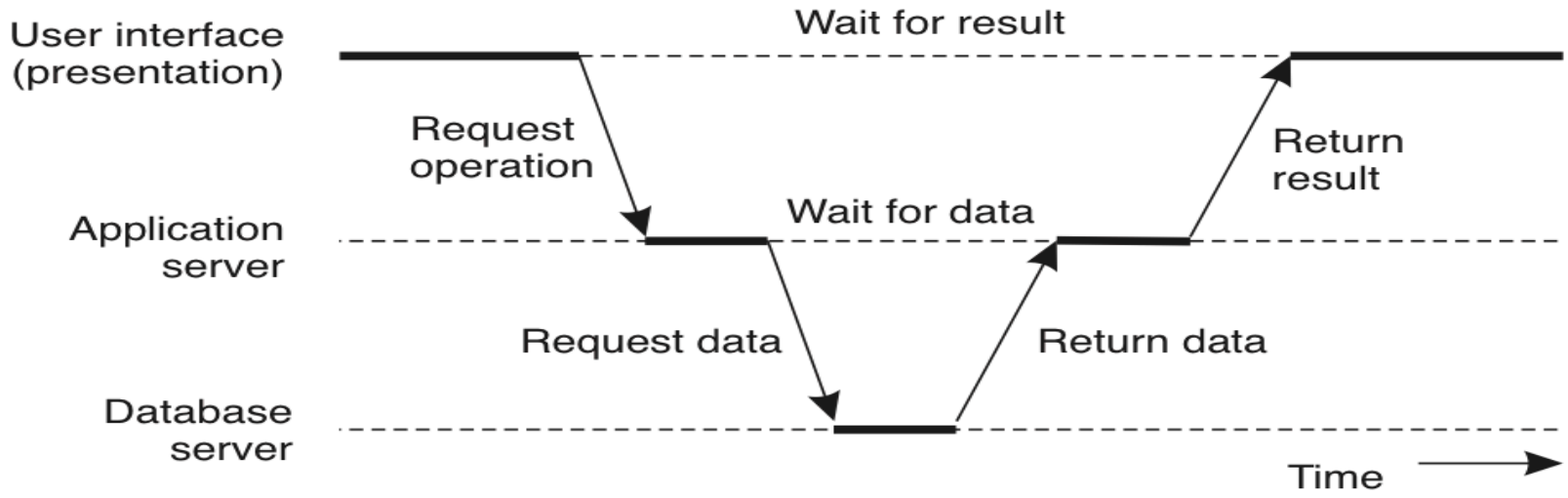
Tanenbaum & van Steen. Distributed Systems. Principles and Paradigms. 2007.

Three-tier Architecture



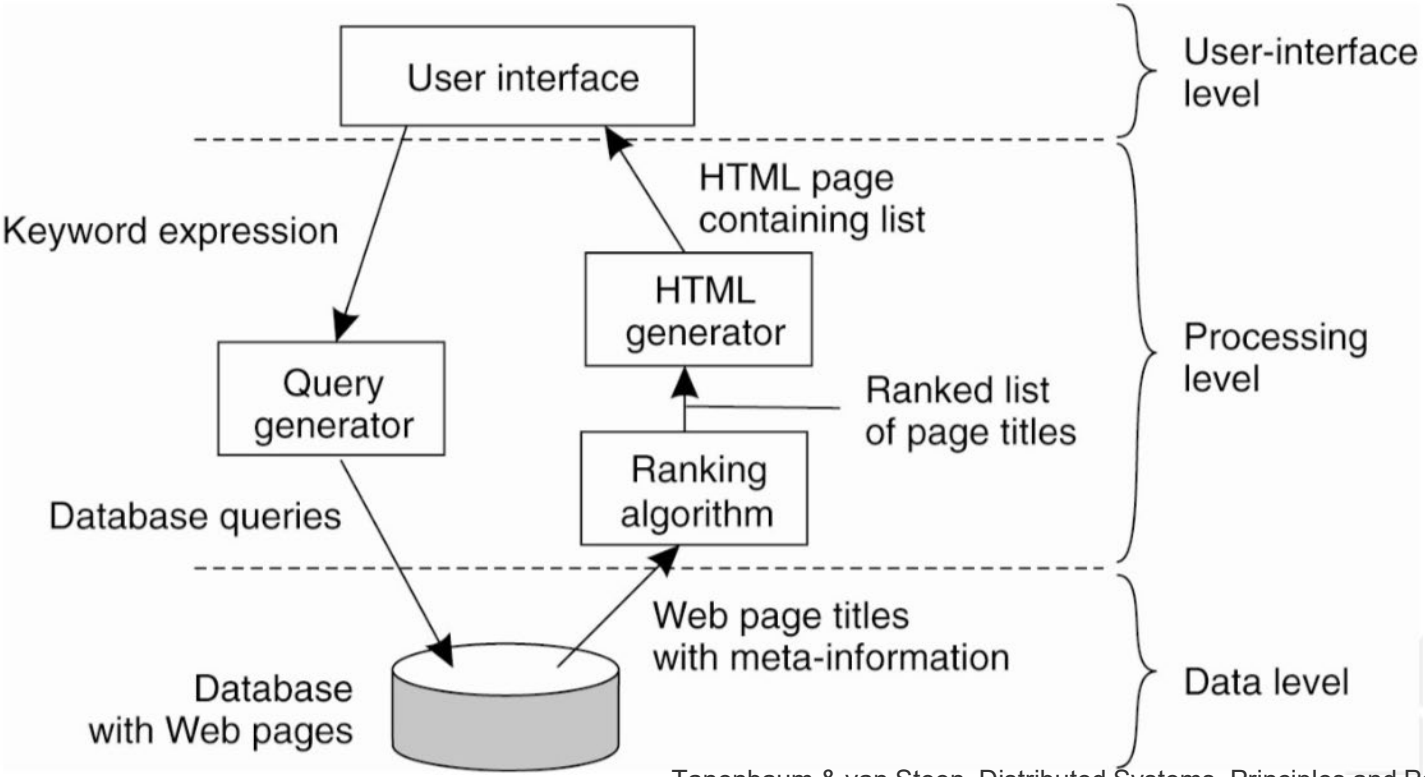
Coulouris, Dollimore, Kindberg: *Distributed Systems: Concepts and Design*. 2011.

Communication in a Multi-Tier System



Coulouris, Dollimore, Kindberg: *Distributed Systems: Concepts and Design*. 2011.

