Netzwerk-Programmierung in C

Oliver Bartels

Fachbereich Informatik
Universität Hamburg

2. Juli 2014

Inhaltsverzeichniss

IP-Adressen intern sowie extern

IPv4 (192.168.0.1)

- 2^{32} Adressen
- 4 mal 8 Bit (1Byte) abgetrennt mit "."
- Erstes wirkliches Protokoll
- Problem: Zu wenig Adressen

IPv6 (2001:db8:0:8d3:0:8a2e:70:7344)

- 2^{128} mögliche Adressen
- 16Bit (2Byte) in hexadezimal mit ":"getrennt
- Problem: Nicht jeder hat bereits den Umstieg gemacht, im Heim- oder Firmennetzwerk ist IPv6 unnötig

$\Rightarrow$ beide Protokolle müssen akzeptiert werden

Speicherung der Daten im Netzwerk und im Rechner

Big-Endian

- Speicherordnung mit großem Byte-Anteil zuerst
- logische Anordnung der Bytes
- Wird im Netzwerk immer verwendet
- b34f ist b3 4f im Speicher

Little-Endian

- Speicherordnung mit kleinem Byte-Anteil zuerst
- umgekehrte Anordnung
- Wird bei Intel-Prozessoren verwendet
- b34f wird zu 4f b3 im Speicher

$\Rightarrow$ Daten für Netzwerk einmal umschreiben

Wie funktioniert das denn?

- IP-Adresse muss bekannt sein
- auch der Port ist wichtig (16Bit)
- Kommunikation mittels "file descriptor"
- file descriptor für Netzwerk über socket()

Sockets im allgemeinen

Sockets

- 2 verschiedene Arten von Sockets:
  Stream und **Datagram Sockets**
- Stream sockets = 2-Wege Verbindung Sockets
- Datagram Sockets = verbindungslose Sockets

Stream Sockets

TCP

- **T**ransmission **C**ontrol **P**rotocol
- stellt eine Verbindung zwischen 2 Punkten her
- Wird fast ausschließlich als Transfer Protokoll genutzt
- 3-way handshake Prinzip zum Verbindungsaufbau

HTTP

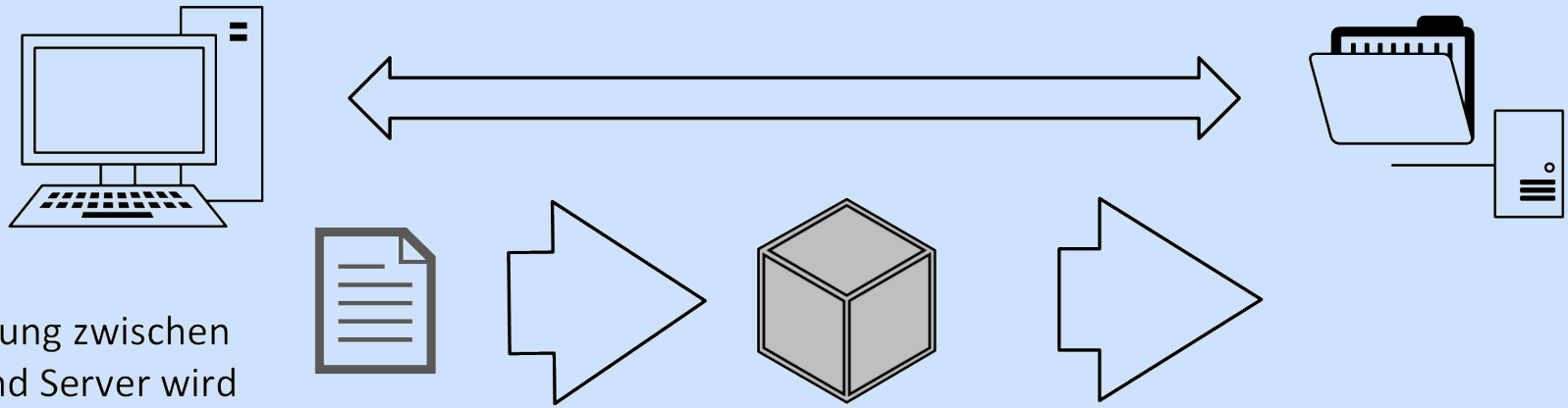
- **H**yper**T**ext **T**ransfer **P**rotocol
- setzt auf TCP auf
- Wird genutzt im Webseiten über das WWW zu laden

