Universität Bamberg

GRABS: Grundlagen der Rechnerarchitektur und Betriebssysteme

Vorlesung 17: Kommunikation

Interprozesskommunikation und Netzwerke

Michael Engel (michael.engel@uni-bamberg.de)
Lehrstuhl für Praktische Informatik, insbes. Systemnahe Programmierung
https://www.uni-bamberg.de/sysnap

Licensed under CC BY-SA 4.0
unless noted otherwise

Prozesse nochmal…

Universität Bamberg

- Prozesse sind normalerweise ***isoliert*** voneinander
  - Eigene Adressräume, getrennt mittels virtuellem Speicher
- …können bei Bedarf auch miteinander ***interagieren***
  - …auf einen Prozess warten (*Synchronisation*)
  - …Daten austauschen (*Kommunikation*)
- Wartemechanismen...
  - werden für *kontrollierte Kommunikation* benötigt
  - können *Deadlocks* verursachen
- Wie sieht es mit der Datenübertragung aus?
  - Threads bzw. Fibers, die sich einen Adressraum teilen

Inter-Prozess-Kommunikation

- ***Inter-Prozess-Kommunikation (Inter-Process Communication, IPC)***
  - Mehrere Prozesse *bearbeiten* eine Aufgabe *gemeinsam*
  - Gleichzeitige Nutzung von Informationen durch mehrere Prozesse
  - *Reduktion* von Rechenzeit durch *Parallelisierung*
  - *Verbergen* von Rechenzeit durch "Hintergrundausführung"
- Kommunikation über ***Nachrichten (messages)***
  - Nachrichten werden zwischen Prozessen ausgetauscht
  - Kein geteilter Speicher erforderlich
- Kommunikation über ***geteilten Speicher (shared memory)***
  - Datenaustausch durch nebenläufige Schreibvorgänge in und Lesevorgänge aus einem gemeinsamen (geteilten) Speicherbereich
  - Synchronisation ist hier wichtig!

Nachrichtenbasierte Kommunikation

- Basiert auf zwei Funktionen ("Primitiven"):

```
send (ziel, nachricht)
nachricht = receive (quelle)
```

(oder ähnlich)

- Implementierungen unterscheiden sich durch
  - Synchronisation
  - Adressierung
  - und möglicherweise in anderen Eigenschaften

Synchronisation

... für nachrichtenorientierte Kommunikation

- Synchronisation beim Senden / Empfangen
  - **Synchroner Nachrichtenaustausch** ("**Rendezvous**")
    - Empfänger blockiert, bis die Nachricht angekommen ist
    - Sender blockiert, bis der Empfang der Nachricht bestätigt wurde
  - **Asynchroner Nachrichtenaustausch**
    - Sender übergibt Nachricht an das Betriebssystem und läuft weiter
    - Blockieren ist auf Sender- und Empfängerseite optional
    - Benötigt immer *Zwischenspeicherung (buffering)*

- Oft implementiert:
  - Asynchroner Nachrichtenaustausch mit möglicherweise blockierenden Sende- und Empfangsoperationen

Adressierung

... für nachrichtenorientierte Kommunikation

- **Direkte Adressierung**
  - Adresse besteht aus
    - Prozess-ID (Signale)
    - Kommunikationsendpunkt eines Prozesses (*Port* oder *Socket*)

- **Indirekte Adressierung**
  - Kanäle (*channels*, *pipes*)
  - Mailboxes und Nachrichtenwarteschlangen (*message queues*)

- Zusätzliche Dimension: **Gruppenadressierung**
  - unicast – Senden an genau einen Empfänger
  - multicast – Senden an eine Teilmenge möglicher Empfänger
  - broadcast – Senden an alle Empfänger

Zusätzliche Eigenschaften

... der nachrichtenbasierten Kommunikation

- **Nachrichtenformat**
  - Datenstromorientiert / Nachrichtenorientiert
  - Feste Länge / variable Länge
  - Typisiert / Untypisiert (Bytestrom)
- **Übertragung**
  - unidirektional / bidirektional (*half duplex*, *full duplex*)
  - zuverlässig / unzuverlässig
  - Erhalt der Reihenfolge / keine Erhaltung

Lokale IPC: Unix-Signale

- Signale sind *Interrupts, die in Software implementiert sind*
  - ähnlich zu Hardware-I/O-Interrupts von Geräten
  - minimale Form der Interprozesskommunikation
    (es wird nur eine *Signalnummer* übertragen)

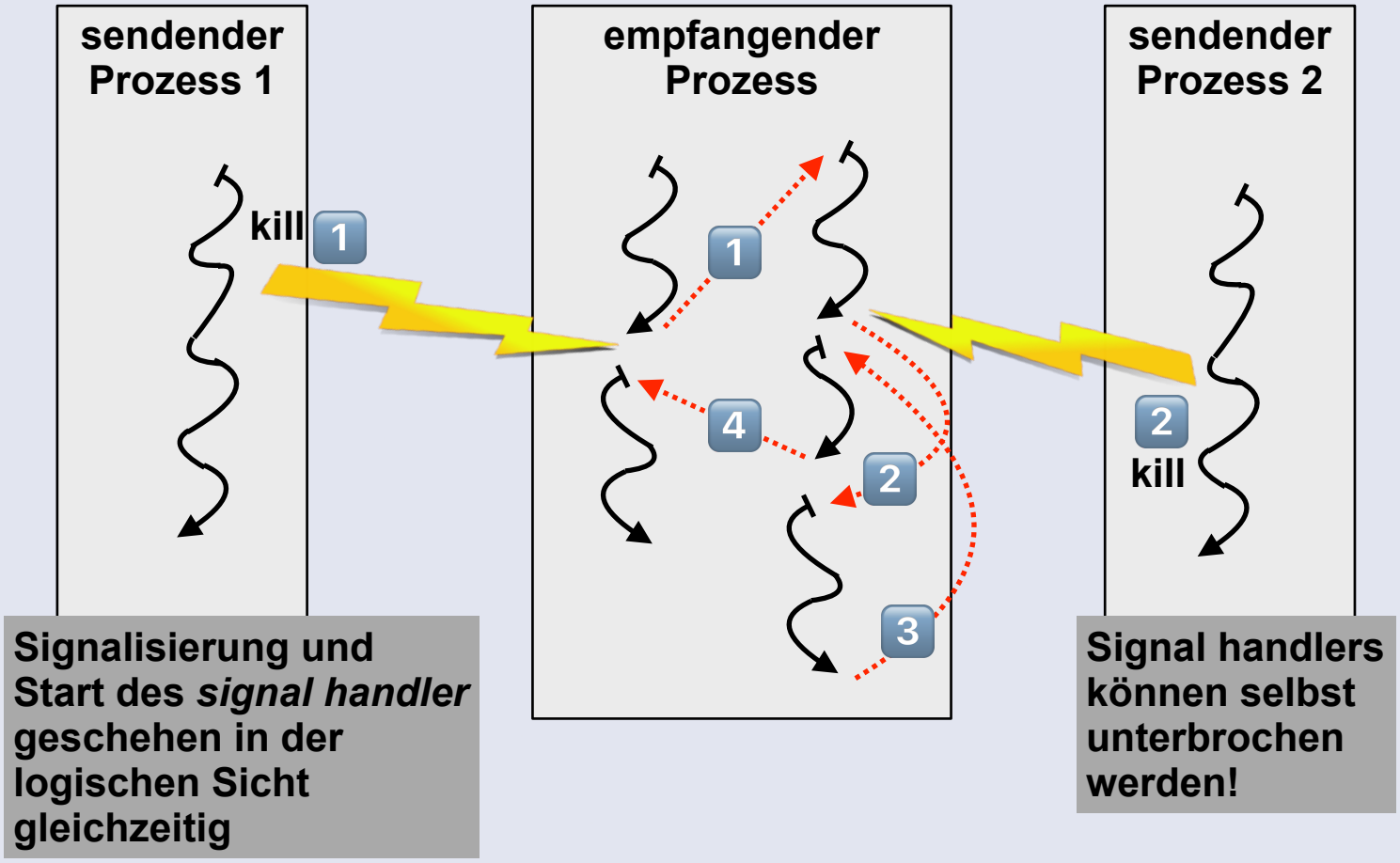
- Sender:
  - *Prozesse:* nutzen den Systemaufruf `kill(2)` zum Versenden
  - *Betriebssystem:* sendet Signale, wenn bestimmte Ereignisse auftreten
- Der empfangende Prozess behandelt Signale durch:
  - Ignorieren,
  - Terminierung des Prozesses oder
  - Ausführung einer *Signalbehandlungsfunktion (signal handler)*
    - nach Behandlung des Signals kann der Prozess an der Stelle fortfahren, an der er unterbrochen wurde