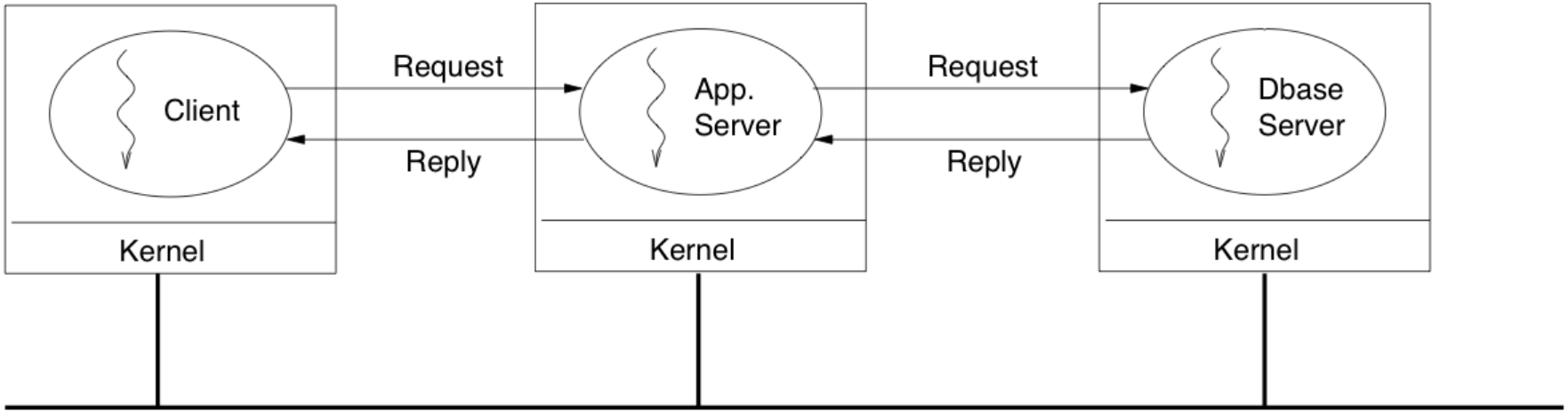
Example: Internet Search Engine



Tanenbaum & van Steen. Distributed Systems. Principles and Paradigms. 2007.

Multi-Tier: Vertical Distribution

Distribution of server functionality over multiple servers.



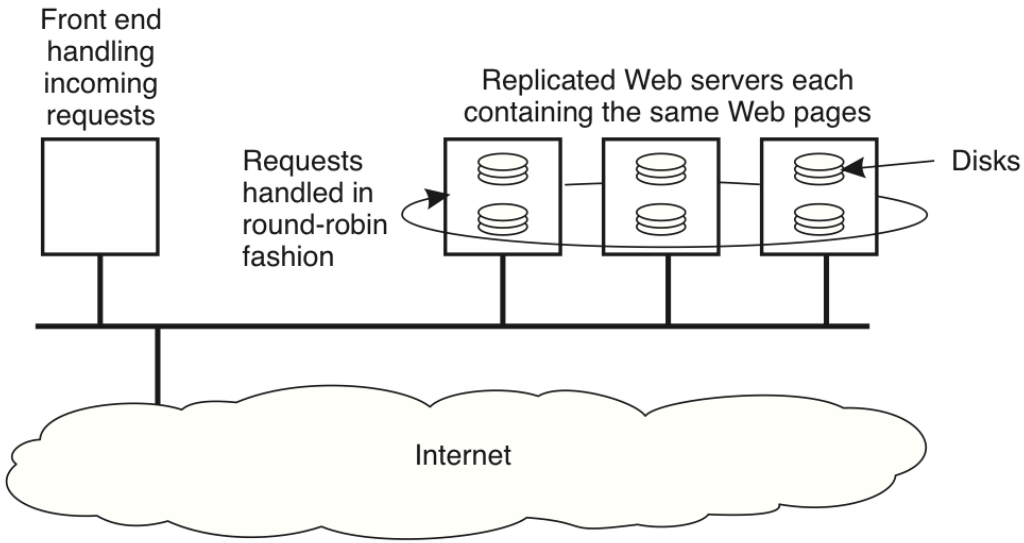
Tanenbaum & van Steen. Distributed Systems. Principles and Paradigms. 2007.

Horizontal Distribution

Involves replicating a server's functionality over multiple computers to improve scalability (by reducing the load on individual servers) and reliability (by providing redundancy).

