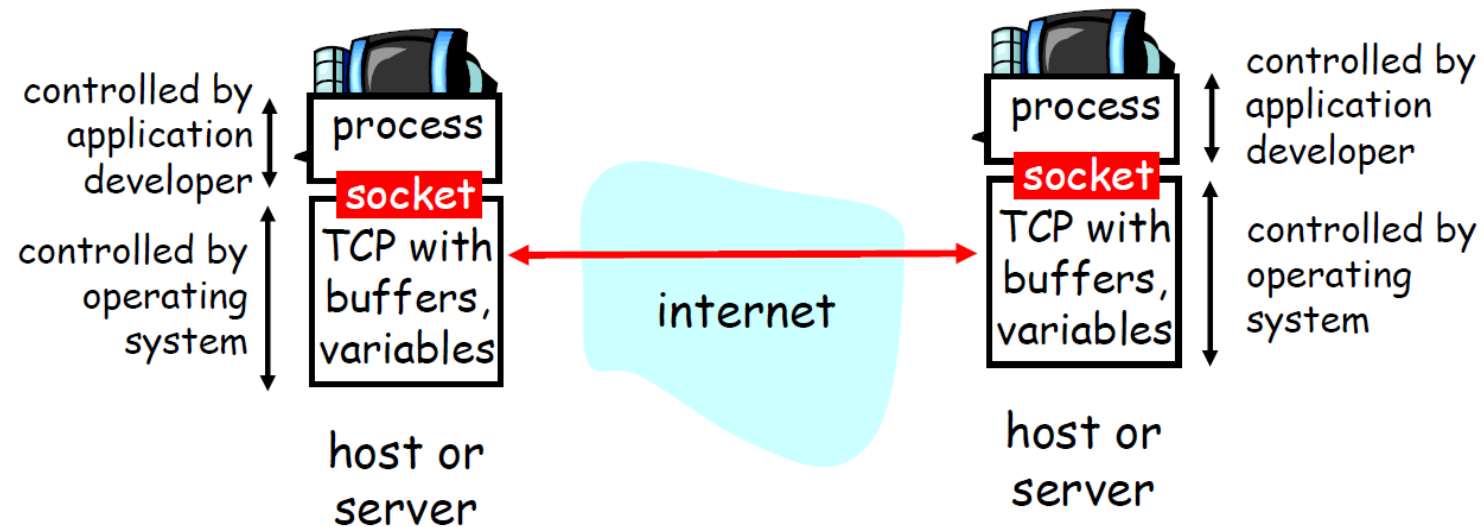


TCP Socket Programmierung und Threads

Kurzer Überblick

Sockets

- Sind im Betriebssystem integriert, TCP und UDP
- Werden von den Applikationen explizit erzeugt, genutzt und freigegeben
- Basieren auf dem Client/Server Paradigma



Socket-Programmierung via TCP

- Der Server Prozess muss aktiv sein und ein Socket erzeugt haben.
- Der Client kontaktiert den Server indem er:
 - Ein Socket erzeugt
 - Die IP Adresse und den Port des Server-Prozesses spezifiziert
 - Durch das Erzeugen des Sockets wird eine Verbindung zum Server hergestellt
- Sobald der Server vom Client kontaktiert wird:
 - Erzeugt der Server ein neues Socket über das er mit dem Client kommuniziert
 - So kann der Server mit mehreren Clients kommunizieren
- Kommuniziert wird über einen Streams