Signale

- Durch Signale können Prozesse über *Ausnahmesituationen* informiert werden
  - ähnlich zu Exceptions in Hardware
- Beispiele:
  - `SIGINT` Terminiere den Prozess (Ctrl-C)
  - `SIGSTOP` Suspendiere den Prozess (Ctrl-Z)
  - `SIGWINCH` Fenstergröße hat sich geändert
  - `SIGCHLD` Kindprozess ist terminiert
  - `SIGSEGV` Ungültiger/nicht erlaubter Speicherzugriff
  - `SIGKILL` Prozess wird (alternativlos) terminiert
- Die meisten Signale haben einen *Standardhandler,* z.B. für Terminierung (`SIGINT`) oder Suspendierung (`SIGSTOP`)
  - Dieser kann für die meisten Signale umdefiniert werden
  - Details in der man page zu `signal(2)`

Unix Signale: logische Sicht

- Hollywood-Prinzip: "***don't call us, we'll call you***."



Unix-Signale: Implementierungssicht

- Signalbehandlung erfolgt immer dann, wenn die ***Programmausführung vom Kernel- zum Benutzermodus zurückkehrt*** (z.B., wenn ein Prozess von einem Systemaufruf zurückkehrt)
- Was geschieht, wenn ein empfangender Prozess…
  - gerade läuft (Prozesszustand `RUNNING`)?
    - Signale können z.B. durch Speicherzugriffsfehler erzeugt werden
    - Signal handler wird sofort gestartet
  - gerade nicht läuft, aber `READY` ist (z.B. `kill`-Syscall von anderem Prozess)?
    - Das Signal wird im Prozesskontrollblock des Empfängers notiert
    - Wenn der Prozess wieder die CPU zugeteilt bekommt, wird das Signal behandelt
  - auf I/O wartet (Zustand `BLOCKED`)?
    - Der I/O-Syscall (z.B. read) wird *unterbrochen* und kehrt schließlich mit einem Fehlercode `EINTR` (Interrupted) zurück
    - Der Prozesszustand wird auf `READY` gesetzt
    - Bei Bedarf wird der unterbrochene System call wiederholt (`SA_RESTART`)

Unix Signale: Beispiel

- Welche Aktionen können bei Empfang eines Signals ausgelöst werden?
  - Auszug aus dem Handbuch des Apache Webservers

Stopping and Restarting Apache

To send a signal to the parent you should issue a command such as:

```
kill -TERM `cat /usr/local/apache/logs/httpd.pid`
```

TERM Signal: stop now

Sending the TERM signal to the parent causes it to immediately attempt to kill off all of its children. It may take it several seconds to complete killing off its children. Then the parent itself exits. Any requests in progress are terminated, and no further requests are served.

HUP Signal: restart now

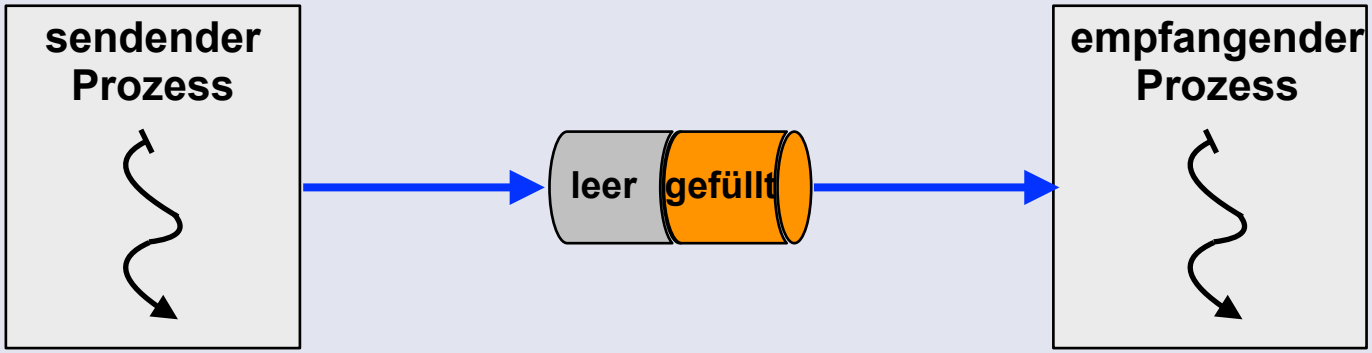
Sending the HUP signal to the parent causes it to kill off its children like in TERM but the parent doesn't exit. It re-reads its configuration files, and re-opens any log files. Then it spawns a new set of children and continues serving hits.

USR1 Signal: graceful restart

The USR1 signal causes the parent process to advise the children to exit after their current request (or to exit immediately if they're not serving anything). The parent re-reads its configuration files and re-opens its log files. As each child dies off the parent replaces it with a child from the new generation of the configuration, which begins serving new requests immediately.

Unix-Pipes

- *Kanal* zwischen zwei kommunizierenden Prozessen
  - Unidirektional
  - Gepuffert (feste Puffergröße)
  - Zuverlässige Übertragung
  - Datenstrom-orientiert
- Operationen: **Lesen** (read) und **Schreiben** (write)
  - Reihenfolge gesendeter Zeichen wird beibehalten (**Zeichenstrom**)
  - Blockiert, wenn die Pipe voll ist (beim Schreiben) bzw. leer (Lesen) ist



Programmieren mit Unix-Pipes

- **Namenlose Pipes**
  - Erzeugt mit Syscall `int pipe (int fdes[2])`
  - Nach erfolgreichem Aufruf (Rückgabewert == 0) kann man...
    - aus der Pipe **lesen** von `fdes[0]` (Syscall `read`)
    - in die Pipe **schreiben** über `fdes[1]` (Syscall `write`)
  - danach muss man ein “Ende” der Pipe einem anderen Prozess mitteilen
    - …auf der nächsten Folie
- **Benannte Pipes**
  - Pipes können auch als *special file* (Dateiname im “path”-Parameter) im Dateisystem angelegt werden:
    `int mkfifo (const char *path, mode_t mode)`
  - Nach Aufruf von `mkfifo` verwendet man open/read/write/close wie bei gewöhnlichen Dateien
  - Die Unix-Zugriffsmechanismen funktionieren auch für Pipes

