

Hands-On Training Part I

Networking and Concurrency

Distributed Systems

Business Informatics, BSc.

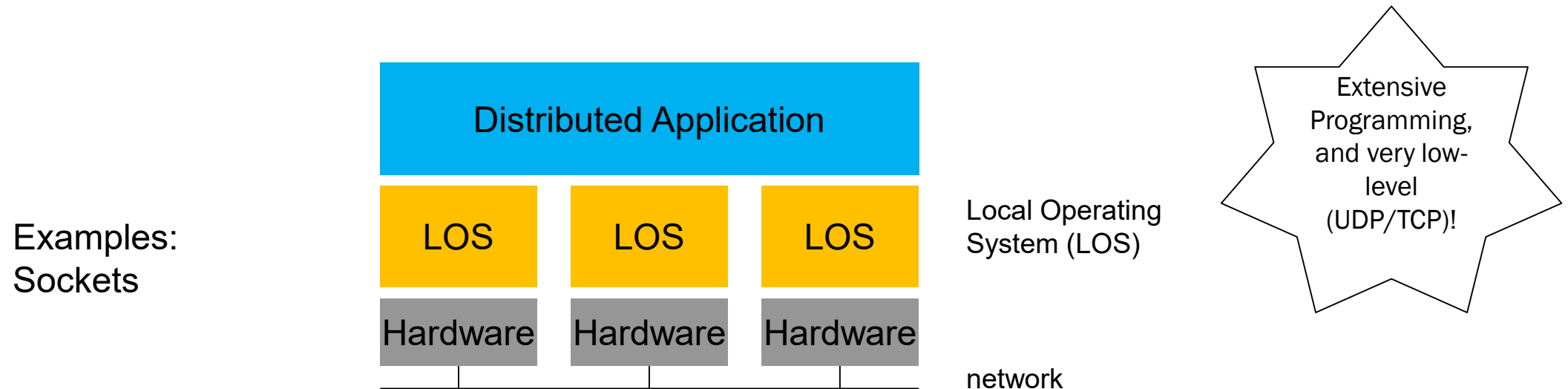
Middleware and Distributed Systems

Network operating system

- Provides basic communication primitives and concurrency capabilities (processes and threads).

Distributed application

- Implements application-specific communication using capabilities of the OS.



Network Communication

Message exchange is the foundation of communication in a network.

- Basic Operations

- **send**

- There is a message queue for each target
 - The sending processes put their message in this remote queue.

- **receive**

- The receiving process pulls messages from their local queue

- Communication modes

- Synchronous Communication
 - Asynchronous Communication

- Combinations of different communication modes is possible

- **send** and **receive** could be blocking or non-blocking

Communication Endpoints

Message endpoints are defined by the pair of internet address and port.

- A port is a message target within a computing node.
 - There is one receiver for each port, but may have many senders.
 - A process may use several ports for receiving messages.
 - Each process can send a message to a known port.
- Hard-coded internet addresses make location transparency difficult. Location transparency could be achieved by a naming service.

Sockets and Protocols

Sockets are the interface to the network between the application or the middleware to the operating system.

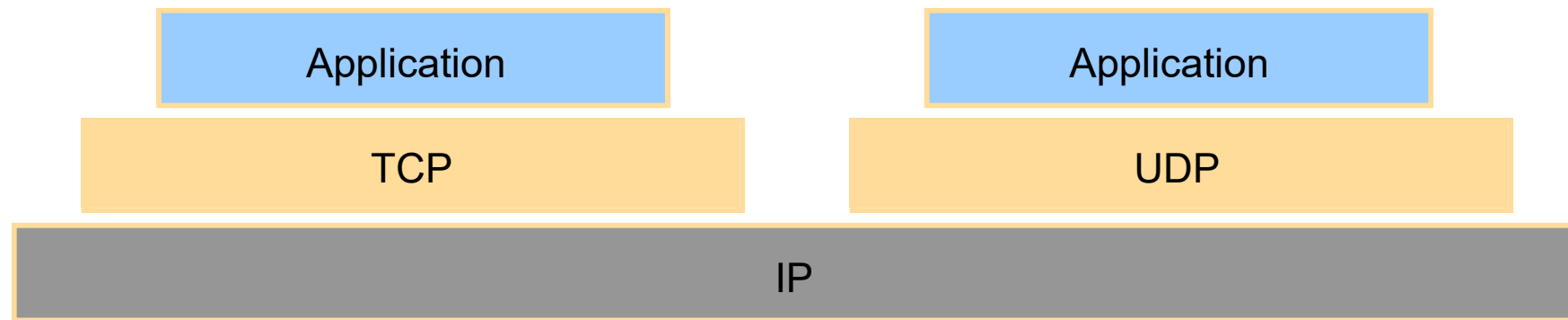
Sockets are endpoints of interprocess communication.