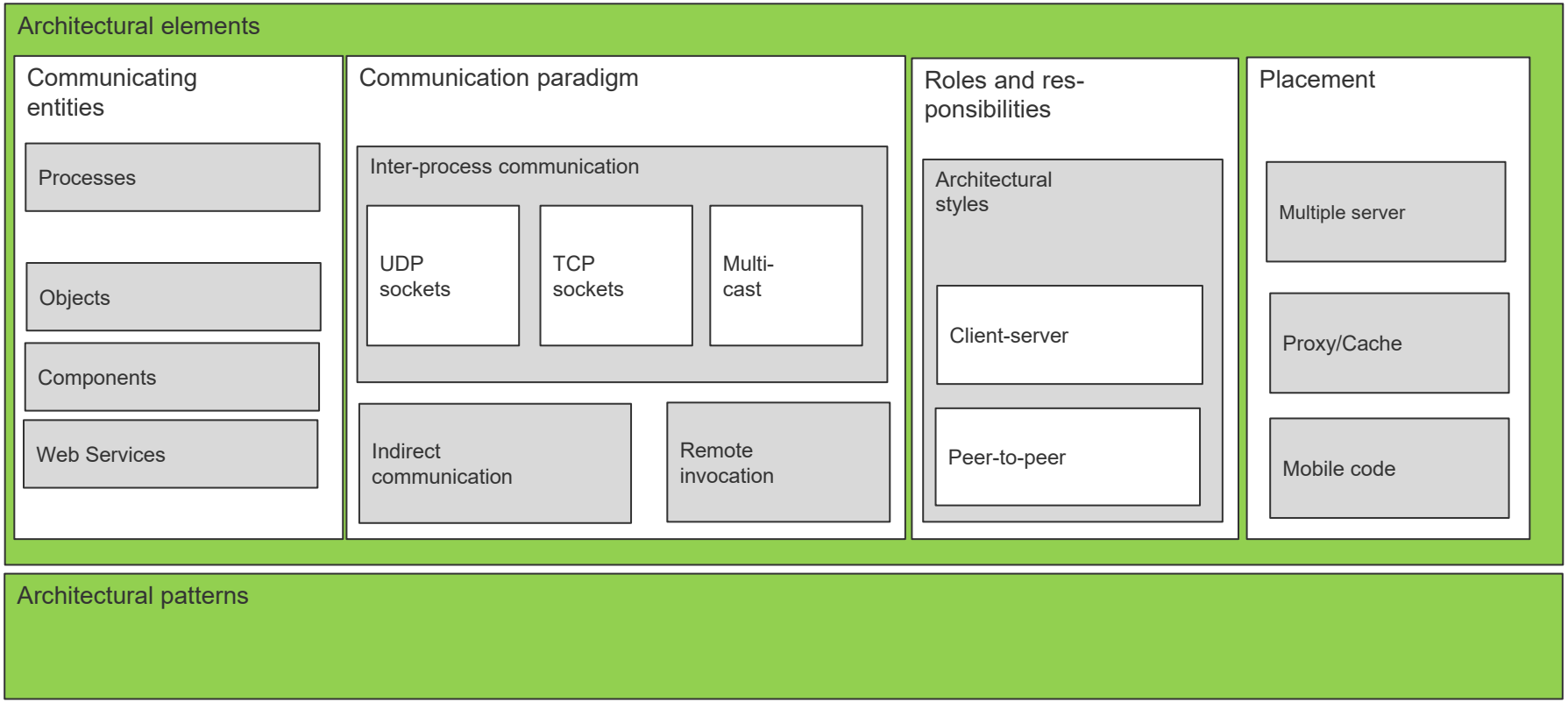
E: Replicated Web server

- Each server machine contains a complete copy of all hosted Web pages.
- Client requests are passed on to the servers in a round robin fashion.



Coulouris, Dollimore, Kindberg: *Distributed Systems: Concepts and Design*. 2011.

Recap: Architectural Model



Architectural elements

COMMUNICATION PARADIGM

An Architectural Model of Distributed Systems

Architectural elements

Communicating entities	Communication paradigm	Roles and responsibilities	Placement

How do they **communicate**, or, more specifically, what communication paradigm is used?