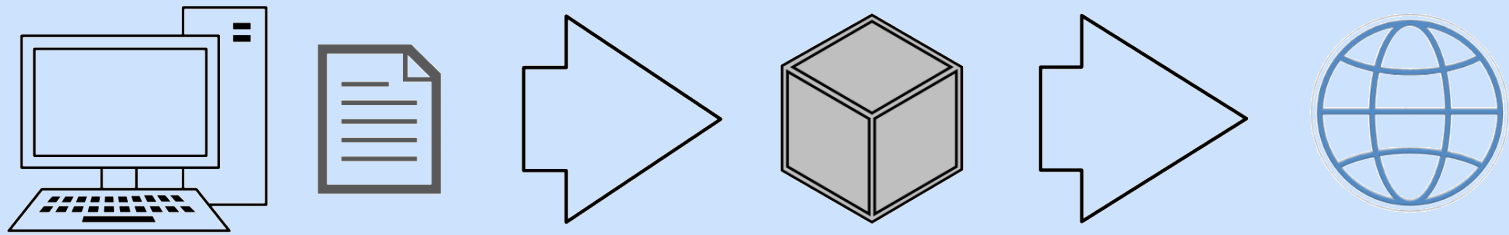
Datagram Socket

UDP

- **U**ser **D**atagram **P**rotocol
- verbindungsloses Netzwerkprotokoll
- Datenübertragung mittels Port
- unempfindlich und kein Datenschutz
- kleinere Datenpakete
- Verwendung bei einer DNS-Anfrage und z.B. VoIP

Veranschaulichung





Benötigte Libraries

```
1 #include <sys/types.h>
2 #include <netdb.h>
3 #include <sys/socket.h>
4 #include <arpa/inet.h>
5 #include <netinet/in.h>
```

Adressinformationen bekommen

```
1  struct addrinfo
2      int              ai_flags;
3      int              ai_family;
4      int              ai_socktype;
5      int              ai_protocol;
6      size_t           ai_addrlen;
7      struct sockaddr *ai_addr;
8      char            *ai_canonname;
9
10     struct addrinfo *ai_next;
```

```
1  int getaddrinfo(const char *node,
2                  const char *service,
3                  const struct addrinfo *hints,
4                  struct addrinfo **res);
```

socket()

Welcher Sockettyp soll gewählt werden

- Entscheidung treffen ob Stream oder Datagram
- IPv4 oder IPv6

```
1  int s;
2  struct addrinfo hints, *res;
3  // ausfüllen von "hints"
4  getaddrinfo("www.example.com", "http",
5                &hints, &res);
6  // error-checking
7  // nach gültigen Einträgen in der linked list
8  // "res" suchen
9  s = socket(res->ai_family, res->ai_socktype,
10             res->ai_protocol);
```

bind()

```
1  int bind(int sockfd, struct sockaddr *my_addr,
2           int addrlen);
```

```
1  memset(&hints, 0, sizeof hints);
2  hints.ai_family = AF_UNSPEC;
3  hints.ai_socktype = SOCK_STREAM;
4  hints.ai_flags = AI_PASSIVE;
5
6  getaddrinfo(NULL, "3490", &hints, &res);
7
8  sockfd = socket(res->ai_family,
9              res->ai_socktype, res->ai_protocol);
10 bind(sockfd, res->ai_addr, res->ai_addrlen);
```

connect()

```
1  int connect(int sockfd, struct sockaddr
2              *serv_addr, int addrlen);
```

```
1  memset(&hints, 0, sizeof hints);
2  hints.ai_family = AF_UNSPEC;
3  hints.ai_socktype = SOCK_STREAM;
4
5  getaddrinfo("www.example.com", "3490", &hints,
6              &res);
7
8  sockfd = socket(res->ai_family,
9              res->ai_socktype, res->ai_protocol);
10
11 connect(sockfd, res->ai_addr, res->ai_addrlen);
```

listen (), accept()

listen()

```
1 int listen(int sockfd, int backlog);
```

accept()

```
1 int accept(int sockfd, struct sockaddr *addr,
2              socklen_t *addrlen);
3
4 new_fd = accept(sockfd, (struct sockaddr *)
5                  &their_addr, &addr_size);
```

send (), recv()

```
1 int send(int sockfd, const void *msg, int len,
2           int flags);
```

```
1 char *msg = "Coding macht Spaß!";
2 int len, bytes_sent;
3
4 len = strlen(msg);
5 bytes_sent = send(sockfd, msg, len, 0);
```

```
1 int recv(int sockfd, void *buf, int len,
2           int flags);
```

sendto (), recvfrom() für UDP

sendto()

```
1 int sendto(int sockfd, const void *msg,
2            int len, unsigned int flags,
3            const struct sockaddr *to,
4            socklen_t tolen);
```

recvfrom()

```
1 int recvfrom(int sockfd, void *buf, int len,
2               unsigned int flags,
3               struct sockaddr *from,
4               int *fromlen);
```

close (), shutdown()

close()

```
1 close(sockfd);
```

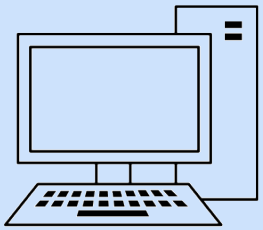
shutdown()