Programmieren mit Unix-Pipes

connect.c

```c
enum { READ=0, WRITE=1 };                /* Indizes in fd-Array */

int main (int argc, char *argv[]) {
  int res, fd[2];
  if (pipe (fd) == 0) {                  /* Erzeuge die Pipe */
    res = fork ();
    if (res > 0) {                       /* Elternprozess */
      close (fd[READ]);                  /* Schließe Leseseite */
      dup2 (fd[WRITE], 1);               /* Leite stdout an Pipe */
      close (fd[WRITE]);                 /* Deskriptor freigeben */
      execlp (argv[1], argv[1], NULL); /* Erzeuger ausführen */
    }
    else if (res == 0) {                 /* Kindprozess */
      close (fd[WRITE]);                 /* Schließe Schreibseite */
      dup2 (fd[READ], 0);                /* Leite stdin an Pipe */
      close (fd[READ]);                  /* Deskriptor freigeben */
      execlp (argv[2], argv[2], NULL); /* Verbraucher ausführen */
    }
  }
  /* …hier käme die Fehlerbehandlung… */
}
```

Programmieren mit Unix-Pipes

“./connect ls wc” entspricht dem Shellbefehl “ls | wc”

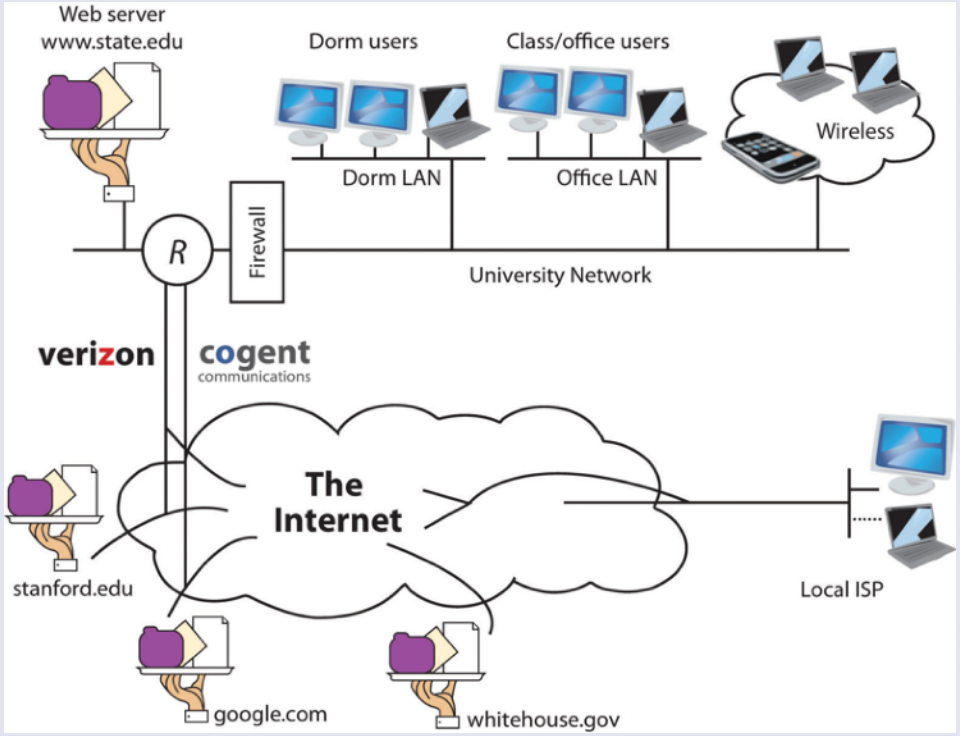
```
enum { READ=0, WRITE=1 };                    /*

int main (int argc, char *argv[]) {
  int res, fd[2];
  if (pipe (fd) == 0) {                      /* Erzeuge die Pipe */
    res = fork ();
    if (res > 0) {                           /* Elternprozess */
      close (fd[READ]);                      /* Schließe Leseseite */
      close (fd[WRITE]);                     /* Schließe Schreibseite */
      dup2 (fd[READ], 0);                    /* Leite stdin an Pipe */
      close (fd[READ]);                      /* Deskriptor freigeben */
      execlp (argv[2], argv[2], NULL); /* Verbraucher ausführen */
    }
  }
  /* …hier käme die Fehlerbehandlung… */
}
```

```
$ ls
connect connect.c execl.c fork.c orphan.c wait.c
$ ./connect ls wc
      6       6      49
```

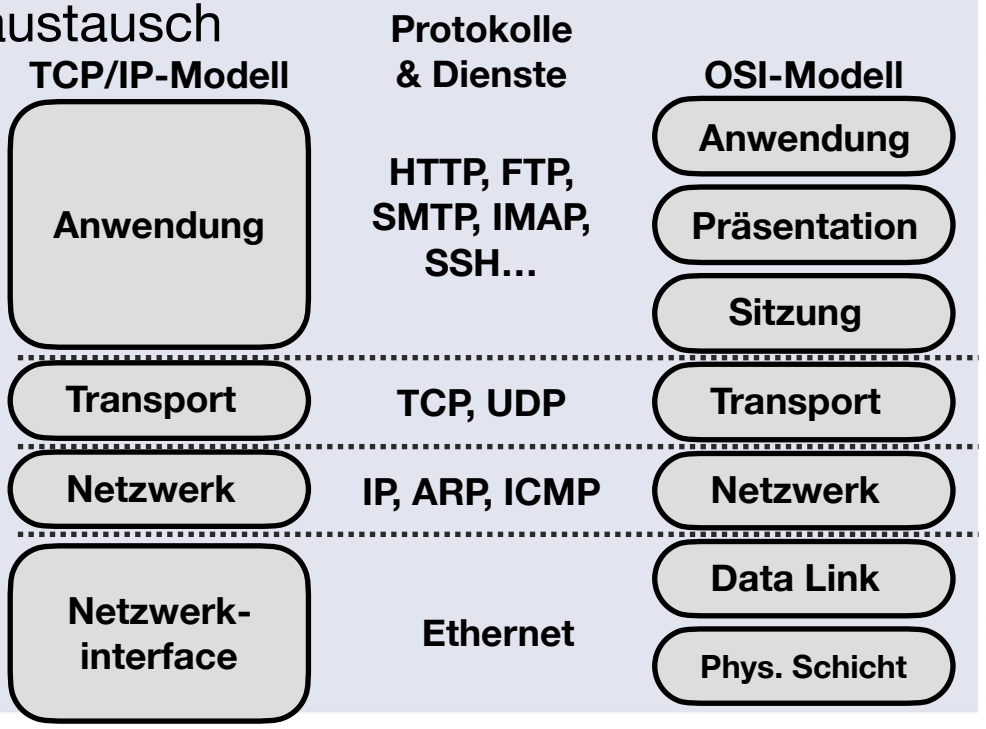
Kommunikation zwischen Computern

- Kommunikation zwischen Prozessen auf einem einzelnen Computer kann über **geteilten Speicher** stattfinden
  - Wie können Prozesse auf unterschiedlichen Rechnern kommunizieren? → **Netzwerk**
- Computer innerhalb einer Organisation (z.B. Uni Bamberg) sind über ein **lokales Netzwerk** (verdrahtet oder drahtlos) miteinander verbunden
  - Die Verbindung vieler lokaler Netze erzeugt ein (oder das) **Internet**
  - Verschiedene Netze werden durch **Router** miteinander verbunden