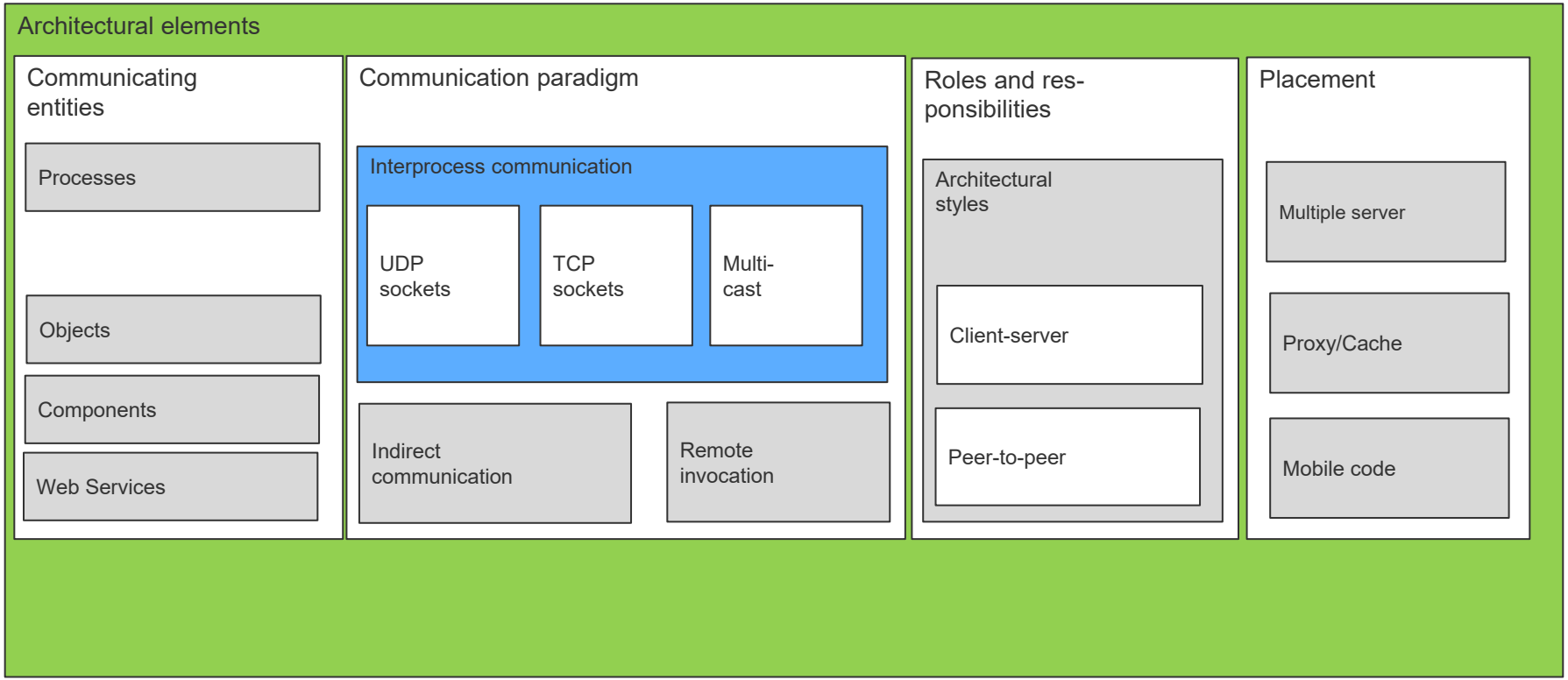
Types of Communication Paradigms

Interprocess communication

Remote invocation

Indirect communication

Types of Communication Paradigms



Freie Universität Berlin

Communication Paradigm

INTERPROCESS COMMUNICATION

Interprocess Communication

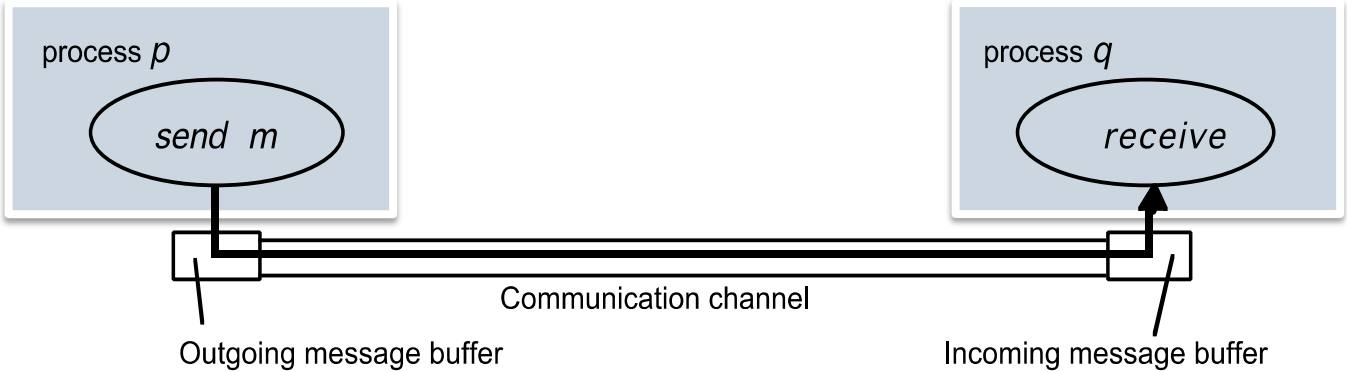
Interprocess Communication (IPC) mechanisms provide a low-level support to enable processes from different address spaces to connect and exchange information.

A process is an object of the operating system through which applications gain secure access to computer resources. Individual processes are isolated from each other for this purpose.

IPC is based on the exchange of messages (= bit sequences).

- A message is sent by the one process (the sender).
- It is received by another process (the receiver).

Motivation for IPC



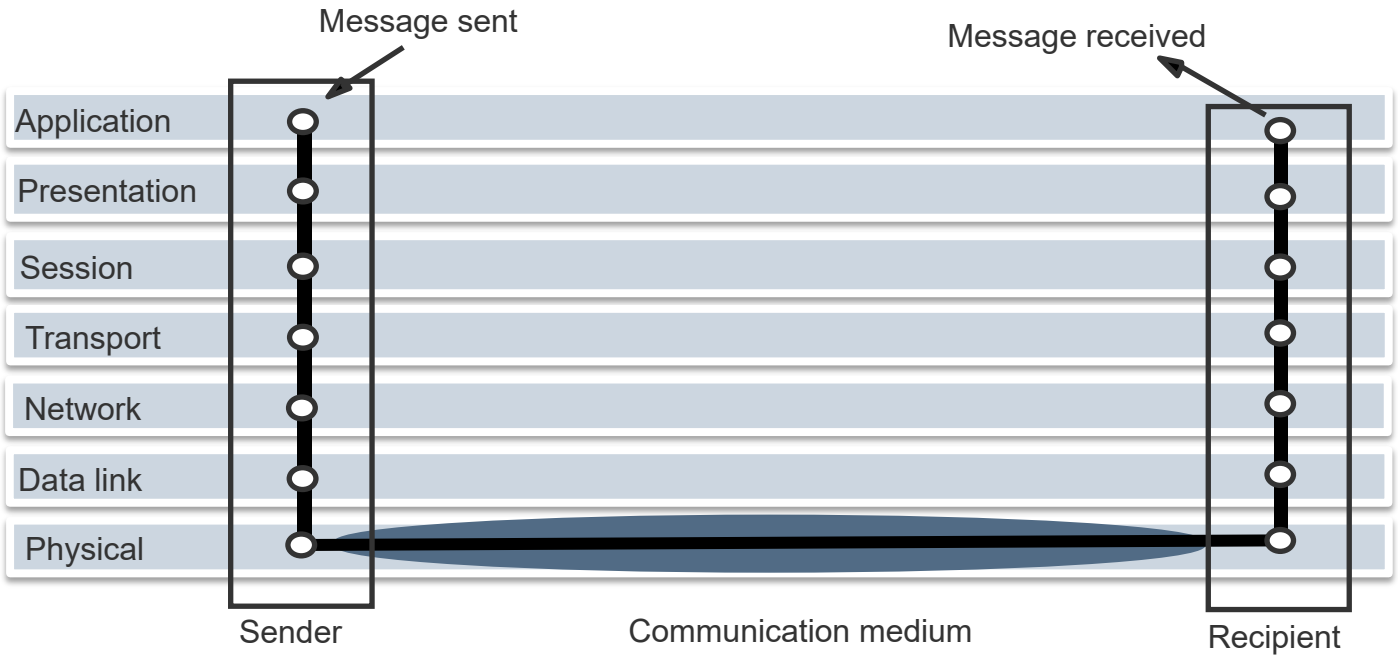
Protocols

- Protocol refers to a set of rules and formats to be used for communication between processes in order to perform a given task.

Specification of the sequence of messages that must be exchanged.

Specification of the format of the data in the messages.

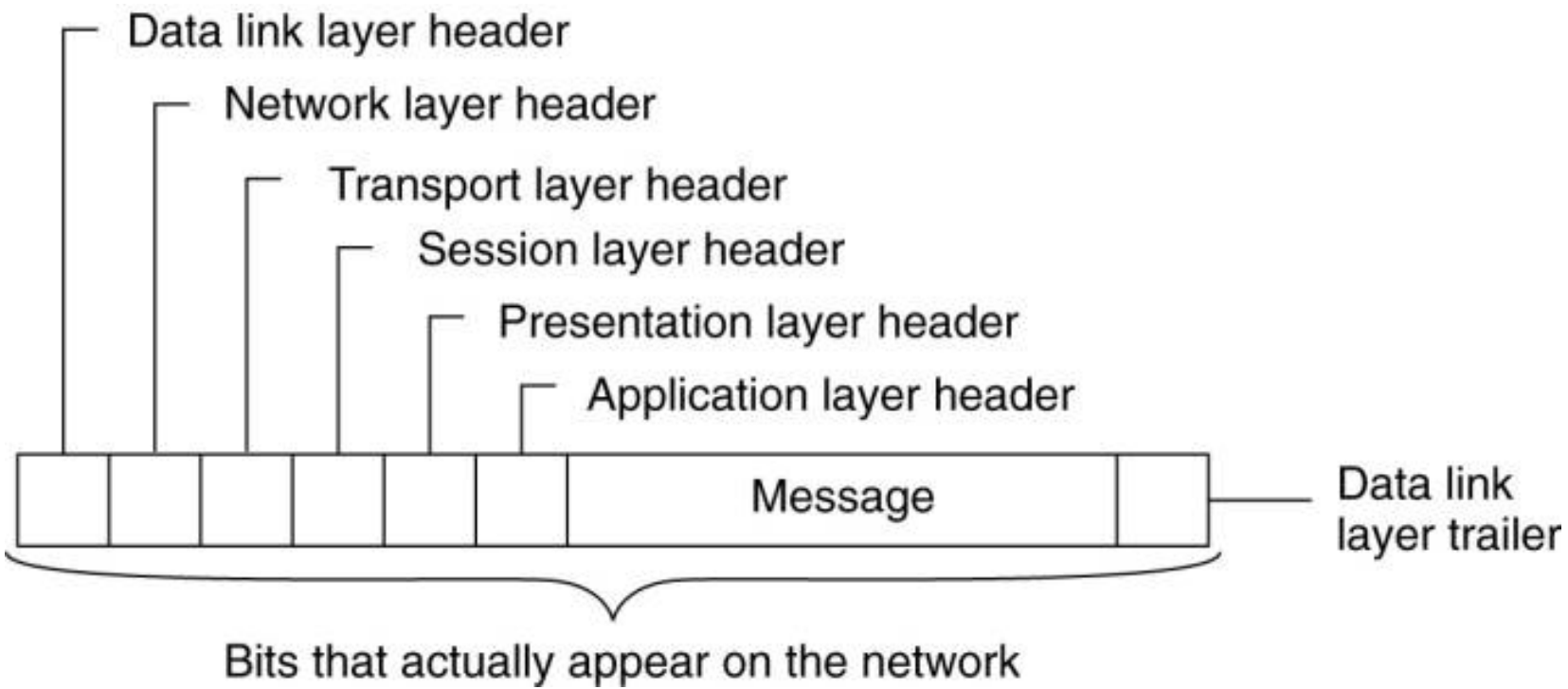
Protocol layers in the ISO Open Systems Interconnection (OSI) model