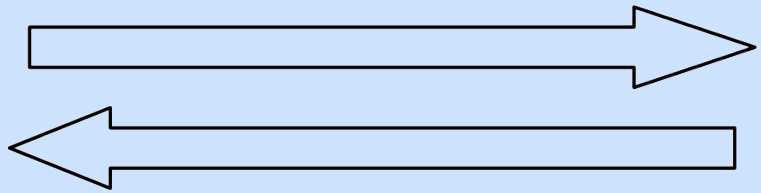
```
1 int shutdown(int sockfd, int how);
```

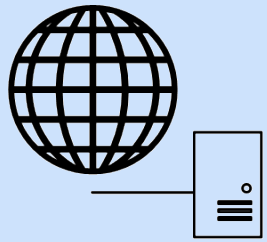
Veranschaulichung

Verbindungsaufbau:
- neuen Socket erstellen
- mit *connect ()* zum Server verbinden

Client







Server im Internet

Besitzt einen Socket

Listener Socket bemerkt Verbindungsaufbau:
- *accept()* wenn Server bereit ist
- *send()* Daten zum Client

Besitzt **einen** *listen()***er** Socket und ein Socket **pro Verbindung** mit einzelnen Clients

Mit *recv()* auf Daten warten

GIO Serverseite

```
1  #include <glib.h>
2  #include <gio/gio.h>
3
4  g_type_init();
5
6  GError * error = NULL;
7
8  GSocketService *service = g_socket_service_new();
9
10 g_socket_listener_add_inet_port (
11                          GSocketListener *listener,
12                          guint16 port,
13                          GObject *source_object,
14                          GError **error);
```

Serverseite

```
1  g_signal_connect(instance, detailed_signal,
2                   c_handler, data)
3
4  g_socket_service_start (service);
5
6  gboolean
7  incoming_callback (GSocketService     *service,
8                     GSocketConnection *connection,
9                     GObject           *source_object,
10                    gpointer           user_data)
```

GIO Clientseite

```
1  #include <glib.h>
2  #include <gio/gio.h>
3
4  g_type_init();
5
6  GError * error = NULL;
7
8  GSocketConnection * connection = NULL;
9
10 GSocketClient * client = g_socket_client_new();
```

```
1  GSocketConnection *
2  g_socket_client_connect_to_host (
3                          GSocketClient *client,
4                          const gchar *host_and_port,
5                          guint16 default_port,
6                          GCancellable *cancellable,
7                          GError **error);
8
9  GInputStream *
10 g_io_stream_get_input_stream (
11                                GIOStream *stream);
12
13 GOutputStream *
14 g_io_stream_get_output_stream (
15                                GIOStream *stream);
```

verschiedene Operationen möglich wie z.B.

```
1 gssize
2 g_output_stream_write (GOutputStream *stream,
3                        const void *buffer,
4                        gsize count,
5                        GCancellable *cancellable,
6                        GError **error);
```

Fehler überprüfung

```
1 if (error != NULL)
2 {
3     g_error (error->message);
4 }
```

Zusammenfassung

- Verbindungsart wählen (Stream oder Datagram socket)
- TCP ist eine gesicherte Verbindung, UDP ist schneller, aber nicht so sicher
- getaddrinfo() vieles ausfüllen lassen
- bind() für Server um Port mit Programm zu verknüpfen, bei Client nicht nötig
- schneller und einfacher geht es mit der GLib
- Umfangreiches Thema, es wurde nicht alles behandelt

Literaturverzeichnis

- http://de.wikipedia.org/wiki/Transmission_Control_Protocol
- http://de.wikipedia.org/wiki/IPv4
- http://de.wikipedia.org/wiki/IPv6
- https://developer.gnome.org/gio/stable/
- http://linux.die.net/man/3/getaddrinfo
- http://stackoverflow.com/questions/9513327/gio-socket-server-client-example
- http://beej.us/guide/bgnet/output/html/multipage/index.html