Each process creates and uses a socket to send or receive messages.

Sockets are associated to a certain communication protocol, e.g. UDP or TCP

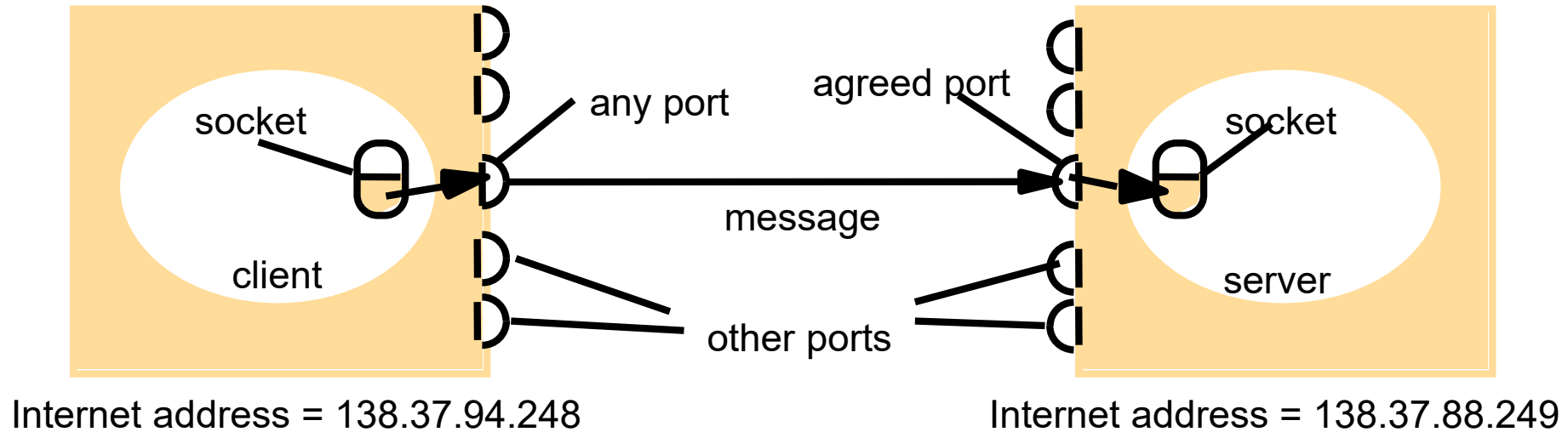
Sockets and Protocols

Conceptional view of a programmer on a TCP/IP and UDP/IP internet



Quelle: Coulouris et al.: Distributed Systems

Sockets and Protocols



A Socket is bound to an internet address and a port.

The port of a socket of a client may be anonymous within the client application.

In general, the port of a server is well-known.

Network Programming with Java

Sockets are provided by the most operating systems

The usage of system calls in the C programming language differ from OS to OS (Windows, Linux, macOS)

Java introduces a OS-independent API to use the OS's sockets (package **java.net**)

The Java implementations hide the details of the OS-specific system calls from the Java programmer.

Java-API Internet Addresses

Internet Addresses

- Class **`java.net.InetAddress`**
- Instances of the class represent the internet address of a host.
- There are several options to create instances. One is e.g. to use the DNS-hostname in a static method

```
InetAddress hostAddress =  
    InetAddress.getByName("www.reutlingen-university.de");
```

- If the hostname cannot be resolved an **`UnknownHostException`** will be thrown.
- It is also possible to use the IP address (v4 or v6) of the host.

UDP communication with Java

Message size

- Receiver provides a byte array of a certain size to store the payload of the incoming message (data).