Connection-oriented communication: sending and receiving processes synchronize at every message = send and receive are blocking operation

Connectionless communication: send and receive operations are non-blocking

A typical message as it appears on the network



Tanenbaum & van Steen. Distributed Systems. Principles and Paradigms. 2007.

Layers ISO Model vs. TCP/IP Model

Application	Application
Presentation	
Session	
Transport	Transport
Network	Internet
Data link	Network Access
Physical	

TCP/IP Model

HTTP DNS SNMP	Application
TCP UDP	Transport
IP (IPv4 / IPv6)	Internet
Ethernet	Network Access

UDP vs. TCP

UDP (User Datagram Protocol)

- UDP differs from the IP service only in the additional specification of the ports: a message is sent as a datagram through the network without the arrival at the destination port being guaranteed (connectionless service).

TCP (Transmission Control Protocol)

- TCP establishes a virtual connection between a client port and a provider port and thus two opposing, reliable, sequence-true (FIFO) byte streams (connection-oriented service).

Realizing Interprocess Communication

UDP features

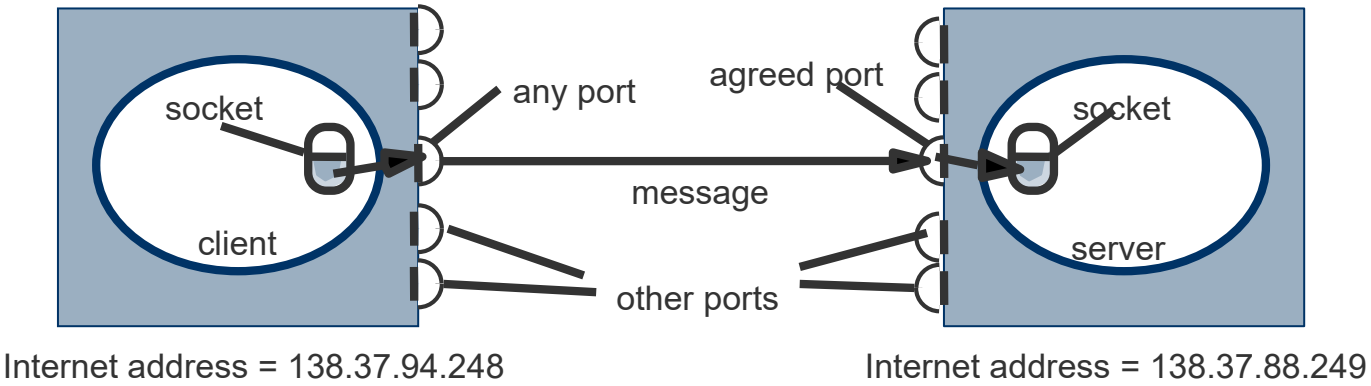
- UDP datagram encapsulated inside an IP package
- Header includes source and destination port numbers
- No guarantee of delivery
- Message size is limited
- Restricted to applications and services that do not require reliable delivery of single or multiple messages

TCP features

- Provides reliable delivery of arbitrarily long sequences of bytes via stream-based programming abstraction
- Connection-oriented service
- Before data is transferred, a bidirectional communication channel is established

Sockets

On approach to realize interprocess communication consists of transmitting a message between a socket in one process to a socket in another process,



Coulouris, Dollimore, Kindberg: *Distributed Systems: Concepts and Design*. 2011.

Ports vs. Sockets

A **port** is a communication endpoint for Internet transport services. It is identified by a port number in the range [0..65535]. The range [0..1023] is reserved for standard services, the range [1024..49151] for other services (cf. Internet Assigned Numbers Authority, www.iana.org).

A **socket** is a communication endpoint provided with a port number for use by processes. In the case of a TCP connection, a provider-side socket is associated with a client-side socket.

Interprocess communication

API FOR INTERNET PROTOCOLS

Java API: package java.net

Java provides class `InetAddress` that represents Internet addresses.

