Transportprotokolle

1. Protocol-Port Konzept
2. Socket Programmierung
3. User Datagram Protocol (UDP)
4. Transmission Control Protocol (TCP)
   1. Verbindungsmanagement
   2. Sicherung
   3. Flußkontrolle, Staukontrolle
   4. Optimierungen
5. Neuere Entwicklungen

Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Zum Inhalt

In diesem Abschnitt werden wir uns den Transport-protokollen im Internet widmen. Ihre Wahl und Eigenschaften werden direkt aus der Anwendungs-entwicklung heraus angesprochen. Deshalb spielt die Programmierung des „Socket"-Interfaces hier eine besondere Rolle.

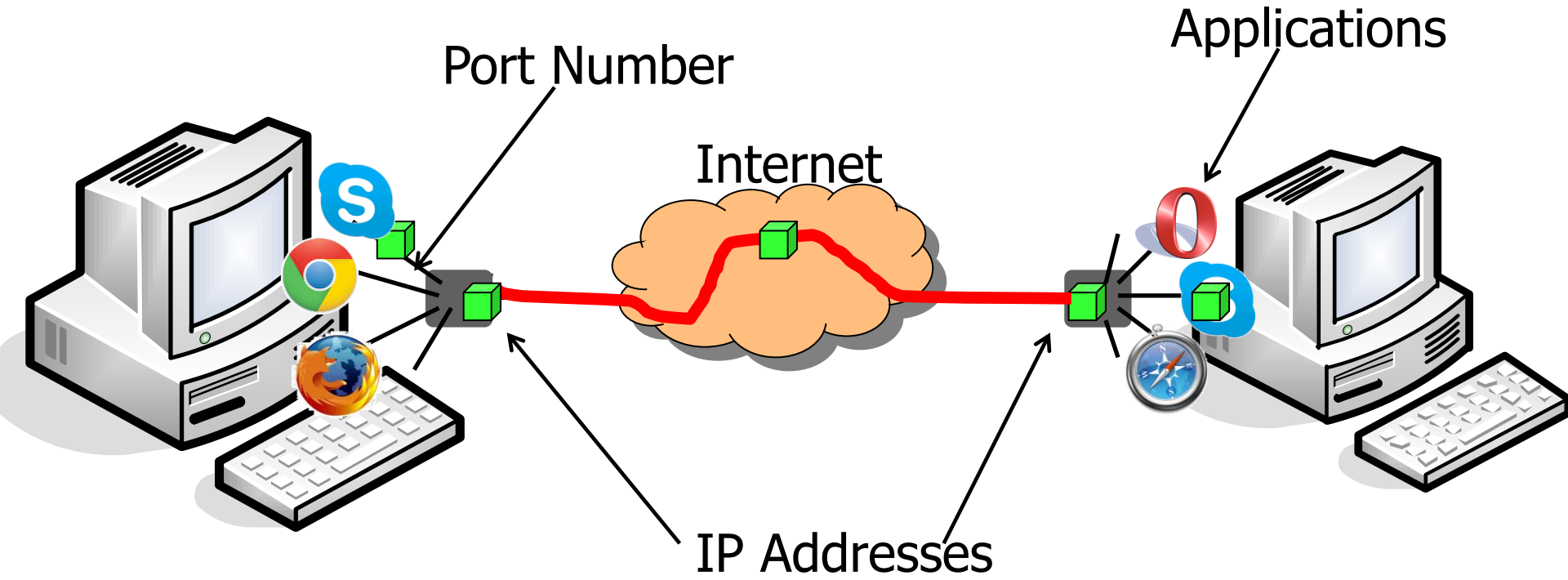
Das zugehörige Kapitel im Tanenbaum ist 6, Bonaventure finden Sie die Inhalte in Teil 2 und die Socket-Programmierung in Teil 3 – bitte Tanenbaum hierfür überspringen.

1. Protokoll-Port-Konzept

Wie werden Kommunikationsprozesse auf einem Rechner identifiziert?