Communication mode

- Non-blocking **send** and blocking **receive**
- Messages are represented by instances of the class Datagram. A transmitted Datagram is stored in a queue of the receiver's socket.
- **receive** pulls a datagram from the socket's queue
- **receive** blocks until a datagram is stored in the queue or until a defined timeout is triggered.

UDP Communication with Java

Timeouts

- A blocking **receive** is used by servers
- Endless waiting is not always the best implementation.
 - A timeout can be defined for a socket.
 - But the choice of timeouts is always difficult for an application.

Receiving messages

- **receive** method accepts messages from different senders.

UDP Communication using Java

When to apply UDP within programs?

- Either programs have to take unreliability into account,
- or they have to implement an own communication error model.
- On the other hand side, the usage may be attractive because of little overhead in setting up the communication.

Let's have a look at the Java API for UDP support!

Java API UDP Communication

DatagramPacket

- Instances of class **DatagramPacket** can be transmitted between remote processes.
- CLASS **DatagramPacket** defines properties for communication
 - The Payload of a **DatagramPacket** object (the message itself) is encoded as a byte array, further the length of the message is given (the array may be larger than the message itself).
 - Internet address and port of the sender (important for the receiver to address its answer).

Byte-Array	Länge der Nachricht	Internetadresse	Port
------------	---------------------	-----------------	------

- The class provides getter methods to access these properties: **getData ()** , **getAddress ()** , **getPort ()** abgefragt werden

