Rechnernetze:
(4) Sockets

Prof. Dr. Klaus-Peter Kossakowski

Gliederung der Vorlesung

- Einführung und Historie des Internets
- Schichtenmodell
- Netzwerk als Infrastruktur
- Layer 7: Anwendungsschicht
- Layer 4/7: Socketprogrammierung
- Layer 4: Transportschicht
- Layer 3: Netzwerkschicht
- Layer 2: Sicherungsschicht

Inhalte dieses Kapitels

In diesem Kapitel betrachten wir die Implementierung von Netzwerk-Anwendungen bei der Verwendung der Socket-Schnittstelle (Übergang von Layer 7 auf Layer 4).

Wir konzentrieren uns dabei auf Unterschiede zwischen UDP- und TCP-basierten Anwendungen, da diese im Internet immer noch für die überwiegende Mehrheit aller Anwendungen verwendet werden.

Konkrete Anforderungen der Anwendungen für die Auswahl der Transportprotokolle werden auch behandelt.

Ziele dieses Kapitels

Sie können die Socket-Schnittstelle mit ihrer Bedeutung für die Anwendungsentwicklung für UDP- und TCP-Anwendungen vom Konzept her verwenden, d.h. das Schema der Funktionsaufrufe erklären.

Sie können unterschiedliche Anforderungen für die Netzwerkkommunikation einer Anwendung in Hinblick auf eine Umsetzung über die Socket-Schnittstelle bewerten und entscheiden, ob eine UDP- oder eine TCP-Anwendung besser geeignet ist.

Anwendungen und Anwendungsschicht-Protokolle

Anwendungen:

kommunizierende, verteilte Prozesse

- laufen auf Endsystemen als Benutzermodus-Prozess
- tauschen Nachrichten aus, um eine verteilte Anwendung zu implementieren

Anwendungen und Anwendungsschicht-Protokolle (2)

Anwendungsschicht-Protokolle:

- sind ein Teil einer Anwendung
- definieren Nachrichtenformate und Aktionen
- benutzen Kommunikationsdienste der darunter liegenden Transportschicht
  - TCP, UDP, ...

Netzwerk-Anwendungen: Einordnung

Prozesse: Programme, die auf Hosts laufen

- **innerhalb eines Hosts kommunizieren zwei Prozesse über Interprozess-Kommunikation (IPC)**
- **auf unterschiedlichen Hosts kommunizieren diese über Protokolle der Anwendungs-schicht**

User Agent: Prozess, der mit dem Benutzer und der darunter liegenden Schicht kommuniziert

- **Implementiert das Protokoll der Anwendung**
  - Browser
  - Email-Programm
  - Media-Player

Wie wird die Interprozess-Kommunikation realisiert?

API: Application Programming Interface

- **Definiert eine Schnittstelle zwischen der Anwendungsschicht und der Transport-Schicht (“Service Access Point”)**
- **Internet API: “Socket”**
  - Zwei Prozesse kommunizieren durch Senden von Daten in die Socket und Lesen von Daten aus der Socket

Wie wird die Interprozess-Kommunikation realisiert? (2)

Wie identifiziert ein Anwendungsprozess den Partnerprozess auf dem anderen Rechner?

- IP-Adresse des entfernten Rechners (weltweit eindeutig)
- Portnummer – Information für den empfangenden Rechner, an welchen lokalen Prozess die Nachricht weitergeleitet werden soll.

Welchen Transportdienst braucht die Anwendung ?

Datenverlust

- Einige Anwendungen (z.B. Audio) können einige Verluste tolerieren
- Andere Anwendungen (z.B.: Dateitransfer, remote shell) brauchen 100% zuverlässigen Datentransfer

Zeitanforderungen

- Einige Anwendungen (z.B. Internet Telefonie, interaktive Spiele) brauchen möglichst geringe Verzögerungen, um effektiv zu sein

Welchen Transportdienst braucht die Anwendung ? (2)

Bandbreite / Übertragungsrate

- Wenige Anwendungen (z.B. Multimedia) brauchen garantierte Datenraten, um effektiv zu sein.
- Einige Anwendungen – “elastisch” genannt – nehmen jede Übertragungsrate, die sie bekommen