Internet-Kommunikation

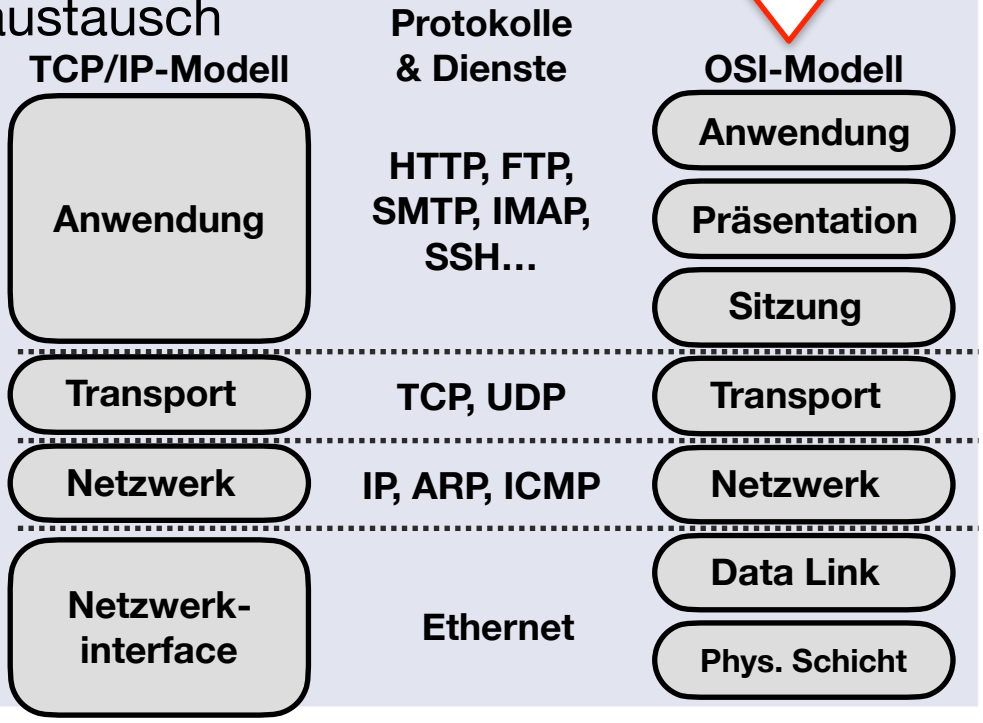
- Anwendungen (Prozesse) kommunizieren meist über spezielle Protokolle
  - HTTP für Kommunikation Webserver – Webbrowser
  - SSH zur Anmeldung auf einen anderen Rechner
  - SMTP + IMAP zur Kommunikation zwischen E-Mailserver und -client
- "Unterhalb" der Anwendungsprotokolle existieren verschiedene generische Protokolle zum Datenaustausch
  - Im selben Netz und zwischen Netzen: TCP, UDP, IP, ICMP
  - Nur im lokalen Netz: ARP
- Daten werden über ein Kommunikationsmedium übertragen
  - Ethernet, WLAN, …



Internet-Kommunikation

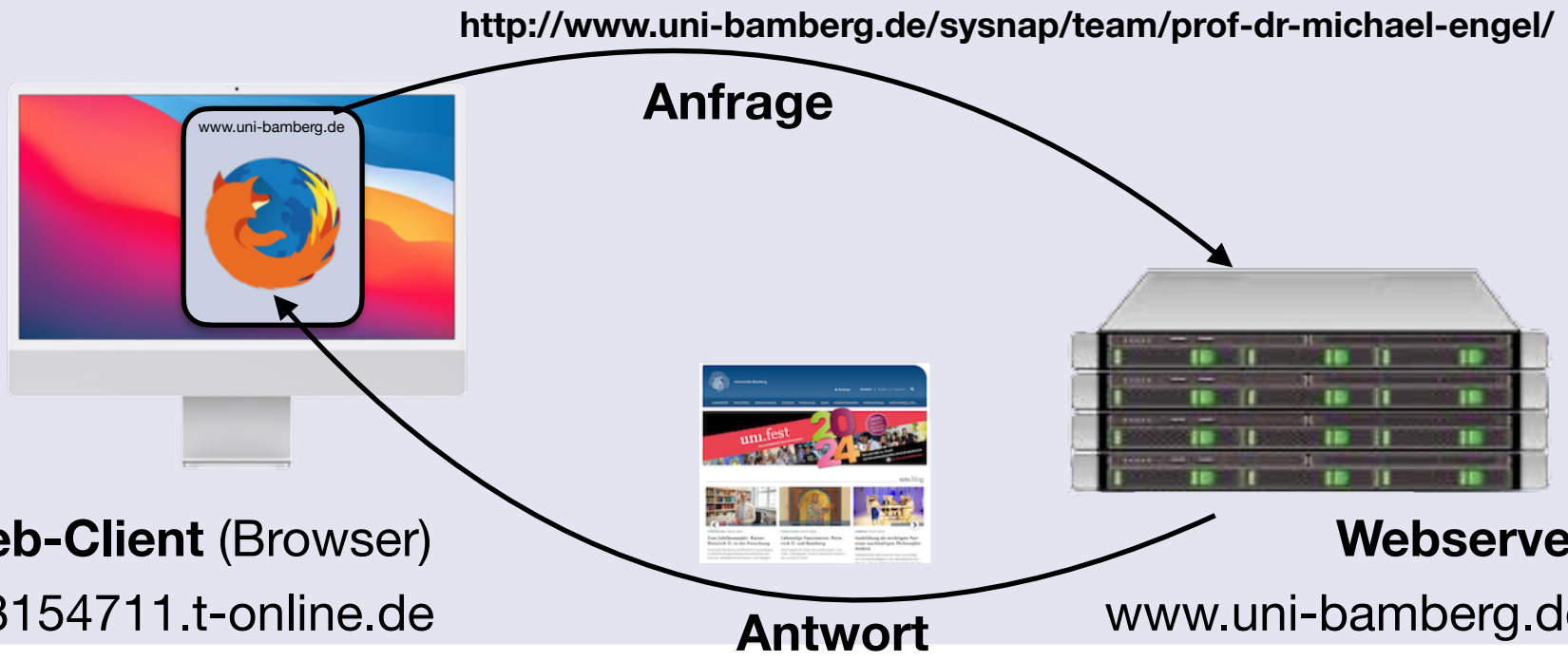
- Anwendungen (Prozesse) kommunizieren meist über spezie
  - HTTP für Kommunikation Webserver – Webbrowser
  - SSH zur Anmeldung auf einen anderen Rechner
  - SMTP + IMAP zur Kommunikation zwischen E-Mailserve
- "Unterhalb" der Anwendungsprotokolle existieren verschiedene generische Protokolle zum Datenaustausch
  - Im selben Netz und zwischen Netzen: TCP, UDP, IP, ICMP
  - Nur im lokalen Netz: ARP
- Daten werden über ein Kommunikationsmedium übertragen
  - Ethernet, WLAN, …

"Please Do Not Throw Salami Pizza Away"