- Direkte Prozessadressierung ist problematisch:
  - Prozesslogik ist betriebssystemabhängig
  - Ziel der Adressierung ist ein Dienst, nicht der Prozeß
  - Prozesse können mehr als einen Dienst anbieten
- Lösung: Verwendung von abstrakten Protokoll Ports
  - Betriebssystem stellt Spezifikation und Zugriff
  - Protokoll-Software (stack) synchronisiert den Zugriff mittels Warteschlangen und Belegungen

1. Multiplexen und Demultiplexen



- IP stellt den Pfad von Quell- zu Zielrechner bereit
- Die Transportschicht multiplext lokal mithilfe der Ports

1. Netzwerkports

```
atlantis %>netstat -a (Auszug)

TCP
   Local Address         Remote Address    Swind Send-Q Rwind Recv-Q  State
-------------------- -------------------- ----- ------ ----- ------ -------
      *.*                  *.*                0      0     0      0 IDLE
      *.sunrpc             *.*                0      0     0      0 LISTEN
      *.*                  *.*                0      0     0      0 IDLE
      *.32771              *.*                0      0     0      0 LISTEN
      *.ftp                *.*                0      0     0      0 LISTEN
      *.telnet             *.*                0      0     0      0 LISTEN
      *.shell              *.*                0      0     0      0 LISTEN
      *.login              *.*                0      0     0      0 LISTEN
      *.exec               *.*                0      0     0      0 LISTEN
      *.uucp               *.*                0      0     0      0 LISTEN
      *.finger             *.*                0      0     0      0 LISTEN
      *.time               *.*                0      0     0      0 LISTEN
      *.x-server           *.*                0      0     0      0 LISTEN
   sol01.rz.fhtw-berlin.de.1021 amor.rz.fhtw-berlin.de.nfsd  8760      0  8760
                                                                    0 ESTABLISHED
localhost.32785      localhost.32783      32768      0 32768   0 ESTABLISHED
localhost.32783      localhost.32785      32768      0 32768   0 ESTABLISHED
   sol01.rz.fhtw-berlin.de.1023 merlin.rz.fhtw-berlin.de.login 49152   0  8760
                                                                    0 ESTABLISHED
   sol01.rz.fhtw-berlin.de.x-server merlin.rz.fhtw-berlin.de.25478 49152     0
                                                                 8760      0 ESTABLISHED
```

1. Zentrale Portvergabe

Zur Kommunikation miteinander müssen sich zwei Rechner auf Portnummern einigen. Hierfür gibt es eine

- Zentrale Vergabe - universal binding to well known ports: System (0...1023) und User (1024...49151)

  Bsp:

Service	Portnummer	Protokoll
ftp-data	20	tcp
ftp	21	tcp
telnet	23	tcp
smtp	25	tcp
tftp	69	udp
finger	79	tcp
portmap	111	tcp
portmap	111	udp

- Die Festlegung befindet sich in der Datei /etc/services

1. Portnutzung

System Ports (< 1024) sind privilegiert (geschützt durch Systemrechte)

User Ports (1024 – 49151) werden von Anwendern registriert

- IANA koordiniert beide in ihrer Port Registry: https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml

Dynamische Zuordnung - (dynamic binding)

- Wird Anwendungsprogrammen vom System zugeteilt
- Portnummern sollten > 49151 sein

1. Transportprotokolle

- IP adressiert nur Zielrechner, nicht einzelne Programme
- Damit Anwendungsprogramme Datagramme senden und empfangen können, stehen traditionell UDP und TCP als Transportprotokolle bereit (Layer 4)
- Beide verwenden das Prinzip der abstrakten Ports
- Beide können via Sockets programmiert werden

2. Programmierschnittstelle

Als Programmierschnittstelle für Software mit 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

Kommunikationsparameter können mit ‚getsockopt' gelesen und mit ‚setsockopt' verändert werden