Anforderungen gängiger Applikationen

Applikation	Verluste	Bandbreite	Zeitkritisch
Dateitransfer	Kein Verlust	Elastisch	nein
Email	Kein Verlust	Elastisch	nein
Web	Kein Verlust	Elastisch	nein
Realtime A/V	Toleranz	Audio: 5kb-1Mb Video: 10kb-5Mb	ja, 100 msec
Stored A/V	Toleranz	Audio: 5kb-1Mb Video: 10kb-5Mb	ja, wenige sec
Interaktive Spiele	Toleranz	> wenige Kb	ja, 100 msec
Finanz-Apps	Kein Verlust	Elastisch	ja und nein

Anforderungen gängiger Applikationen

Applikation	Protokoll	Transport
Dateitransfer	FTP, SCP	TCP
Email	SMTP, POP3, IMAP	TCP
Web	HTTP	TCP
Realtime A/V	RTP	UDP
Stored A/V	RTP	UDP
Dateiserver	NFS, AFS	TCP oder UDP
Internet-Telefonie	SIP	TCP und UDP

Qual der Wahl: TCP oder UDP?

TCP – Dienste:

- Zuverlässig
- Datenstrom
- Reihenfolge erhaltend
- Flusskontrolle durch Empfänger
- Staukontrolle
- Nicht geboten:
  - Garantien über Verzögerung oder Kapazität

UDP - Dienste:

- Unzuverlässig
- einzelne Pakete
- geringer Overhead
- Nicht geboten:
  - Verb.-aufbau
  - Flusskontrolle
  - Staukontrolle
  - Garantien über Verzögerung und Kapazität

Programmierschnittstellen

Zur Programmierung von Kommunikations-kanälen haben sich die sog. Berkeley Sockets etabliert:

- Windows: winsock.dll
- Unix: libsocket.so, <sys/socket.h>
- Java: java.net.*

Die beiden Tripel (Internet-Adresse, Protokoll, Port) von Sender und Empfänger bilden die (eindeutigen) Kommunikationseckpunkte

Programmierschnittstellen (2)

- **Parameter können mit ‚getsockopt' gelesen und mit ‚setsockopt' verändert werden**
- **Können Anwendungsprogramme einfach in einer Netzwerkschnittstelle lesen und schreiben?**
  - Zielvorstellung ist eine Nutzung wie `read` and `write` – nach entsprechendem `open`
  - Allerdings sind Netzwerkschnittstellen komplexer als Dateisysteme
  - Lösung ist die Standardisierung

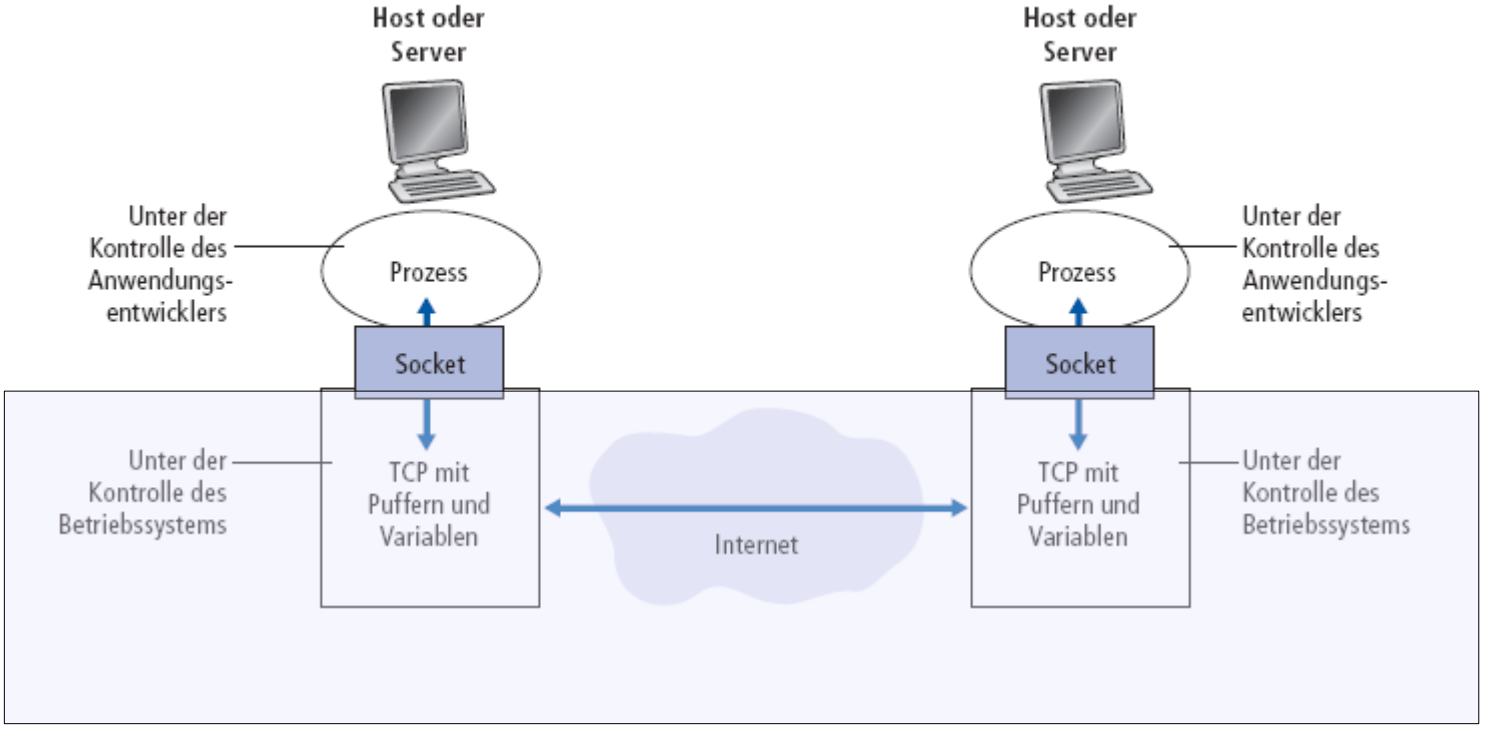
Programmierschnittstellen (3)