Beispiel: HTTP-Protokoll

- **HTTP** (Hyper Text Transfer Protocol) ist die Basis für Web-Kommunikation
  - Definiert das Format der Kommunikation zwischen **Web-Client** (Browser) und **Webserver**
  - Besteht aus **Anfragen** (*requests*) und **Antworten** (*responses*)
  - Erste Version 1989 von Tim Berners-Lee am CERN entwickelt

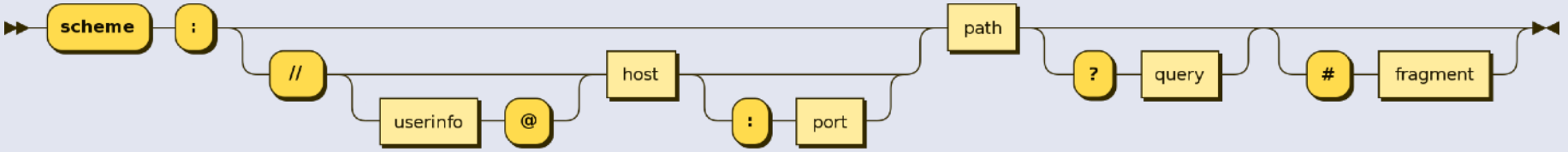


Format von URLs

Universität Bamberg

- **URL** (Universal Resource Locators) identifizieren **eindeutig** eine Ressource im Internet
  - Allgemeinform: **URI** (Universal Resource Indicator)

Format einer URL:



Beispiel-URLs:

`http://www.uni-bamberg.de/sysnap/team/prof-dr-michael-engel/`

Scheme **Host** **Path**

`http://me@example.com:8080/search?term=music`

Scheme **User info** **Host** **Port** **Path** **Query**

Adressierung im Internet

- Wie finden sich Webbrowser und Webserver?
  - z.B. Verbindung von i08154711.t-online.de zu www.uni-bamberg.de?
- **Hostnamen** wie www.uni-bamberg.de werden in numerische **Adressen** übersetzt (wie im Telefonbuch…)
  - IPv4: 4 Bytes (z.B. 141.13.240.24)
  - IPv6: 16 Bytes…
- Verschiedene **Dienste** auf einem Server werden durch **Portnummern** identifiziert
  - …wie Durchwahlnummern beim Telefon
  - Beispiel: 0951/863-**2926**
- Übersetzung zwischen Hostnamen und Adressen ist ein Internetdienst:
  Domain Name Service (**DNS**)

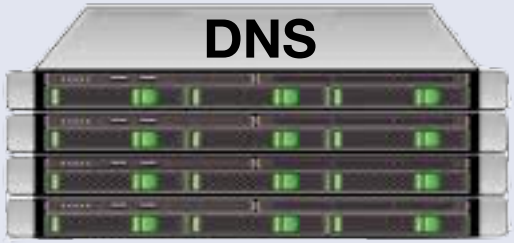
- Portnummern sind für ein Protokoll (wie HTTP) festgelegt oder werden dynamisch zwischen Client und Server verhandelt

HTTP-Protokoll mit DNS

Universität Bamberg

Anfrage

"Adresse von www.uni-bamberg.de?"



DNS-Server

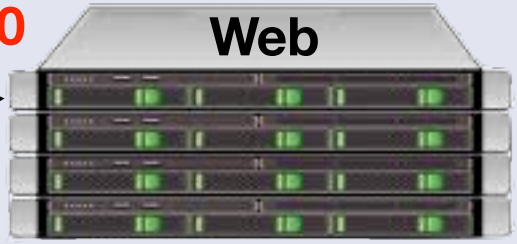
1.1.1.1

Antwort

141.13.240.24

www.uni-bamberg.de

Anfrage Port 80



Web-Client (Browser)

i08154711.t-online.de

Antwort

Webserver

www.uni-bamberg.de

= 141.13.240.24

HTTP-"Gesprächs"protokoll

Universität Bamberg

http://www.uni-bamberg.de/sysnap/team/prof-dr-michael-engel/

DNS-Server (1.1.1.1)

Adresse von www.uni-bamberg.de?

141.13.240.24

Öffne Verbindung zu
141.13.240.24 Port 80

Webserver
(141.13.240.24)

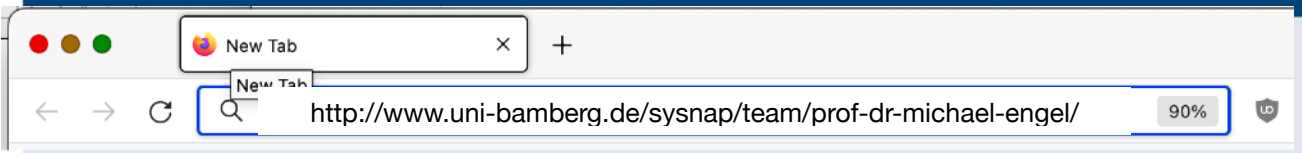
Gib mir http://www.uni-bamberg.de/sysnap

Bitteschön 😀

Schließe Verbindung zu
141.13.240.24 Port 80

HTTP-"Gesprächs"protokoll

Universität Bamberg



Bei HTTP werden Anfragen und Antworten als ASCII-Text versendet

Webserver
(141.13.240.24)

Anfrage

```
GET /sysnap/team/prof-dr-michael-engel/ HTTP/1.1
Host: www.uni-bamberg.de
User-Agent: Mozilla/5.0 (en-US; rv:1.9.1.8) Firefox/3.5.5
Accept: text/html
```


```
HTTP/1.1 200 OK
Server: nginx/1.18.0 (Ubuntu)
Date: Mon, 08 Jul 2024 07:00:38 GMT
Content-Type: text/html
Connection: Closed

<!DOCTYPE html>
<html dir="ltr" lang="de">
<head>
```

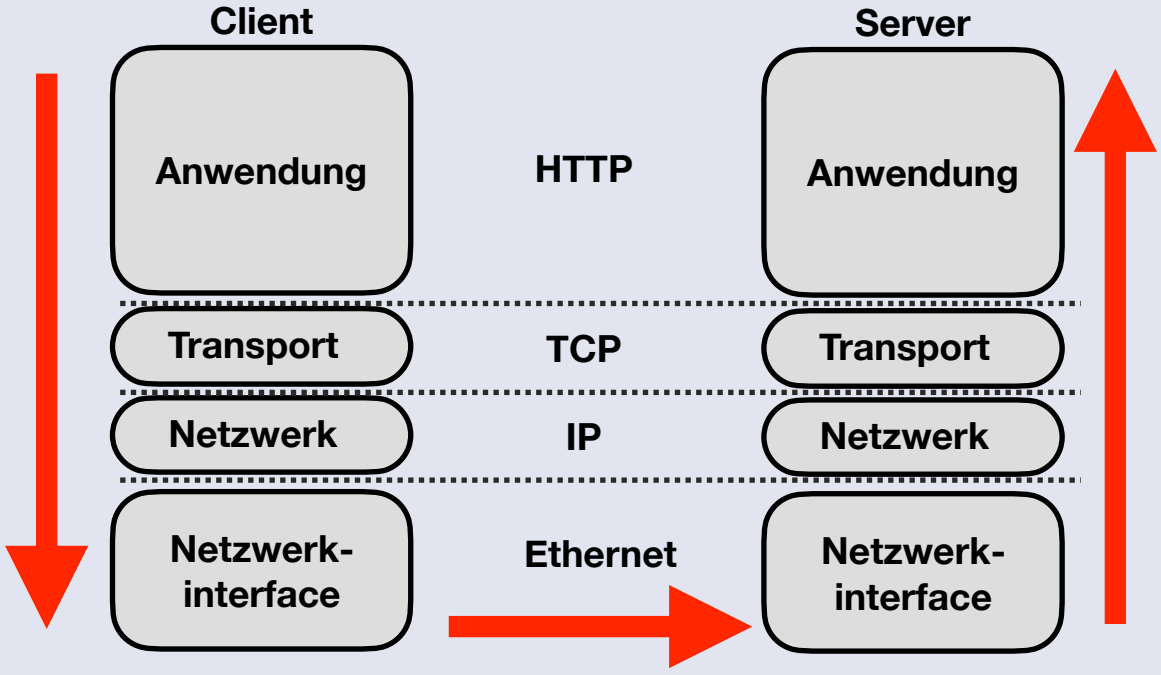
Antwort:
Status (200 = OK)