Java API UDP Communication

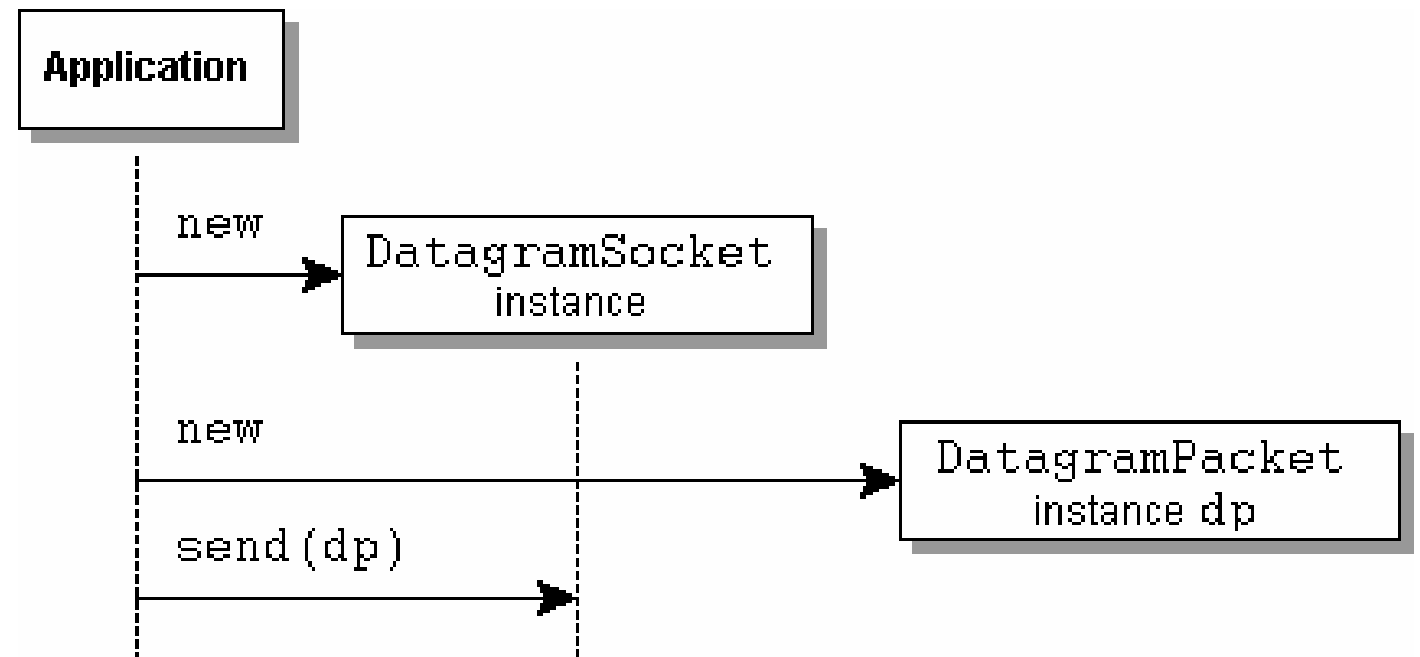
DatagramSocket

- class **DatagramSocket** provides a socket implementation for sending and receiving **DatagramPacket** objects.
- The port number may be passed as a parameter to the constructors of the class.
 - Client anonymous (a free port is selected by the implementation)
 - Server needs an explicit, well-known port.
- The creation may fail, in this case a **SocketException** will be thrown (e.g. port already in use).
- Important methods of class **DatagramSocket**:
 - **send**,
 - **receive**,
 - **setSoTimeout**, -> Timeout for receive (SocketTimeoutException)

Java API UDP Communication

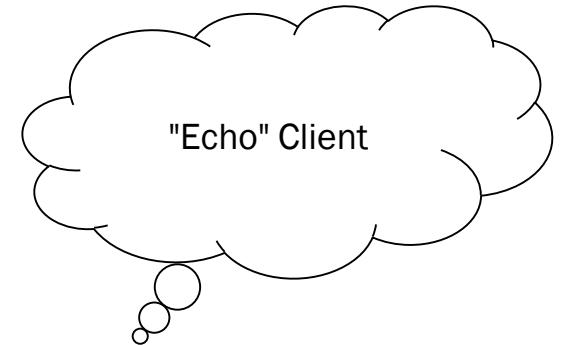
Timing Diagram UDP Client

- Create a new Socket.
- Create a DatagramPacket containing your message.
- Send the DatagramPacket via the socket object.



Java API UDP Communication

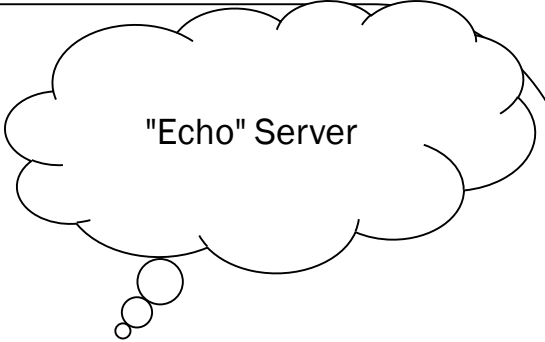
```
//imports missing!
public class UDPClient{
    public static void main(String args[]){
        // args give message contents and server hostname
        DatagramSocket aSocket = null;
        try {
            aSocket = new DatagramSocket();
            byte [] m = args[0].getBytes();
            InetAddress aHost = InetAddress.getByName(args[1]);
            int serverPort = 6789;
            DatagramPacket request =
                new DatagramPacket(m, m.length, aHost, serverPort);
            aSocket.send(request);
            byte[] buffer = new byte[1000];
            DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
            aSocket.receive(reply);
            System.out.println("Reply: " + new String(reply.getData()));
        }catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
        }catch (IOException e){System.out.println("IO: " + e.getMessage());}
    }finally {if(aSocket != null) aSocket.close();}
}
```



Java API UDP Communication

```
//imports missing!
public class UDPServer{
    public static void main(String args[]){
        DatagramSocket aSocket = null;
        try{
            aSocket = new DatagramSocket(6789);
            byte[] buffer = new byte[1000];
            while(true){
                DatagramPacket request = new DatagramPacket(buffer, buffer.length);
                aSocket.receive(request);
                DatagramPacket reply = new DatagramPacket(request.getData(), request.getLength(),
                                                            request.getAddress(), request.getPort());

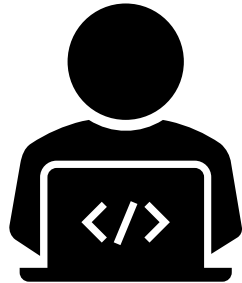
                aSocket.send(reply);
            }
        }catch (SocketException e){
            System.out.println("Socket: " + e.getMessage());
        }catch (IOException e) {
            System.out.println("IO: " + e.getMessage());
        }finally {
            if(aSocket != null) aSocket.close();
        }
    }
}
```