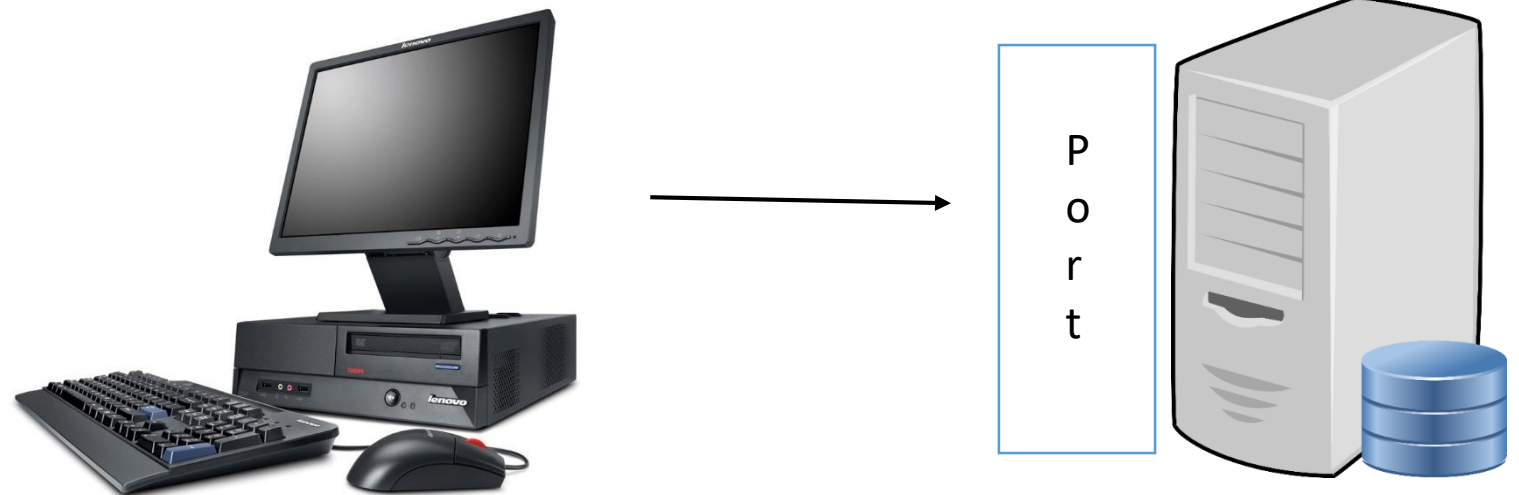
Ports

Um eintreffende Daten einem laufenden Prozess zuzuordnen werden Ports verwendet.

Gültige Portnummern sind 0 bis 65535

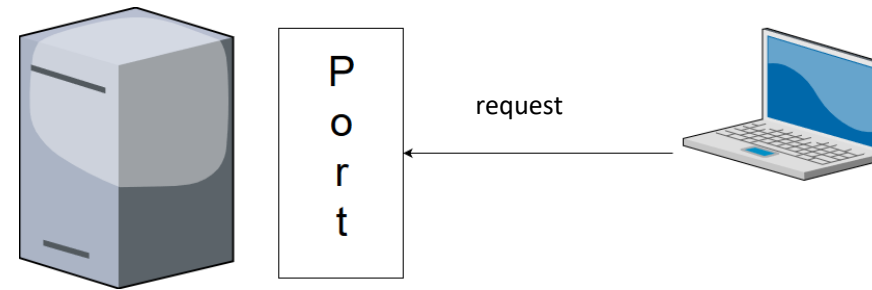
Bekannte Services:

ftp	(tcp) 21
telnet	(tcp) 23
smtp	(tcp) 25
login	(tcp) 513

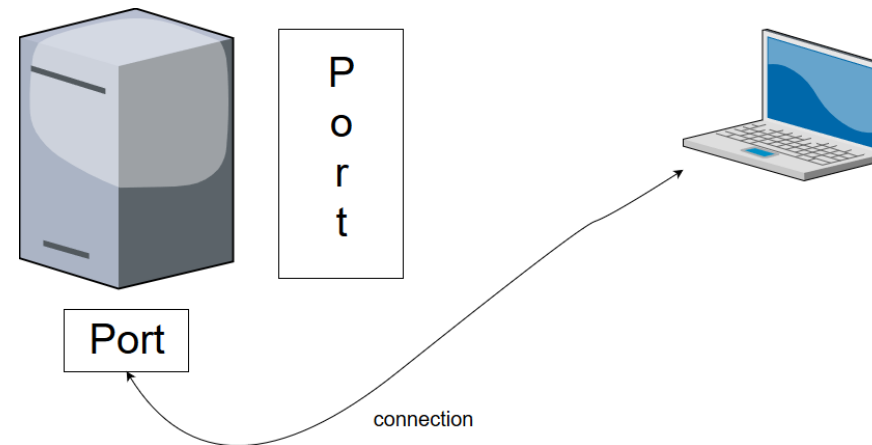


Verbindungsaufbau

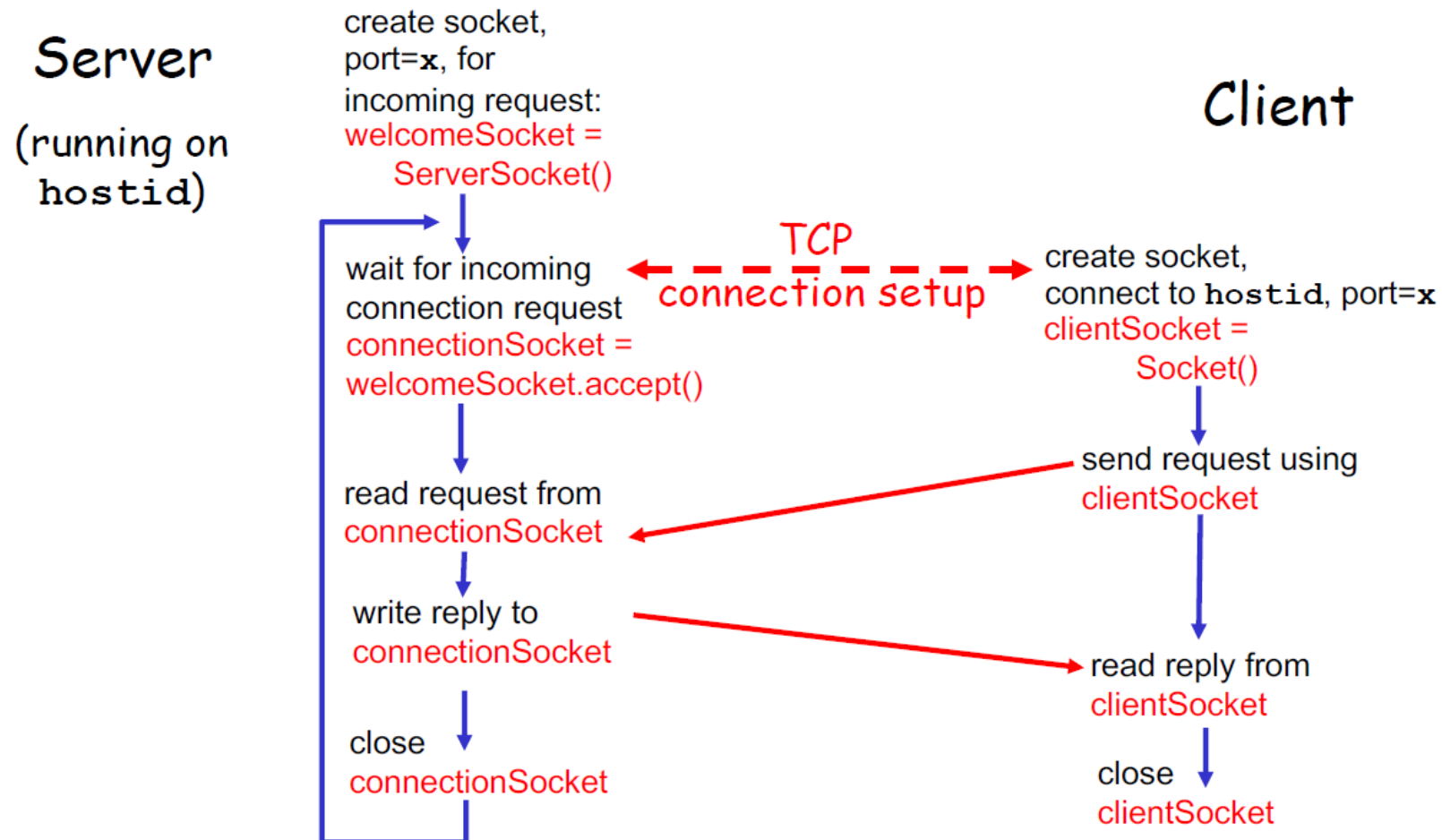
1. Schritt



2. Schritt



Client/Server Interaktion TCP


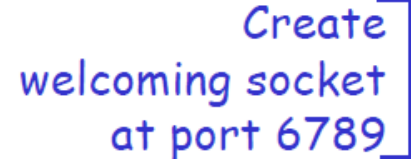


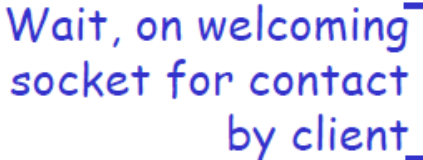
Example: Java Server (TCP)