Leerzeile

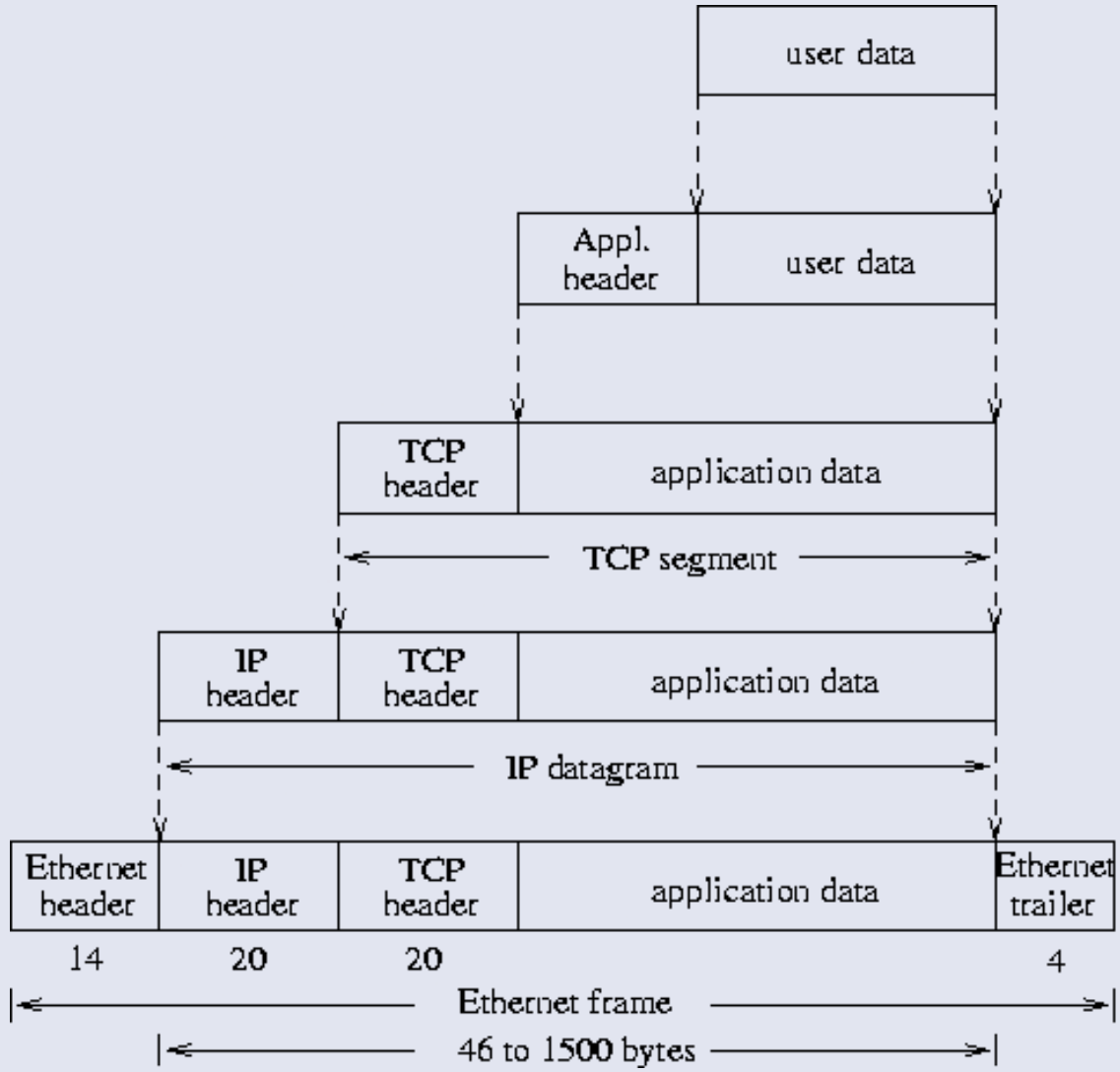
Inhalt der Webseite
(HTML)

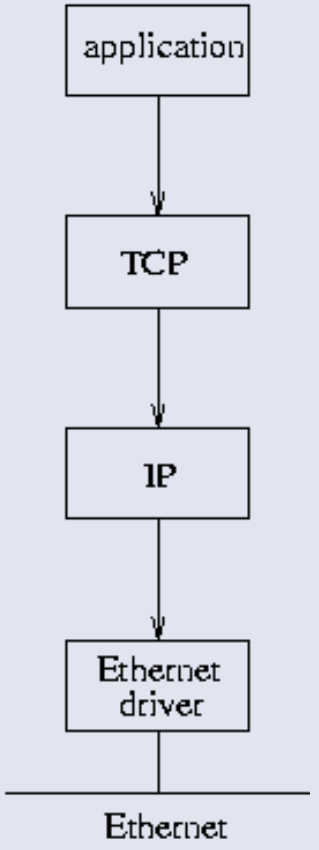
Internet-Kommunikation

- Daten eines Anwendungsprotokoll werden mit Hilfe von Protokollen der niederen Schichten übertragen
- Jede Schicht des Senders fügt den Daten eigene Kontrollinformationen *(Header)* auf dem Weg von Anwendung zum Netz hinzu, beim Empfänger werden die Header wieder Stück für Stück entfernt
  - Transport: Portnummer
  - Netzwerk: Sender/Empfängeradr.
  - Netwerkinterface: Hardware (MAC)-Adr.
  - Enthalten teils auch Checksummen zur Sicherstellung der Korrektheit