- **Netzwerkschnittstelle unterscheiden sich in Details (Protokoll, Port, …) und dem**
- **Kommunikationsparadigma (Client/Server, Message Passing, Broadcast, …)**
- **Systemübergreifende Realisierung ist erforderlich**
  - Vielfältige Betriebssysteme
  - Enkodierungen und
  - Programmiersprachen

Sockets

- **Ursprünglich die Netzwerk-API von BSD 4.3 (Unix)**
- **Seit Jahren am meisten verbreiteter Programmierstandard (C/C++, Java, …)**
- **Drei Typen:**
  - Stream (SOCK_STREAM) für TCP
  - Datagram (SOCK_DGRAM) für UDP
  - Raw (SOCK_RAW) für IP & ICMP
- **Erstellung in C/C++:**
  - s = socket(domain, type, protocol)

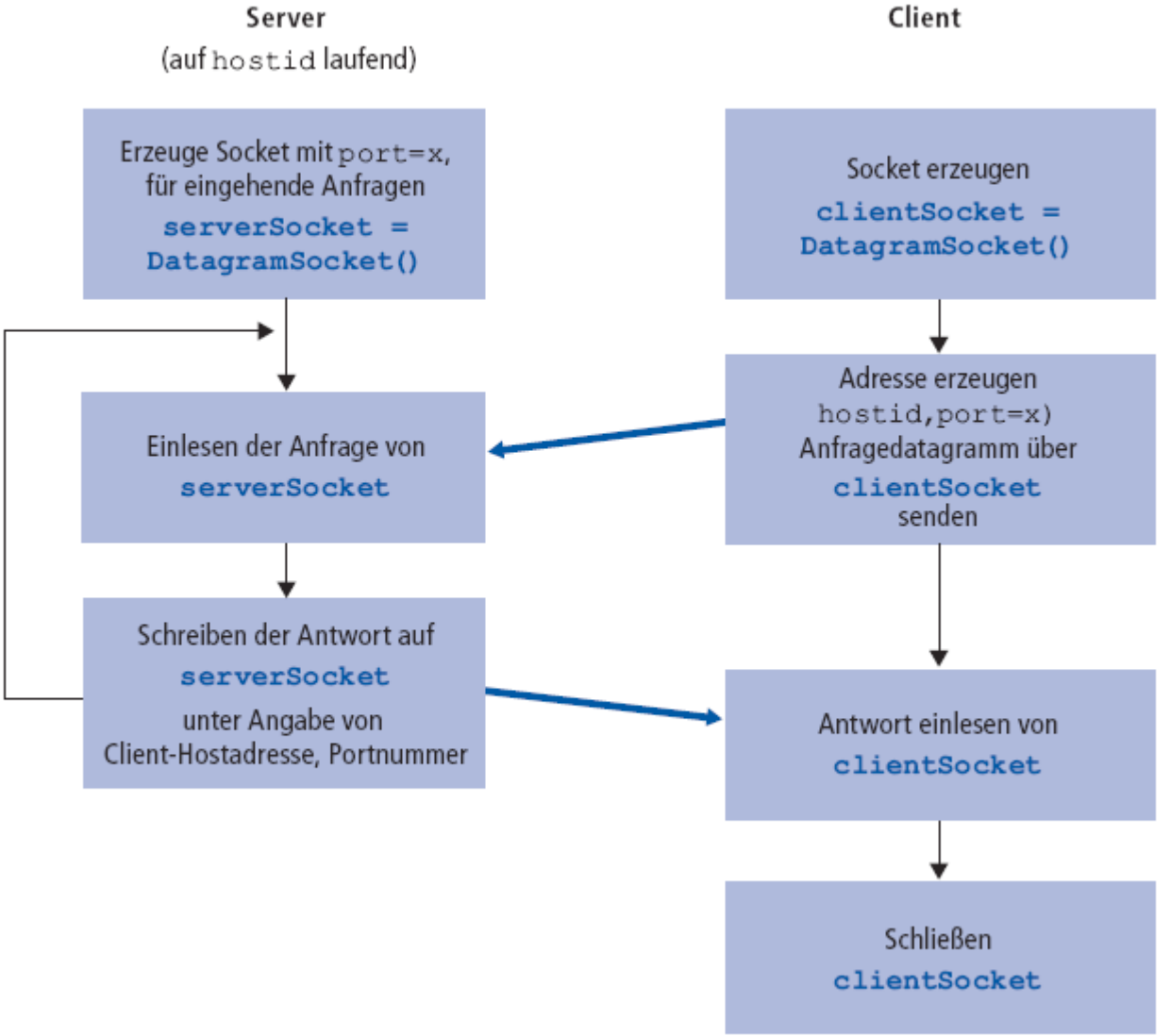
Sockets als Schnittstelle zwischen Anwendung und Betriebssystem



JAVA:
Socket Programmierung mit UDP

UDP stellt eine unzuverlässige Paket-Übertragung zwischen Client und Server bereit. Dazu muss:

- **Der Server-Prozess muss als Erster laufen**
  - und ein Socket erzeugt haben, das auf eingehende Pakete reagieren kann
- **Client muss etwas senden:**
  - Erzeugung einer UDP-Socket und dabei Angabe von IP-Adresse und Port-Nummer
  - Jedes Paket muss diese Angaben haben


Server
(auf hostid laufend)
Client