Method static `InetAddress getByName(String host)`

Can throw an `UnknownHostException`

Example

```
System.out.println(InetAddress.getByName("www.fu-berlin.de"));
        www.fu-berlin.de/160.45.170.10
System.out.println(InetAddress.getByName("localhost"));
        localhost/127.0.0.1
System.out.println(InetAddress.getLocalHost());
        wing.local/192.168.183.35
```

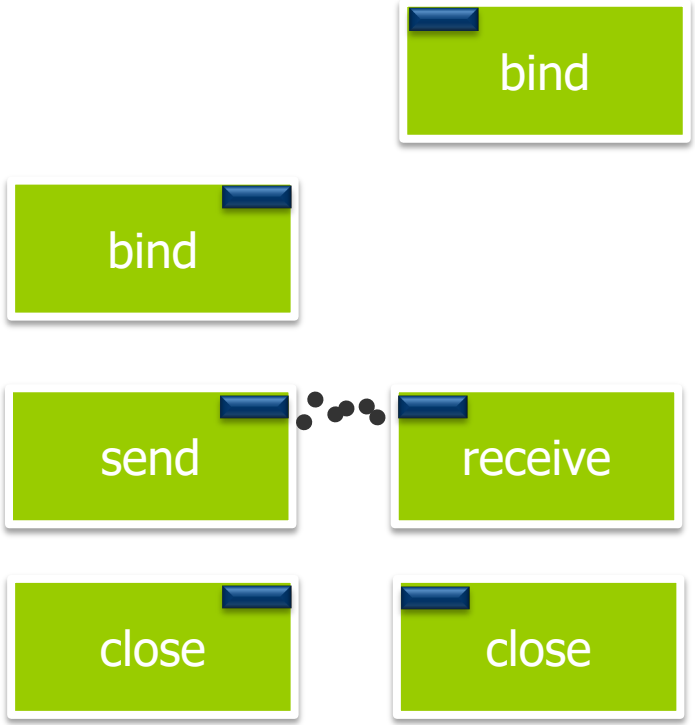
Slide adapted from Peter Löhr/Robert Tolksdorf
http://download.oracle.com/javase/6/docs/api/java/net/InetAddress.html

API for Internet protocols

UDP DATAGRAM COMMUNICATION

UDP Sockets

1. Client creates socket bound to a local port
2. Server binds its socket to a server port
3. Client/Server send and receive datagrams
4. Ports and sockets are closed



Slide adapted from Peter Löhr/Robert Tolksdorf

Issues related to Datagram Communication

Message Size

- Receiving process needs to specify an array of bytes of a particular size in which to receive a message. If the received message is to big it is truncated.

Blocking

- Datagram communication is carried out with a non-blocking *send* and a blocking *receive* operation.

Timeouts

- Timeouts can be set, in order to avoid that the receive operation waits indefinitely.

Receive from any

- The receive method does not specify an origin for messages.

Failure Model of UDP Datagrams

Integrity

- Messages should not be corrupted or duplicated
- Use of checksum reduces probability that received message is corrupted

Failures

- Omission failures: messages maybe dropped occasionally because of checksum error or no buffer space is available at source/destination
- Ordering: Messages can sometimes be delivered out of order

Using UDP for Applications

Advantage of UDP datagrams is that they do not suffer from overheads associated with guaranteed message delivery.

Example 1: Domain Name System

- DNS primarily uses UDP on port number 53 to serve requests
- DNS queries consist of a single UDP request from the client followed by a single UDP reply from the server

Example 2: VOIP

- No reason to re-transmit packets with bad speech data
- Speech data must be processed at the same rate as it is sent - there is no time to retransmit packets with errors

UDP datagram communication

JAVA API FOR UDP DIAGRAMS

Java API for UDP diagrams

Datagram communication is provided by two classes `DatagramPacket` and `DatagramSocket`

`DatagramPacket`

- Constructor that makes an instance out of an array of bytes comprising a message
- Constructor for use when receiving a message, message can be retrieved by the method `getData`

`DatagramSocket`

- Constructor that takes port number as argument for use by processes
- No-argument constructor for choosing a free local port

Example: UDP Echo Server

Create datagram socket at port 9876

```
import java.net.*;

class UDPServer {

        public static void main(String args[]) throws Exception {

        DatagramSocket serverSocket = new DatagramSocket(9876);

        byte[] receiveData = new byte[1024];
        byte[] sendData = new byte[1024];

...
```

...

Create space for new datagram

Receive datagram

Get IP addr port #, of sender

Create datagram to send to client

```
while (true) {
    DatagramPacket receivePacket = new
        DatagramPacket(receiveData, receiveData.length);

    serverSocket.receive(receivePacket);

    String sentence = new String(receivePacket.getData());
    InetAddress IPAddress = receivePacket.getAddress();
    int port = receivePacket.getPort();

    String capitalizedSentence = sentence.toUpperCase();

    sendData = capitalizedSentence.getBytes();

    DatagramPacket sendPacket = new DatagramPacket(sendData,
            sendData.length, IPAddress, port);
    serverSocket.send(sendPacket);
  }
}
}
```

Example: Java Client (UDP)

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


class UDPClient {
    public static void main(String args []) throws Exception {

        BufferedReader inFromUser = new BufferedReader(new
             InputStreamReader(System.in));
        DatagramSocket clientSocket = new DatagramSocket();

        InetAddress IPAddress = InetAddress.getByName("localhost");

        byte[] sendData = new byte[1024];
        byte[] receiveData = new byte[1024];

        String sentence = inFromUser.readLine();
. . .
```

Create input stream

Create client socket

Translate host-name to IP address using DNS

Create datagram with data-to-send, length, IP addr, port

Send datagram to server

Read datagram from server

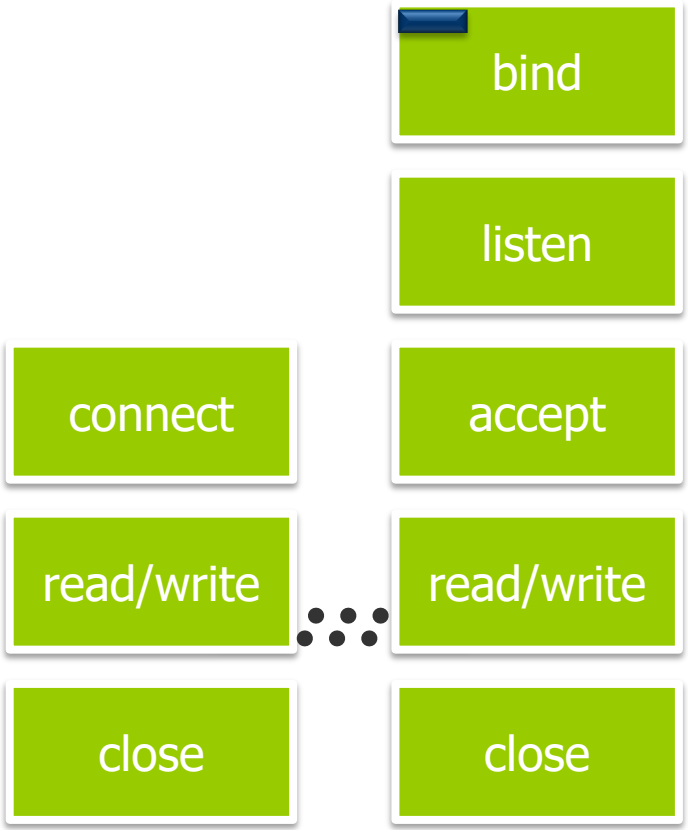
```
. . .
        sendData = sentence.getBytes();
        DatagramPacket sendPacket = new DatagramPacket(sendData,
                 sendData.length, IPAddress, 9876);
        clientSocket.send(sendPacket);