2. Applikationsprogramm-Interface (API)

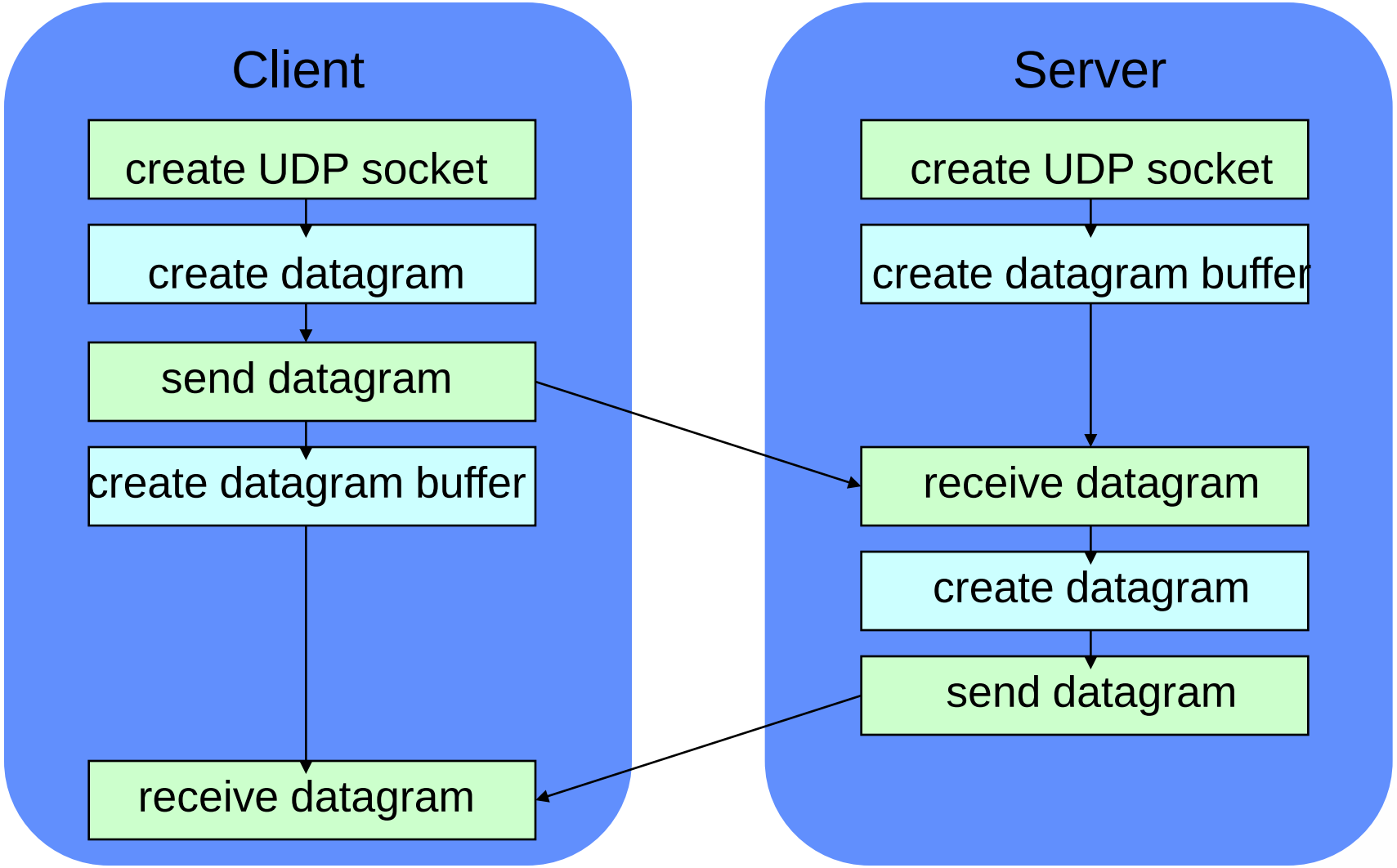
Wie können Anwendungsprogramme einfach in einer Netzwerkschnittstelle lesen und schreiben?