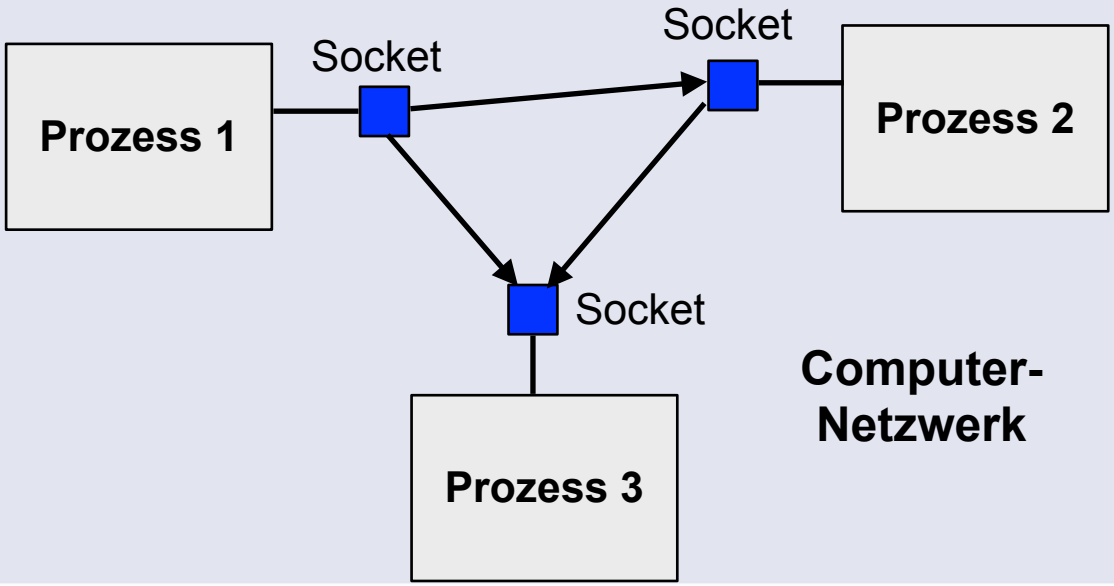
Verschachtelte Pakete





Programme für TCP/IP – Sockets

- Software-Abbildung von *Kommunikationsendpunkten* in einem *Computernetzwerk*
  - Bidirektional
  - Gepuffert
- Abstrahieren von Details des verwendeten Kommunikationssystems
  - Beschrieben durch eine *Domäne (domain* = Protokollfamilie), einen *Typ* und ein *Protokoll*



Sockets: Domänen

Universität Bamberg

- Unix-Domäne
  - Unix-Domänen-Sockets funktionieren wie *bidirektionale Pipes*
  - Können als *special file* im Dateisystem angelegt werden
- Internet-Domäne
  - Verwendet für Kommunikation zwischen Computern über *Internetprotokolle*
- Appletalk-Domäne, DECnet-Domäne, …
  - Viele weitere Domänen für (heute veraltete) Netzwerkprotokolle

- Domänen bestimmen die verwendbaren *Protokolle*
  - z.B. in der Internet-Domäne: TCP/IP oder UDP/IP
- Domänen geben die *Adressfamilie* vor
  - z.B. in der Internet-Domäne: IP-Adresse und Portnummer

Sockets: Typ und Protokoll

- Die wichtigsten Socket-Typen:
  - *Datenstromorientiert* und *zuverlässig*
  - *Nachrichtenorientiert* und *unzuverlässig*
  - *Nachrichtenorientiert* und *zuverlässig*
- Protokolle der Internet-Domäne:
  - TCP/IP-Protokoll
    - Datenstrom- und verbindungsorientiert und zuverlässig
  - UDP/IP-Protokoll
    - Nachrichtenorientiert, verbindungslos und unzuverlässig
    - Nachrichten können verloren gehen oder dupliziert (wiederholt) werden
    - Reihenfolge kann verdreht werden
    - Grenzen von Paketgrößen sind zu beachten (Datagramm-Protokoll)
- Spezifikation eines Protokolls ist oft *redundant*
  - Da es von der Domäne vorgegeben ist

Socketprogrammierung

Achtung: für **IPv6** existieren separate **sockaddr_in6** und **AF_INET6**-Typen

- Erzeugen eines Sockets
  - Systemaufruf:
    ```
    int socket (int domain, int type, int proto);
    ```
    (Rückgabewert ist ein Dateideskriptor)

- Adresszuweisung
  - Sockets werden ohne zugewiesene Adresse erzeugt
  - Adresszuweisung durch:
    ```
    int bind (int socket,
              const struct sockaddr *address,
              socklen_t address_len);
    ```
- `struct sockaddr_in` (für die Internet-Protokollfamilie) enthält:
  - `sin_family`: `AF_INET`
  - `sin_port`: 16-Bit Portnummer
  - `sin_addr`: Struktur mit IP-Adresse, z.B. 192.168.2.1

Socketprogrammierung

- ***Datagramm-Sockets***
  - Kein Verbindungsaufbau notwendig
  - Senden eines Datagramms durch

```
ssize_t sendto (int socket, const void *message,
                size_t length, int flags,
                const struct sockaddr *dest_addr,
                socklen_t dest_len);
```

  - Empfangen eines Datagramms durch

```
ssize_t recvfrom (int socket, void *buffer,
                  size_t length, int flags,
                  struct sockaddr *address,
                  socklen_t *address_len);
```

Socketprogrammierung

- ***Datenstrom-Sockets***
  - Verbindungsaufbau erforderlich
  - *Clients* (Benutzerprozess) erzeugen eine Kommunikationsverbindung zu einem *Server* (Serverprozess)

- ***Client:*** Verbindungsaufbau für datenstromorientierte Sockets
  - Verbinden des Sockets mit

```
int connect (int socket,
             const struct sockaddr *address,
             socklen_t address_len);
```

  - Senden/Empfangen mit `write` und `read` (oder `send` und `recv`)
  - Verbindungsabbau mit `close` (Schließt den Socket)

Socketprogrammierung

- ***Server:*** akzeptiert Anfragen/Aufträge
  - bindet Socket an eine Adresse (sonst nicht erreichbar)
  - bereitet Socket auf Verbindungsanforderungen vor durch

```
int listen (int s, int queuelen);
```

  - akzeptiert einzelne Verbindungsanforderungen durch

```
int accept (int s, struct sockaddr *addr,
            socklen_t *addrlen);
```

  - gibt einen neuen Socket zurück, der mit dem Client verbunden ist
  - blockiert, falls kein Verbindungswunsch vorhanden
  - liest Daten mit read und führt den angebotenen Dienst aus
  - schickt das Ergebnis mit write zurück zum Sender
  - schließt den neuen Socket mit close

Socketprogrammierung


```
#define PORT 6789
#define MAXREQ (4096*1024)

char buffer[MAXREQ], body[MAXREQ], msg[MAXREQ];
void error(const char *msg) { perror(msg); exit(1); }

int main() {
  int sockfd, newsockfd;
  socklen_t clilen;
  struct sockaddr_in serv_addr, cli_addr;
  int n;
  sockfd = socket(PF_INET, SOCK_STREAM, 0);
  if (sockfd < 0) error("ERROR opening socket");
  bzero((char *) &serv_addr, sizeof(serv_addr));
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_addr.s_addr = INADDR_ANY;
  serv_addr.sin_port = htons(PORT);
  if (bind(sockfd, (struct sockaddr *) &serv_addr,
      sizeof(serv_addr)) < 0) {
    error("ERROR on binding");
  }
  listen(sockfd,5);
  …
```

Hier wird der Socket erzeugt und an eine Adresse gebunden

Socketprogrammierung

Universität Bamberg

```
  …
  while (1) {
    clilen = sizeof(cli_addr);
    newsockfd = accept (sockfd, (struct sockaddr *) &cli_addr,
                        &clilen);
    if (newsockfd < 0) error("ERROR on accept");
    bzero(buffer,sizeof(buffer));
    n = read (newsockfd,buffer,sizeof(buffer)-1);
    if (n < 0) error("ERROR reading from socket");
    snprintf (body, sizeof (body),
              "<html>\n<body>\n"
              "<h1>Hello web browser</h1>\nYour request was\n"
              "<pre>%s</pre>\n"
              "</body>\n</html>\n", buffer);
    snprintf (msg, sizeof (msg),
             "HTTP/1.0 200 OK\n"
              "Content-Type: text/html\n"
              "Content-Length: %d\n\n%s", strlen (body), body);
    n = write (newsockfd,msg,strlen(msg));
    if (n < 0) error("ERROR writing to socket");
    close (newsockfd);
  }
}
```