"Echo" Server

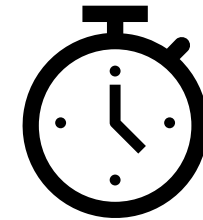
Hands-On Training: 1 Networking and Concurrency

Before you start
the exercise,
implement the UDP
Echo Client/Server
from the slides



Exercise 1.1

Time Server and Client



60 min

Client and Server
are always a new
Gradle project

TCP Streams

TCP streams provide a byte stream abstraction hiding the following network properties:

1. message size
2. lost messages
3. control flow
4. duplicate messages and ordering
5. message targets (instead a connection is established)

General problems concerning stream communication

1. Agreement on data elements

- Sender and receiver both know how data is formatted.

2. Synchronization of sender and receiver

- Blocking of receiver if there is no data in the queue and
- Blocking of the sender if queue is full (data flow control).

3. Threads

- Improve availability of server during a communication.

TCP Streams

Error model

- Checksums to detect and delete defect packages
- Running numbers to detect and delete duplicate packets
- Timeouts to repeat transmission of packets if no confirmation was received.
- However, there is no guarantee that a connection can be established or is constantly established for the desired amount of time.

Java API for TCP Streams

Client and server have different roles in the process of establishing a connection

- Server waits on a **connect** performed by the client on its **ServerSocket**
- Server manages a queue for **connect** requests
- Connection is created by an **accept** of the server
- The connection is bidirectional!

Java API for TCP Streams

class **ServerSocket**

- **ServerSocket**
 - waits for **connect** request of the client.
- **accept**—blocks if the queue of request is empty.
- Otherwise the server creates a new instance of class **Socket** that is used for the following communication with the client.