- Zielstellung: Nutzung wie `read` and `write`
- Probleme:
  - Netzwerkschnittstelle komplexer als Filesystem (Protokolle, -arten, Sicherheit)
  - Andersartige Kommunikationsparadigmen (C/S, Message Passing, Broadcast,…)
  - Systemübergreifende Realisierung (Vielfältige OSes, Enkodierungen & Programmiersprachen)
- Lösung: Standardisierte Netzwerk-API zur Bedienung von Standardprotokollen - Berkeley Sockets und SystemV Transport Layer Interface (TLI)

2. Sockets

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

2. Verbindungslose Kommunikation: Struktur eines UDP-Programms



2. Beispiel: UDP-Client in Java

```
import java.net.*;
import java.io.*;
public class UDPClient{
    public static void main(String args[]){
        // args give message contents and server hostname
          try {
                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());}}
    }
}
```

Sendet Nachricht an Server

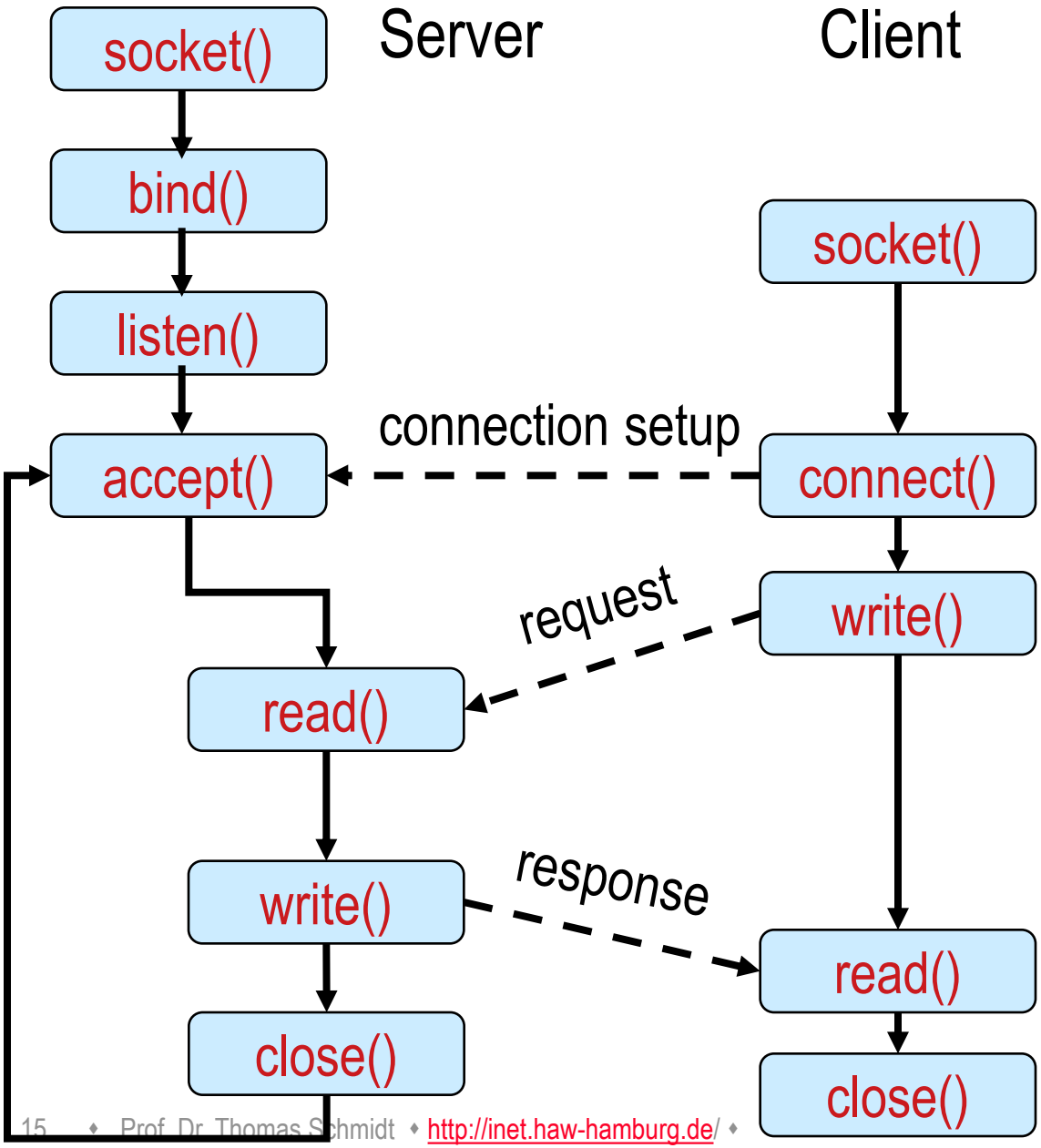
Empfängt Antwort vom Server

2. Beispiel: UDP-Server in Java

```
import java.net.*;
import java.io.*;
public class UDPServer{
        public static void main(String args[]){
            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());}
    }
}
```

Empfängt Nachricht von Client

Sendet Antwort an Client



Verbindungs-orientierte Kommunikation: Call-Sequenz für TCP

2. Wichtige Funktionen (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);

2. IPv4 → IPv6 Koexistenz – RCF 3493: Versionsübergreifende Adress-API

	IPv4	IPv6	
Data structures	AF_INET	AF_INET6	
	in_addr sockaddr_in	in6_addr sockaddr_in6	
Address conversion functions	inet_aton() inet_addr()	inet_pton()	IPv4 and IPv6 functions
	inet_ntoa()	inet_ntop()	
Name-to-address functions	gethostbyname() gethostbyaddr()	getnameinfo() * getaddrinfo() *	

*** POSIX protocol independant functions**

2. Benutzung der Koexistenz-API

Problem: `sockaddr_in` und `sockaddr_in6` sind inkompatible Datenstrukturen ...