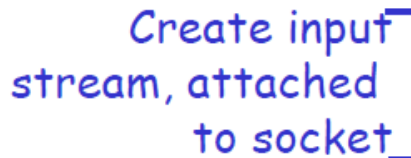
```
import java.io.*;
import java.net.*;

class TCPServer {

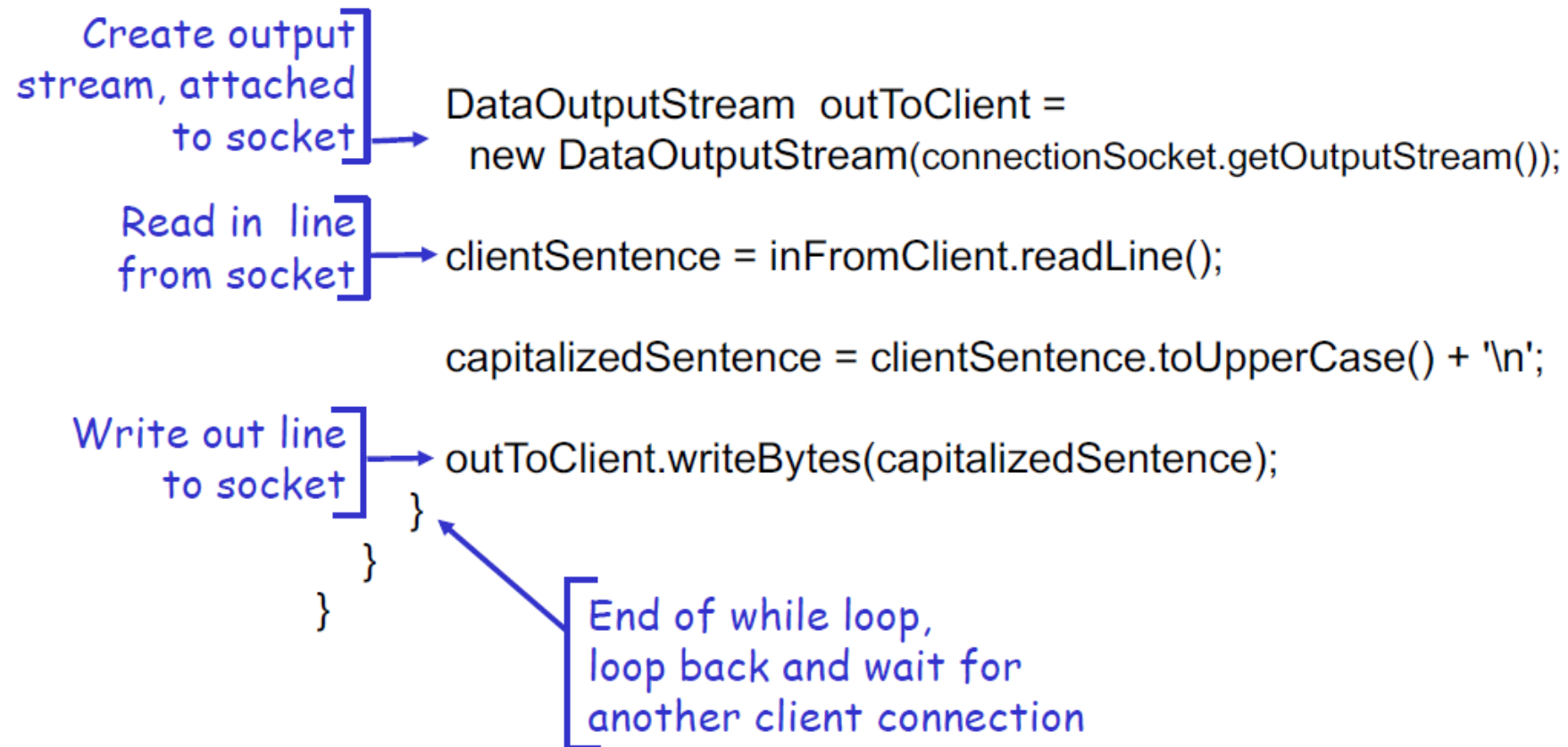
    public static void main(String argv[]) throws Exception
    {
        String clientSentence;
        String capitalizedSentence;

        
        
        ServerSocket welcomeSocket = new ServerSocket(6789);

        
        while(true) {
            Socket connectionSocket = welcomeSocket.accept();

            
            BufferedReader inFromClient =
                new BufferedReader(new
                    InputStreamReader(connectionSocket.getInputStream()));
        }
    }
}
```