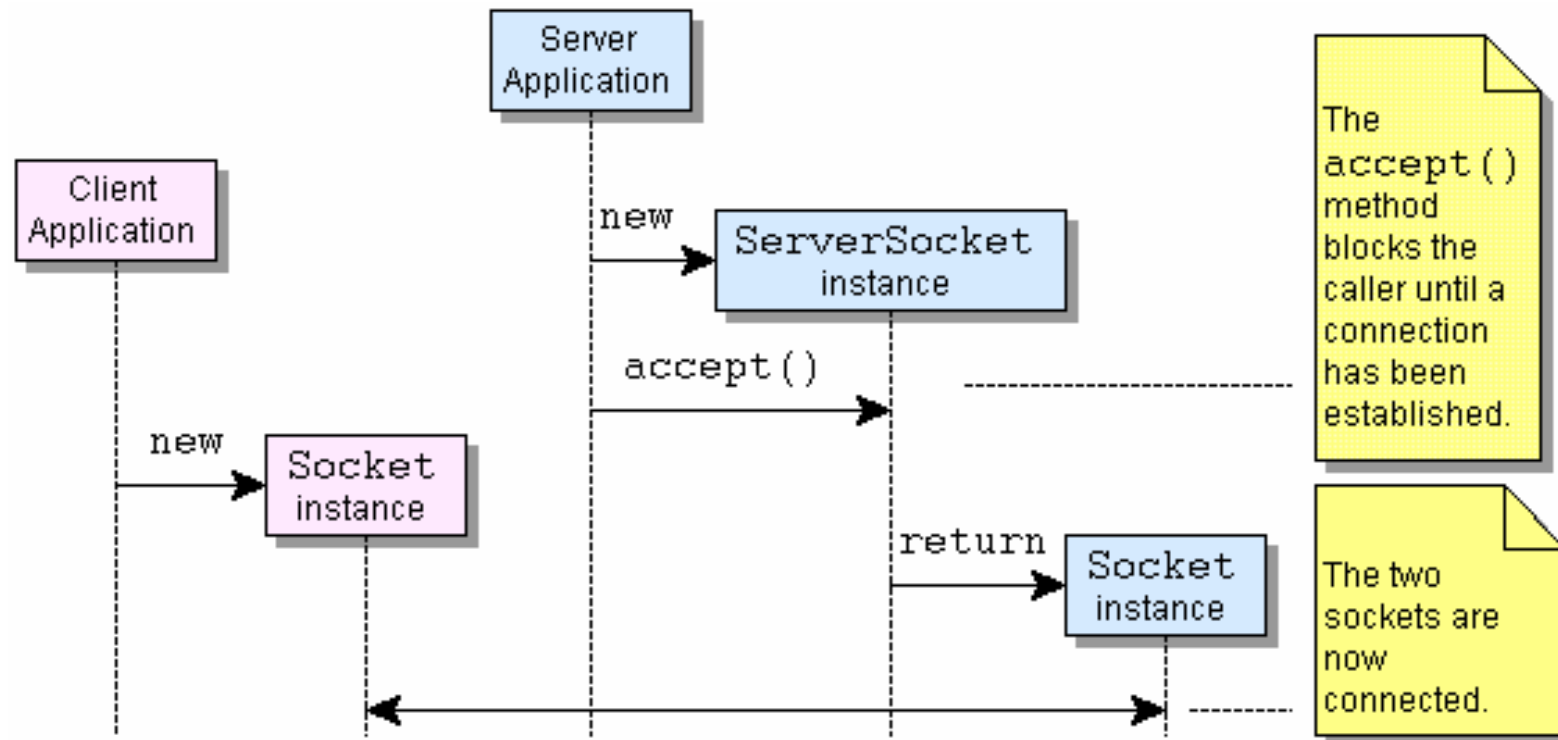
Java API for TCP Streams

Class **Socket**

- Client constructs a new **Socket** passing information about hostname and port of the server.
 - Constructor of class is overloaded, several variants exist (see Java API doc).
- Server gets the counterpart of the connection as a new instance of class **Socket** as a result of the **ServerSocket**'s **accept** method.
- Methods **getInputStream** und **getOutputStream** return references to byte streams with which data can be written or read using the socket.

Java API for TCP Streams

Timing Diagram for establishing a TCP connection



Java API for TCP Streams

```
public class TCPClient {
    public static void main (String args[]) {
        // arguments supply message and hostname of destination
        Socket s = null;
        try{
            int serverPort = 7896;
            s = new Socket(args[1], serverPort);
            DataInputStream in = new DataInputStream( s.getInputStream());
            DataOutputStream out =
                new DataOutputStream( s.getOutputStream());
            out.writeUTF(args[0]);           // UTF is a string encoding
            String data = in.readUTF();
            System.out.println("Received: "+ data) ;
        }catch (UnknownHostException e){
            System.out.println("Sock:"+e.getMessage());
        }catch (EOFException e){System.out.println("EOF:"+e.getMessage());
        }catch (IOException e){System.out.println("IO:"+e.getMessage());
        }finally {if(s!=null)
            try {s.close();}
            catch (IOException e){System.out.println("close:"+e.getMessage());}}
    }
}
```

Imports are missing!
Formatting is not
perfekt!

Java-API for TCP Streams

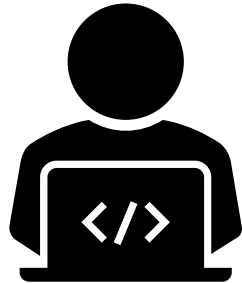
```
import java.net.*;
import java.io.*;
public class TCPServer {
    public static void main (String args[]) {
        try{
            int serverPort = 7896;
            ServerSocket listenSocket = new ServerSocket(serverPort);
            while(true) {
                Socket clientSocket = listenSocket.accept();
                DataInputStream in =
                    new DataInputStream( clientSocket.getInputStream());
                DataOutputStream out =
                    new DataOutputStream( clientSocket.getOutputStream());
                String data = in.readUTF();
                out.writeUTF(data);
            }
        } catch(Exception e) {System.out.println("Listen :"+e.getMessage());}
    }
}
```

DataInputStream
and
DataOutputStream
are just used for
convenience!