Erzeuge Socket mit port=x, für eingehende Anfragen
serverSocket = DatagramSocket()
Socket erzeugen
clientSocket = DatagramSocket()
Einlesen der Anfrage von
serverSocket
Adresse erzeugen
hostid,port=x)
Anfragedatagramm über
clientSocket
senden
Schreiben der Antwort auf
serverSocket
unter Angabe von
Client-Hostadresse, Portnummer
Antwort einlesen von
clientSocket
Schließen
clientSocket

JAVA: Socket-Methoden (UDP)

- **public DatagramSocket ()**
  - Erzeugt eine UDP-Socket und bindet diese an irgendeinen freien Port zum Senden
- **public DatagramSocket (int port)**
  - Erzeugt eine UDP-Socket und bindet diese an den angegebenen Port zum Empfang
- **public void send (DatagramPacket pkt)**
  - Sendet ein Packet
- **public void receive (DatagramPacket pkt)**
  - Empfängt ein Paket, blockiert bis zum Empfang.
- **public void close()**

JAVA:
Socket-Methoden (UDP)

- **public DatagramPacket (byte[ ] buf, int length, InetAddress destAddress, int destPort)**
  - Erzeugt ein Paket zum Versenden
- **public DatagramPacket (byte[ ] buf, int length)**
  - Erzeugt einen Paket-Puffer zum Empfang
- **public InetAddress getAddress()**
  - Gibt die IP-Adresse zurück, sowohl beim Empfang als auch beim Versenden