Neue Verbindung annehmen

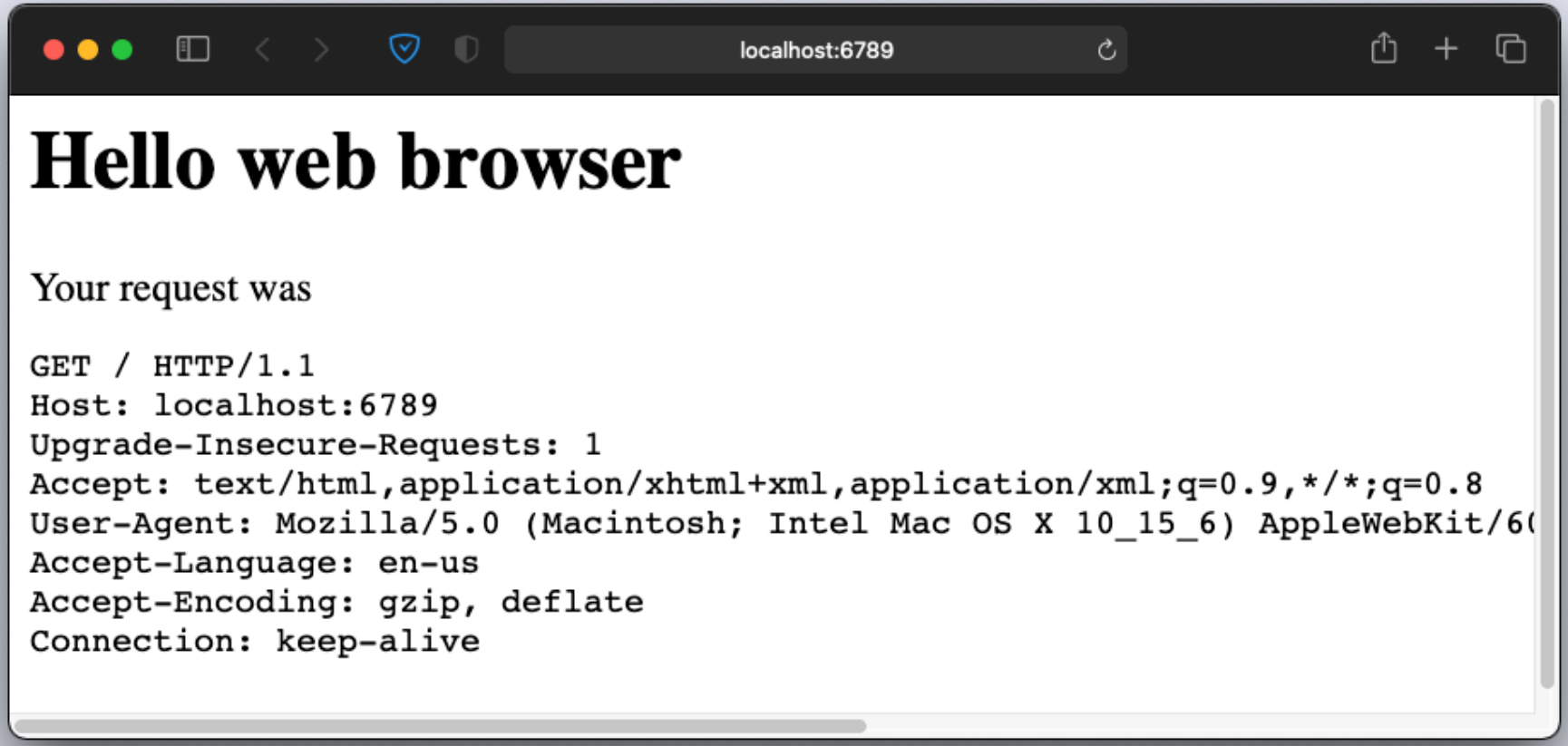
Lies die HTTP-Anfrage

Erzeuge Antwort und sende sie zurück

Schließe die Verbindung wieder

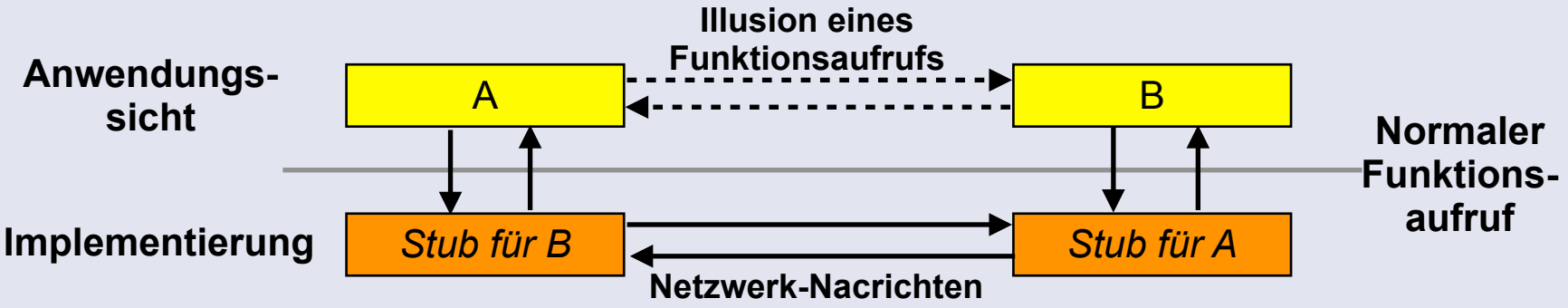
Sockets: Beispiel für HTTP-Echo

Universität Bamberg



Remote procedure calls (RPC)

- RPCs sind *Funktionsaufrufe* zwischen *unterschiedlichen Prozessen*
  - Besitzen hohen Abstraktionsgrad (keine Protokolle etc. programmieren)
  - Meist nicht direkt durch das Betriebssystem bereitgestellt
    - Abbildung auf andere Kommunikation (z.B. Nachrichten) notwendig
  - Abbildung mit Hilfe mehrerer Nachrichten
    - Anfragenachricht enthält Anfrage, die Funktion des anderen Prozesses mit den übergebenen Parametern aufzurufen
    - Antwortnachricht enthält Ergebnis(se) des Aufrufs



- Beispiele: NFS (ONC RPC), Linux D-BUS

Referenzen

[1] James F. Kurose, Keith W. Ross
Computer Networking: A Top-Down Approach
Pearson 2021, ISBN-13: 978-1292405469

[2] Wie funktioniert das Internet? – Bibliothek der Sachgeschichten
https://www.youtube.com/watch?v=fpqhjEtznVk

[3] W. Richard Stevens, Bill Fenner, Andrew M. Rudoff
Unix Network Programming,
Volume 1: The Sockets Networking API
Prentice Hall, Third edition 2003, ISBN-13: 978-0131411555

[4] Kevin Fall, W. Richard Stevens
TCP/IP Illustrated: The Protocols, Volume 1
Addison-Wesley 2011, ISBN-13: 978-0321336316