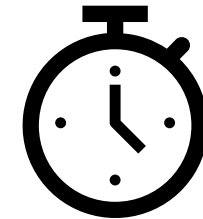
TCP Echo Server

Hands-On Training: 1 Networking and Concurrency

Before you start
the exercise,
implement the TCP
Echo Client/Server
from the slides



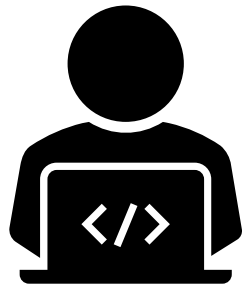
Exercise 1.2
Guess Game



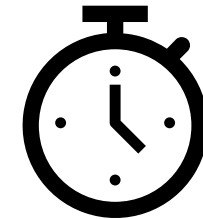
90 min

Hands-On Training: 1 Networking and Concurrency

Remember the
Java Thread API
from the lectures



Exercise 1.3 Concurrent Search



90 min

Working with URLs and HTTP connections in Java

Class `java.net.URL`

- References a resource in the internet
- Several constructors
 - E.g. String representation of URL as the only parameter
- Throws a **MalformedURLException**, if
 - the protocol is not valid, or
 - the String contains not a valid URL format.

Working with URLs and HTTP connections in Java

Class `java.net.URLConnection`

- Abstract super class for all sub classes representing a certain kind of URL connection
- Instances can be used for reading and writing
 - E,g, read a web site from an URL.
- Examples of subclasses for special URL connections:
 - `HttpURLConnection`, `HttpsURLConnection`
 - `JarURLConnection`

Working with URLs and HTTP connections in Java

Usage:

1. Create a URL object

- `URL myUrl = new URL("http://www.reutlingen-university.de");`

2. Open a connection

- `URLConnection urlCon = myUrl.openConnection();`

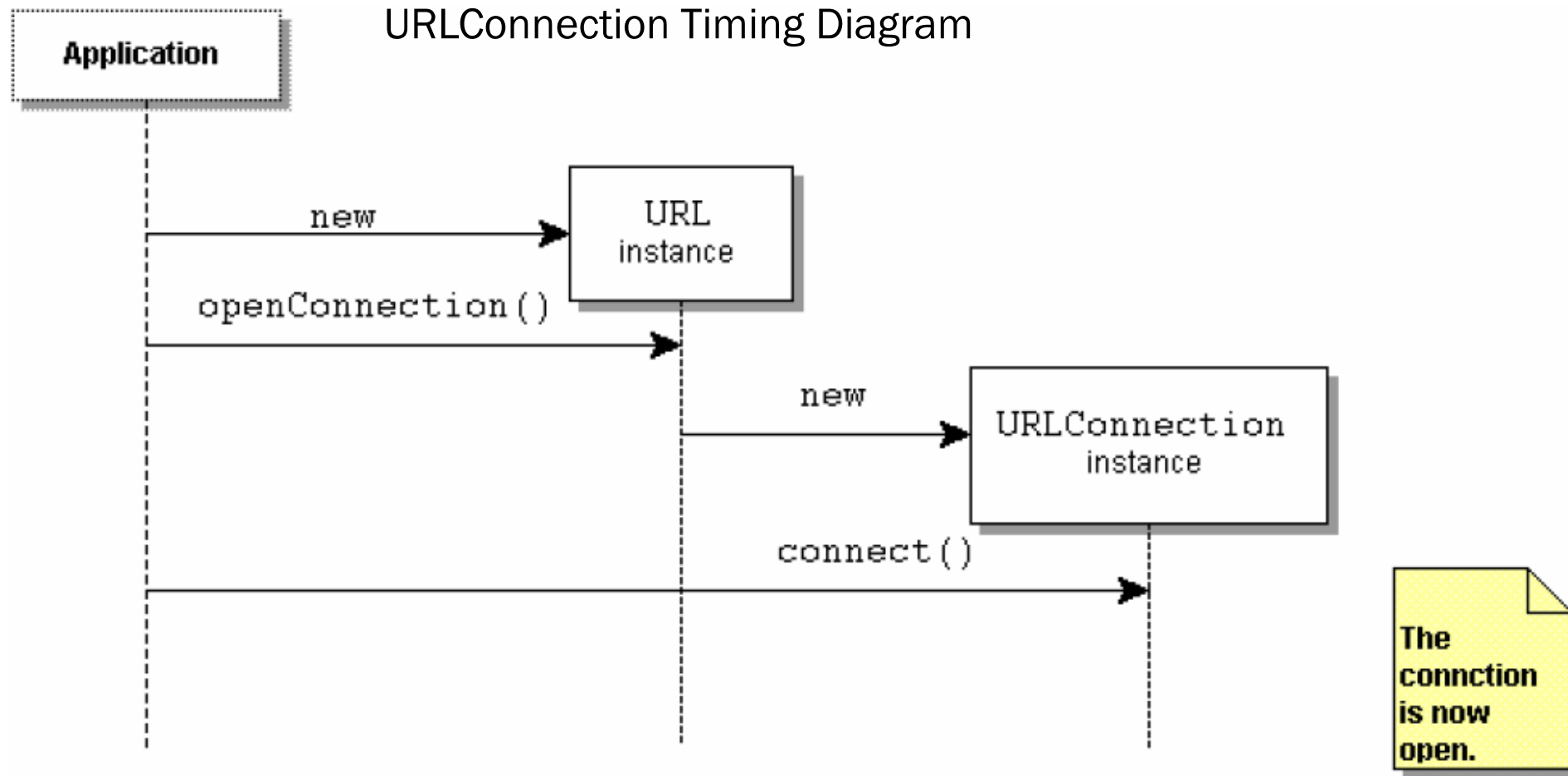
3. Get a byte stream for input or output operations