Example: Java Server (TCP)



Example: Java Client (TCP)

```
import java.io.*;
import java.net.*;
class TCPClient {

    public static void main(String argv[]) throws Exception
    {
        String sentence;
        String modifiedSentence;

        Create input stream ] → BufferedReader inFromUser =
                               new BufferedReader(new InputStreamReader(System.in));

        Create client socket, connect to server ] → Socket clientSocket = new Socket("hostname", 6789);

        Create output stream attached to socket ] → DataOutputStream outToServer =
                                                    new DataOutputStream(clientSocket.getOutputStream());
```

Example: Java Client (TCP)

Create
input stream
attached to socket

Send line
to server

Read line
from server

```
BufferedReader inFromServer =  
    new BufferedReader(new  
        InputStreamReader(clientSocket.getInputStream()));  
  
sentence = inFromUser.readLine();  
  
outToServer.writeBytes(sentence + '\n');  
  
modifiedSentence = inFromServer.readLine();  
  
System.out.println("FROM SERVER: " + modifiedSentence);  
  
clientSocket.close();  
  
}  
}
```

Threads

- Nebenläufigkeit ist die Fähigkeit eines Systems, zwei oder mehr Aufgaben „gleichzeitig“ auszuführen
- In Java kann die Ausführungsparallelität innerhalb eines Programmes mittels Threads erzwungen werden
- Laufen mehrere Threads parallel, so spricht man auch von Multithreading
- Es gibt zwei Möglichkeiten Threads zu erzeugen.
 1. Erben von der Klasse Thread
 2. Implementieren vom Interface Runnable

Prinzipielle Vorgehensweise:

1. Eine Klasse abgeleitet von Thread erstellen/Runnable implementieren
2. Die Thread-Methode *public void run()* überschreiben
3. Instanzen der Thread-Klasse bilden
4. Die Thread-Instanz(en) mittels *public void start()* starten

Beispiel: Erben von Thread

```
class PrimeThread extends Thread {  
    long minPrime;  
    PrimeThread(long minPrime) {  
        this.minPrime = minPrime;  
    }  
  
    public void run() {  
        // compute primes larger than minPrime  
        . . .  
    }  
}
```

Der folgende Code würde dann den Thread erzeugen und starten:

```
PrimeThread p = new PrimeThread(143);  
p.start();
```

Beispiel 2: Implementieren von Runnable

```
class PrimeRun implements Runnable {  
    long minPrime;  
    PrimeRun(long minPrime) {  
        this.minPrime = minPrime;  
    }  
  
    public void run() {  
        // compute primes larger than minPrime  
        . . .  
    }  
}
```

Erzeugen und Starten des Threads

```
PrimeRun p = new PrimeRun(143);  
new Thread(p).start();
```


Auf das Ende eines Threads warten

void join()

Warte auf das Ende eines Threads

void join(long millis)

void join(long millis, int nanos)

Warte längstens millis (+nanos) auf das Ende eines Threads

Beispiel: Auf das Ende eines Threads warten

```
public class SimpleThread extends Thread
{
    public void run()
    {
        for(int i = 0; i<=1000; i++)
        {
            System.out.println(getState());
        }
    }
}
```

```
public class App
{
    public static void main(String[] args)
    {
        SimpleThread t = new SimpleThread();

        t.start();

        try
        {
            t.join();
        }
        catch (InterruptedException ie)
        {
            // ..
        }

        System.out.println(t.getState());
    }
}
```

Background processing in Android

- Per default läuft der Code im Main-Thread
- Während lange dauernden Berechnungen wird die Applikation blockiert
- Code der GUI sollte getrennt von länger dauernden Operationen (Netzwerk-, Datei- und Datenbankzugriff) ausgeführt werden (asynchron)
- Realisiert wird dies durch Erzeugung eines neuen Threads der die Aufgabe übernimmt und das Ergebnis an den Main Thread zurückgibt.
- Nützliche Klassen/interfaces für Thread Erzeugung in Java: *AsyncTasks*, *Executor*, *ThreadPoolExecutor*
- Umsetzung von Nachrichtenaustausch im lokalen Netzwerk unter Android z.B. mit Wifi-Direkt