        DatagramPacket receivePacket = new DatagramPacket(receiveData,
                 receiveData.length);
        clientSocket.receive(receivePacket);

        String modifiedSentence = new String(receivePacket.getData());

        System.out.println("FROM SERVER: " + modifiedSentence);

        clientSocket.close();
    }
}
```

API for Internet protocols

TCP STREAM COMMUNICATION

TCP Sockets Communication

1. Server bind port
2. Server is ready and listening
3. Server is waiting for request, client sends request, server accepts
4. Client and server are connceted - bidirectional!
5. Connection is closed



Socket Programming with TCP

Client must contact server

- Server process must first be running
- Server must have created socket that welcomes client's contact

Client contacts server by:

- Creating client-local TCP socket
- Specifying IP address, port number of server process

When client creates socket: client TCP establishes connection to server TCP

When contacted by client, server TCP creates new socket for server process to communicate with client , which allows server to talk with multiple clients.

Hiding Network Characteristics by TCP

Application can choose the **message size**, means how much data it writes to a stream or reads from it.

TCP protocol uses an **acknowledgement** scheme to avoid lost data.

TCP supports **flow control** that means if the writer is too fast for the reader, then the writer is blocked until the reader consumed sufficient data.

Message identifiers are used by each IP packet. The recipient can therefore detect and reject **duplicates** or can **reorder** message if needed.

Before a pair of communication processes communicate they **establish a connection**.

Failure Model of TCP

In order to realize **reliable communication**, TCP streams use **checksums** to detect and reject corrupt packages and **sequence numbers** to detect and reject duplicate packets.

To deal with lost packages TCP streams use **timeouts and retransmissions**.

A broken connection has the following effects:

- The processes using the connection cannot distinguish between network failure and failure of the process at the other end of the connection
- The communication processes cannot tell whether the messages they have sent recently have been received or not.

Use of TCP

Many frequently used services run over TCP connections with reserved port numbers.

- **HTTP** [RFC 2068]: The Hypertext Transfer Protocol is used for communication between web browser and web server.
- **FTP** [RFC 959]: The File Transfer Protocol allows directories on a remote computer to be browsed and files to be transferred from one computer to another over a connection.
- **Telnet** [RFC 854]: Telnet provides access by means of a terminal session to a remote computer.
- **SMTP** [RFC 821]: The Simple Mail Transfer Protocol is used to send mail between computer.

http://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers

TCP Stream Communication

JAVA API FOR TCP

Java API for TCP streams

Java interface provides two classes `ServerSocket` and `Socket`

`ServerSocket`

- Class is intended to be used by server to create a socket at a server port for listening for connect requests from clients.

`Socket`

- Class is for use by a pair of processes with a connection
- The client uses a constructor to create a socket, specifying the DNS hostname and port of a server

TCP Echo Server

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

Create Server Socket

```
ServerSocket listen = new ServerSocket(1234);
```

```
while (true) {
```

Listens for a connection and accepts it.

```
Socket socket = listen.accept();
```

Input-Stream

```
BufferedReader in = new BufferedReader(new
                InputStreamReader(socket.getInputStream()));
```

Output-Stream

```
PrintStream out = new PrintStream(socket.getOutputStream());
```

. . .

```
while (true) {

    String message = in.readLine();

    if (message == null) {
            break;
    }
    String answer = message.replace('i', 'o');
    out.println(answer);
}
in.close();
out.close();

socket.close();
System.out.println("Socket closed.");

}}}
```

Read Input stream

Send Output stream

TCP Client

```
public class Client {

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

Create Socket And connect to server

```
    Socket socket = new Socket("localhost", 1234);

    PrintStream out = new PrintStream(socket.getOutputStream());

    BufferedReader in = new BufferedReader(new
    InputStreamReader(socket.getInputStream()));
```

BufferedReader reads Keyboard Input

```
    BufferedReader keyboard = new BufferedReader(new
            InputStreamReader(System.in));
```

. . .

```
while (true) {
        String message = keyboard.readLine();

        if (message == null)
                break;

        out.println(message);
        String answer = in.readLine();
        System.out.println("echo: " + answer);
}

in.close();
out.close();
socket.close();
}
}
```

Send text via output stream

Notice!

However, the service echo is quite limited in that it cannot have several sessions at the same time. If you want to use the service, you may have to wait until an active user closes the session.

What might be a solution?

Example EchoServer Extended

```
public class EchoServerExtended extends Thread {
    private Socket socket;
    private BufferedReader in;
    private PrintStream out;


    public EchoServerExtended(Socket socket) throws IOException{
        this.socket = socket;
        this.in = new BufferedReader(new
                  InputStreamReader(socket.getInputStream()));
        this.out = new PrintStream(socket.getOutputStream());
    }
```