Lösung:

- Eine Indirektionsstruktur `addrinfo` erlaubt den transparenten Zugriff auf die (versionsabhängigen) `sockaddr*` Strukturen
- Diese werden automatisch gefüllt durch `getaddrinfo`: liefert im `result` einen Pointer auf eine verkettete Liste von `addrinfo` Adressstrukturen.
- Um einen erfolgreichen Kommunikationsweg zu finden, müssen die Adressen der Liste ausprobiert werden.
- Zusätzlich ist `sockaddr_storage` eine versionsübergreifende Datenstruktur, die gem. Versionen gecasted werden kann.

2. Versionsneutrale Programmierung

Client Seite:

- Der Client muss eine IP-Version wählen, die der Server versteht. Hierzu nutzt er den DNS (`getaddrinfo`), ggfs. Anwendereingaben bzw. probiert die Versionen durch.
- In der Regel wählt der Client also zwischen mehreren Adressoptionen des Servers zur Kommunikationsaufnahme.

Server Seite:

- Der Server muss in allen IP-Versionen ansprechbar sein. Hierfür benötigt er eine Adressstruktur, die für alle Adressen geeignet ist (`sockaddr_storage`). Gegenwärtig ist diese identisch mit `sockaddr_in6` und IPv4 wird eingebettet.
- Der Server antwortet in derselben Version, in der er angesprochen wurde.

2. Programmbeispiele

In der Vorlesung

client.c **server.c**

2. Für richtige Programmierer

- Namen/Adressen werden dynamisch gesetzt – im Programm stehen natürlich keine ‚Magic Numbers'
- Server-Implementierungen multiplexen normalerweise mehrere Verbindungen
  - Das geht ohne Threads im *Reactive Pattern*
  - In Posix mit `Select()` - wählt Daten-Events aus Descriptor-Menge: `fd_set`
  - Performanter ist `epoll()` unter Linux
- UDP multiplext Pakete unterschiedlicher Sender. Anwendungen unterscheiden Sender mit `recvfrom()`

2. Wichtige Funktionen (JAVA)

Socket (Hostname/InetAddress, Port) / ServerSocket (Port) / DatagramSocket (Port)

mit den Methoden:

- bind
- connect/accept
- close
- getInputStream/getOutputStream