JAVA:
Socket-Methoden (UDP)

- **public int getPort()**
  - Gibt die Port-Nummer zurück, von dem das Paket empfangen oder an den es gesendet wurde
- **public byte[ ] getData()**
  - Gibt den Datenpuffer zurück

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

public class UDPClient{
    public static void main(String args[]) throws Exception {
      try {
        DatagramSocket aSocket = new DatagramSocket();
        byte [] m = args[0].getBytes();
        InetAddress aHost = InetAddress.getByName(args[1]);
        int serverPort = 6789;

        DatagramPacket request = new DatagramPacket(m, args[0].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()));
                aSocket.close();
      } catch (SocketException e) {
        System.out.println("Socket: " + e.getMessage());
      } catch (IOException e) {
        System.out.println("IO: " + e.getMessage());}
      }
    }
}
```

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

public class UDPServer{
    public static void main(String args[]) throws Exception {
      try{
        DatagramSocket 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()); }
    }
}
```

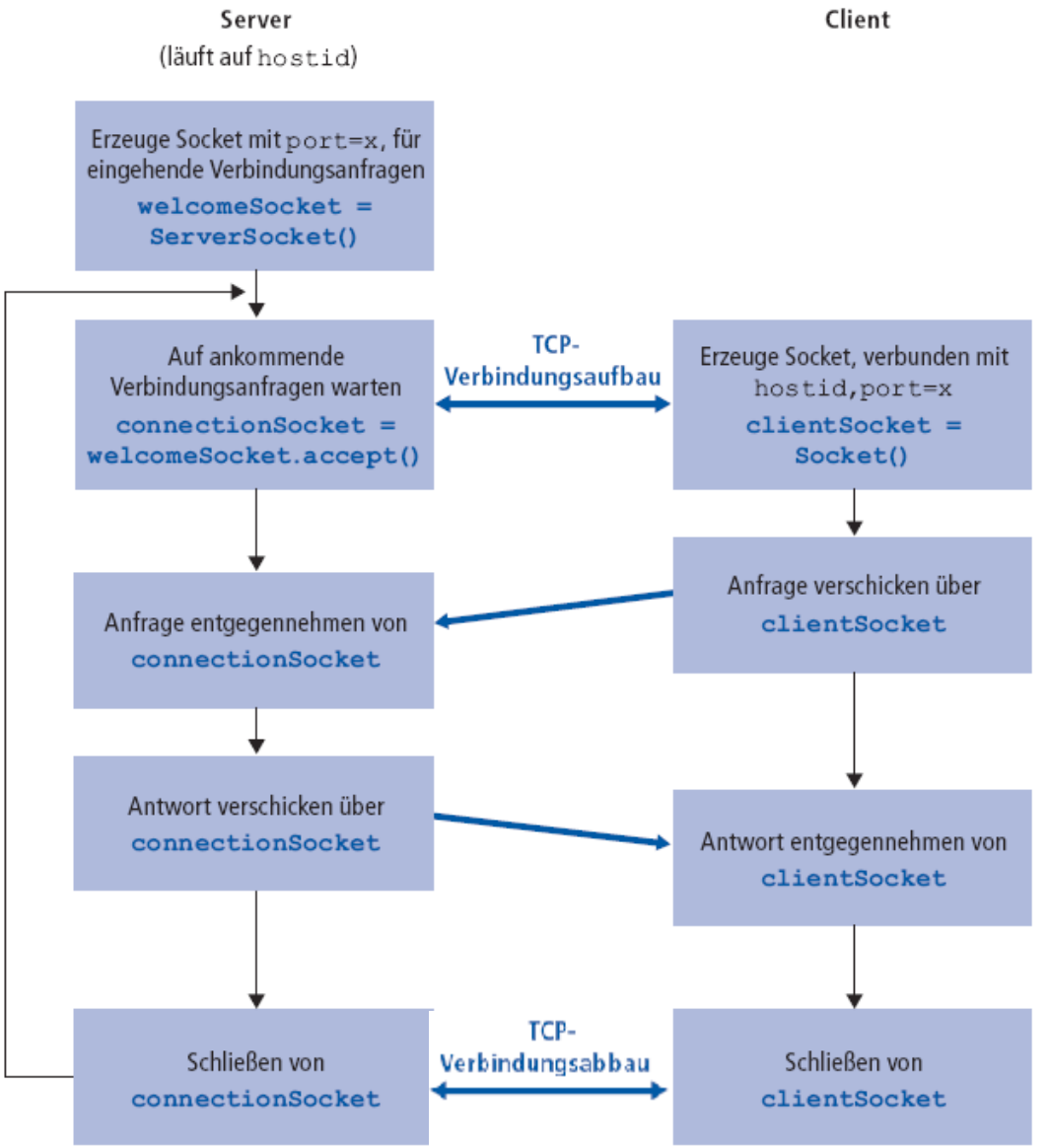
Socket-Programmierung mit TCP

TCP stellt zuverlässige, die Reihenfolge erhaltende Byte-Übertragung (“pipe”) zwischen Client und Server bereit. Dazu muss:

- **Der Server-Prozess als Erstes laufen**
  - und ein Socket erzeugt haben, das den Client-Kontakt bemerken kann
- **Der Client den Server kontaktieren:**
  - Erzeugung einer TCP-Socket und dabei Angabe von IP-Adresse und Port-Nummer
  - TCP-Three-Way-Handshake (später mehr)

Socket Programmierung mit TCP

- **Server nimmt Anfrage entgegen**
  - Es wird eine neue “Arbeits”-Socket generiert, mit der danach die eigentliche Kommunikation erfolgt
  - Dies erlaubt dem Server, gleichzeitig mit vielen Clients zu kommunizieren, d.h. weitere Anfragen entgegen zu nehmen


Server
(läuft auf hostid)
Client
Erzeuge Socket mit port=x, für eingehende Verbindungsanfragen
welcomeSocket = ServerSocket()
Auf ankommende Verbindungsanfragen warten
connectionSocket = welcomeSocket.accept()
TCP-Verbindungsaufbau