- `urlCon.getInputStream(); //urlCon.getOutputStream();`

4. Read or write data to the stream.

5. Close the streams

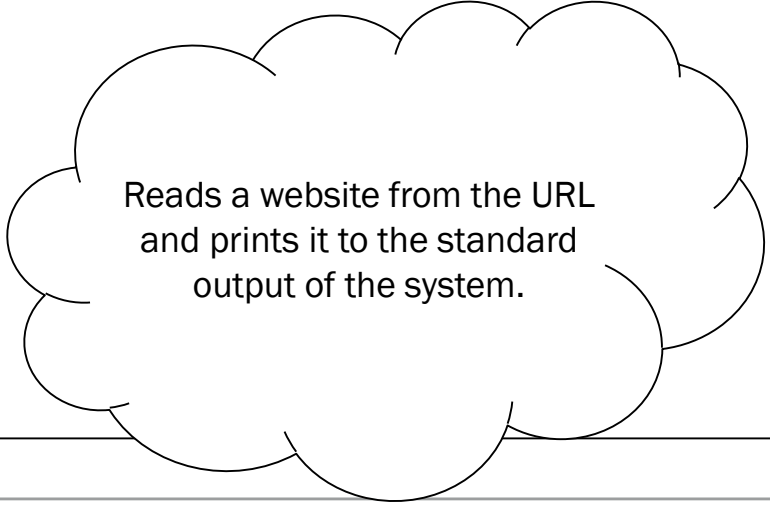
Working with URLs and HTTP connections in Java



Working with URLs and HTTP connections in Java

```
import java.net.*;

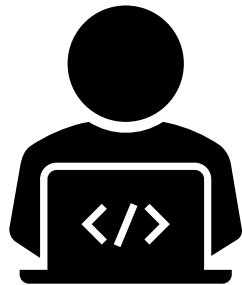
public class WebCrawler {
    public static final void main(String args[]) throws Exception {
        if (args.length!=1) {
            System.out.println("Pass the target URL via command line arguments (args)");
            System.exit(1);
        }
        URL myUrl = new URL(args[0].trim());
        HttpURLConnection con = (HttpURLConnection)myUrl.openConnection();
        BufferedReader read= new BufferedReader(new InputStreamReader(con.getInputStream()));
        String line = read.readLine();
        while(line!=null) {
            System.out.println(line);
            line = read.readLine();
        }
    }
}
```



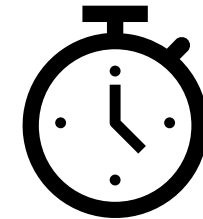
Reads a website from the URL
and prints it to the standard
output of the system.

Hands-On Training: 1 Networking and Concurrency

Start by realizing and testing the WebCrawler class from the previous slide.



Exercise 1.4
Web-Crawler



120 min

Conclusions

Developing distributed systems without an additional middleware in Java is possible, but very low-level.

Package `java.net` provides Sockets to realize inter-process communication over the network.

The API supports 1:1 communication using TCP or UDP, as well as 1:n communication (Multicast) based on UDP (not used in the hands-on training).

Besides socket-based communication, the Java API also provides classes for URL based communication using HTTP.

Further, the Java Thread API allows the development of efficient clients and servers in the context of distributed systems.