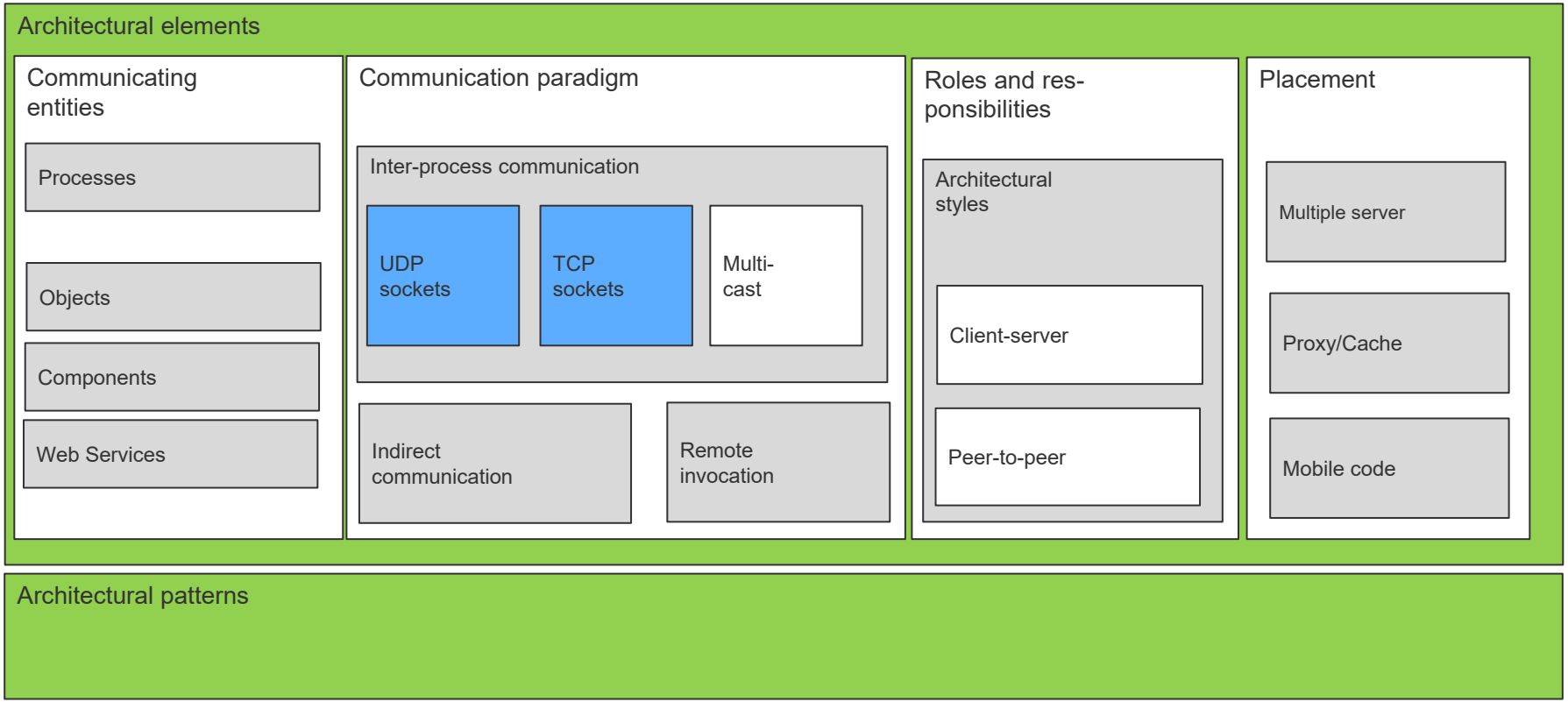
Further tasks of you as developer…

Sockets only provide the basic mechanisms, there is still work to be done, for example the implementation of more complex system models such as Request-Reply (for client-server) or group communication.

Above all, however, the necessity of homogeneous data representation in heterogeneous environments is a major issue.

These are the basic techniques for more complex middleware such as RPC, Java RMI. We talk about it soon ☺

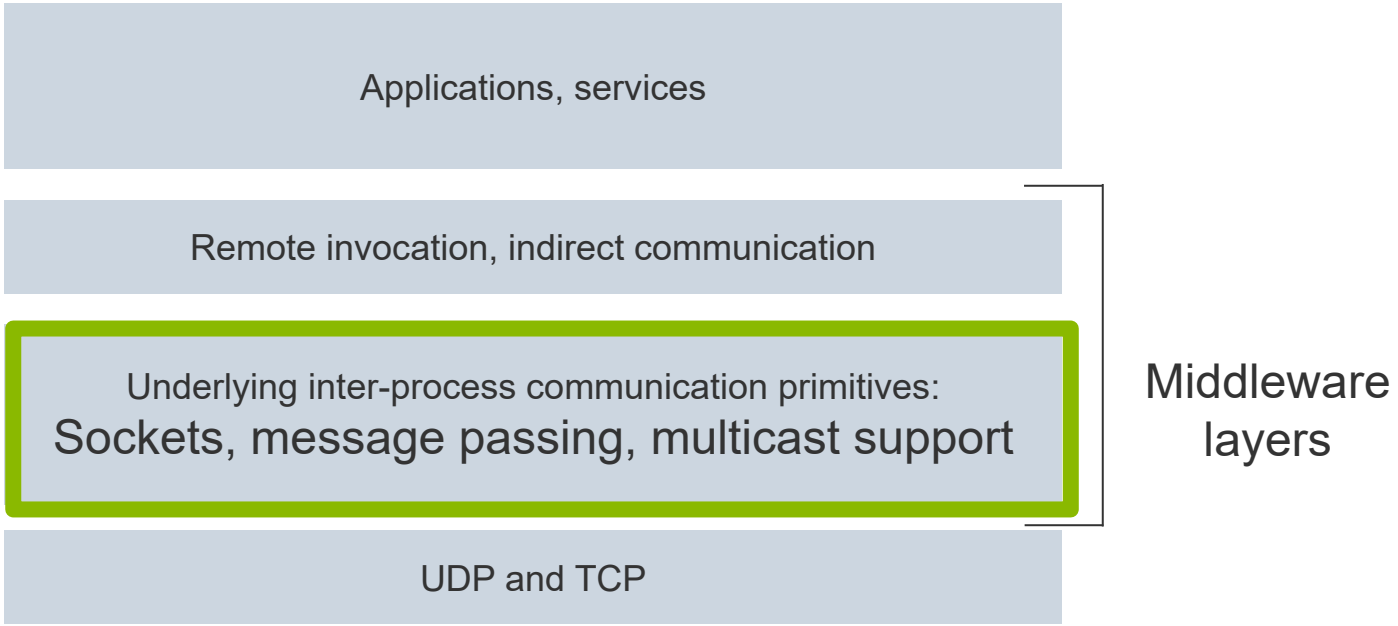
Recap: Architectural Model



Open Topics Interprocess Communication

- External data representation and marshalling
- Multicast communication
- Network virtualization: Overlay networks

Middleware Layers



Coulouris, Dollimore, Kindberg: *Distributed Systems: Concepts and Design*. 2011.

References

Main resources for this lecture:

- George Coulouris, Jean Dollimore, Tim Kindberg: *Distributed Systems: Concepts and Design*. 5th edition, Addison Wesley, 2011.
- Andrew S. Tanenbaum and Marteen van Steen.Distributed Systems. Principles and Paradigms.Pearson Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 2007.

Institute of Computer Science
Department of Mathematics and Computer Science

Freie Universität Berlin

Algorithms and Programming IV

Interprocess Communication (*cont.*)

Summer Term 2021 | 14.06.2021

Barry Linnert

Institute of Computer Science
Department of Mathematics and Computer Science

Freie Universität Berlin

Algorithms and Programming IV

Remote Invocation

Summer Term 2021 | 14.06.2021

Barry Linnert