Erzeuge Socket, verbunden mit hostid,port=x
clientSocket = Socket()
Anfrage verschicken über
clientSocket
Anfrage entgegennehmen von
connectionSocket
Antwort verschicken über
connectionSocket
Antwort entgegennehmen von
clientSocket
Schließen von
connectionSocket
TCP-Verbindungsabbau
Schließen von
clientSocket

JAVA:
Socket-Methoden (TCP)

- **public Socket (String host, int port)**
  - Erzeugt ein TCP-Socket und bindet diesen an den angegebenen Rechner/Port
- **public Socket (InetAddress addr, int port)**
  - Erzeugt ein TCP-Socket und bindet diesen an die angegebene IP-Adresse/Port
- **public InputStream getInputStream()**
  - Liest von der TCP-Socket
- **public OutputStream getOutputStream()**
  - Schreibt an die TCP-Socket
- **public void close()**

JAVA: ServerSocket-Methoden (TCP)

- **public ServerSocket (int port)**
  - Realisiert eine TCP-Server-Socket für die angegebene Portnummer
- **public Socket accept()**
  - Wartet auf eine Verbindung an die zuvor etablierte TCP-Server-Socket und akzeptiert diese.
  - Die Methode „blockiert“, bis eine Verbindung akzeptiert ist, mit neuer Socket.
- **public void close()**
  - Das Schließen der TCP-Server-Socket

Funktionen in C

```
int socket (int af, int type, int protocol);
int bind (int s, const struct sockaddr *addr,
            socklen_t addrlen);
int listen(int s, int backlog);
int accept(int s, struct sockaddr *addr,
            socklen_t *addrlen);
int connect(int  s, const struct sockaddr *serv_addr,
            socklen_t addrlen);
ssize_t send(int s, const void *buf, size_t len,
            int flags);
ssize_t recv(int s, void *buf, size_t len,
            int flags);
int close(int s);
int set/getsockopt(int s, int  level, int optname,
            const void *optval, socklen_t optlen);
```

Programmbeispiele in C

… z.B. in den Unterlagen von Prof. Thomas C. Schmidt:

http://inet.cpt.haw-hamburg.de/teaching/ss-2011/rechnernetze/socket_client.c/view

http://inet.cpt.haw-hamburg.de/teaching/ss-2011/rechnernetze/socket_srv.c/view

Kontakt

Prof. Dr. Klaus-Peter Kossakowski

Email: klaus-peter.kossakowski@haw-hamburg.de

Mobil: +49 171 5767010

http://users.informatik.haw-hamburg.de/~kpk/