C - Grundlagen und Konzepte

— Networking in C —

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

Vorgelegt von: Oliver Bartels
E-Mail-Adresse: 3bartels@informatik.uni-hamburg.de
Matrikelnummer: 6527101
Studiengang: Informatik

Betreuer: Michael Kuhn

Hamburg, den 04.09.2014

Inhalt der Ausarbeitung

In dieser Ausarbeitung, werde ich meine etwas genauer als in der Präsentation auf das Netzwerkprogrammieren in C und mit der GLib eingehen.
Ich gehe zuerst auf die allgemeinen Informationen für Netzwerk und Kommunikation über das Internet ein und werde den Bericht mit Informationen zur Syntax bei der Programmierung beenden.
Dabei werde ich, wie in der Präsentation, nicht den gesamten Bereich abdecken können. Ich werde lediglich eine Einleitung in das sehr große Thema geben.

Inhaltsverzeichnis

1 Einleitung und Netzwerkinfos

In diesem Kapitel gebe ich eine generelle Einleitung über die Funktionalität des Netzwerks und wie auch Daten im Internet verschickt werden.

1.1 IPv4 und IPv6

Es gib 2 Internetprotokolle, einmal das IPv4 (Internet protocol 4) und das neuere IPv6. Für das spätere Programmieren ist es wichtig dies nicht außer acht zu lassen.

1.1.1 IPv4

Dies ist das ältere Protokoll, welches auch noch sehr verbreitet ist und wenn wir heute von IP-Adressen sprechen haben viele eine IPv4-Adresse im Kopf.
Eine solche Adresse sieht beispielsweise so aus: 192.168.0.1 und hat dementsprechend folgende Eigenschaften

- 2^{32} Adressen
- es sind jeweils 4 mal 8 Bit (1Byte) abgetrennt mit einem Punkt

Wie man sieht, ist die maximale Anzahl der Adressen mit 2^{32} begrenzt und kann nicht erhöht werden, da niemand vor über 20 Jahren, als das Protokoll erfunden wurde, gedacht hatte dass die Anzahl elektronischer Geräte mit Anschluss ans Netzwerk und Internet so stark steigen würde.

1.1.2 IPv6

Bereits 1998 wurde daraufhin ein neues Protokoll entwickelt, das IPv6, welches 2^{96} mehr Adressen als das IPv4 anbietet.
Eine IPv6 Adresse sieht z.B. so aus: 2001:db8:0:8d3:0:8a2e:70:7344 Woraus sich folgende Eigenschaften erschließen:

- 2^{128} Adressen
- 8 mal 16Bit (2Byte) in Hexadezimal mit einem Doppelpunkt getrennt

Die Anzahl der Adressen in IPv6 ist sogar so hoch, dass ein großer Teil der Adressen reserviert wurde, die ausgeschriebene Gesamtzahl der Adressen ist übrigens 340 Sextillionen $(3,4 \cdot 10^{38})$
Das einzige Problem mit IPv6 ist, dass noch längst nicht jedes Gerät oder jede Seite den Umstieg auf IPv6 gemacht hat.
Folglich müssen beide Protokolle berücksichtigt und akzeptiert werden in der Netzwerktechnik.

1.2 Wie werden Daten verschickt?

Weiterhin ist es wichtig zu wissen wie die Daten auf dem Computer gespeichert werden, denn dies ist unter Umständen anders als im Netzwerk sein.

1.2.1 Big-Endian

Hier Erfolgt die Speicherung mit dem großem Byte-Anteil zuerst. Dies ist auch gleich die logische Anordnung, so wie wir uns die Zahlen auch merken.
Diese Anordnung wird im Netzwerk verwendet und auch bei manchem Prozessorarchitekturen (was wiederum heißt, dass der Computer diese Anordnung verwendet).
Als Beispiel würde b34f zu b3 4f im Speicher werden. Wie man also sieht, wird der große Anteil zuerst gespeichert und dann der Kleine.

1.2.2 Little-Endian

Hier Erfolgt die Speicherung mit dem kleinem Byte-Anteil zuerst. Dies ist die umgekehrte Anordnung des Big-Endians.
Diese Anordnung wird bei Intel-Prozessoren verwendet und deshalb nutzen viele Computer Little-Endian.
Als Beispiel würde b34f zu 4f b3 im Speicher werden. Wie man gut sehen kann steht der kleine Anteil nun vorne und der Große hinten.

Weil es diese verschiedene Arten der Speicherung bei aktuellen Computern gibt, hat man sich geeinigt erst einmal alle Daten für das Netzwerk umzuschreiben, damit am Ende auch alles richtig zugeordnet wird.

1.3 Funktionsweise

Damit man über das Netzwerk etwas schicken kann, braucht man noch mehr als nur die IP-Adresse. Der Port (16Bit) spielt auch eine wichtige Rolle damit der Computer weiß wohin die Daten müssen.
Allerdings passiert die eigentliche Kommunikation nur über einen „file descriptor".
Dieser file descriptor wird für das Netzwerk von der Funktion socket() erstellt.
Allgemein ist ein file descriptor nichts weiter als ein Integer, welcher mit einer geöffneten Datei zusammenhängt.

2 Sockets

In diesem Kapitel werde ich etwas näher auf Sockets im allgemeinen eingehen und die verschiedenen Arten von Sockets erklären.

Es gibt 2 Arten von Sockets:

- Stream Sockets
- Datagram Sockets

Stream Sockets stellen eine 2-Wege Verbindung her, d.h. es besteht eine feste Verbindung zwischen 2 Rechnern.

Datagram Sockets sind verbindungslose Sockets. Sie stellen keine Verbindung an sich her und müssen jedes mal ihr Ziel mit auf das Paket schreiben.

2.1 Stream Sockets

2.1.1 TCP

TCP steht für Transmission Control Protocol und stellt eine Verbindung zwischen 2 Punkten her. Diese Verbindungsart wird fast ausschließlich als Transfer Protokoll genutzt, da sie recht sicher ist. Zum Verbindungsaufbau dient hier der 3-way handshake.

2.1.2 HTTP

HTTP steht für Hypertext Transfer Protocol und setzt auf TCP auf. Diese Verbindungsart wird für Webseiten im WWW genutzt.

2.2 Datagram Sockets

2.2.1 UPD

UDP steht für User Datagram Protocol und und ist ein verbindungsloses Netzwerkprotokoll.
Die Datenübertragung findet mittels Port statt, welcher in dem gesendeten Paket steht. Das Protokoll ist verhältnismäßig unempfindlich, da dies Gegenseite nicht überprüft ob Fehler vorliegen, es wird also jedes Paket akzeptiert. Dies hat zur Folge das die Daten nicht so gut geschützt sind.
Die Datenpakete sind i.d.R kleiner, aber dafür häufiger.
Verwendung findet dieses Protokoll besonders bei DNS-Anfragen oder bei VoIP, wie z.B. Teamspeak oder Skype.

2.2.2 Veranschaulichung

Als Bild hab ich das Ganze mal so dargestellt

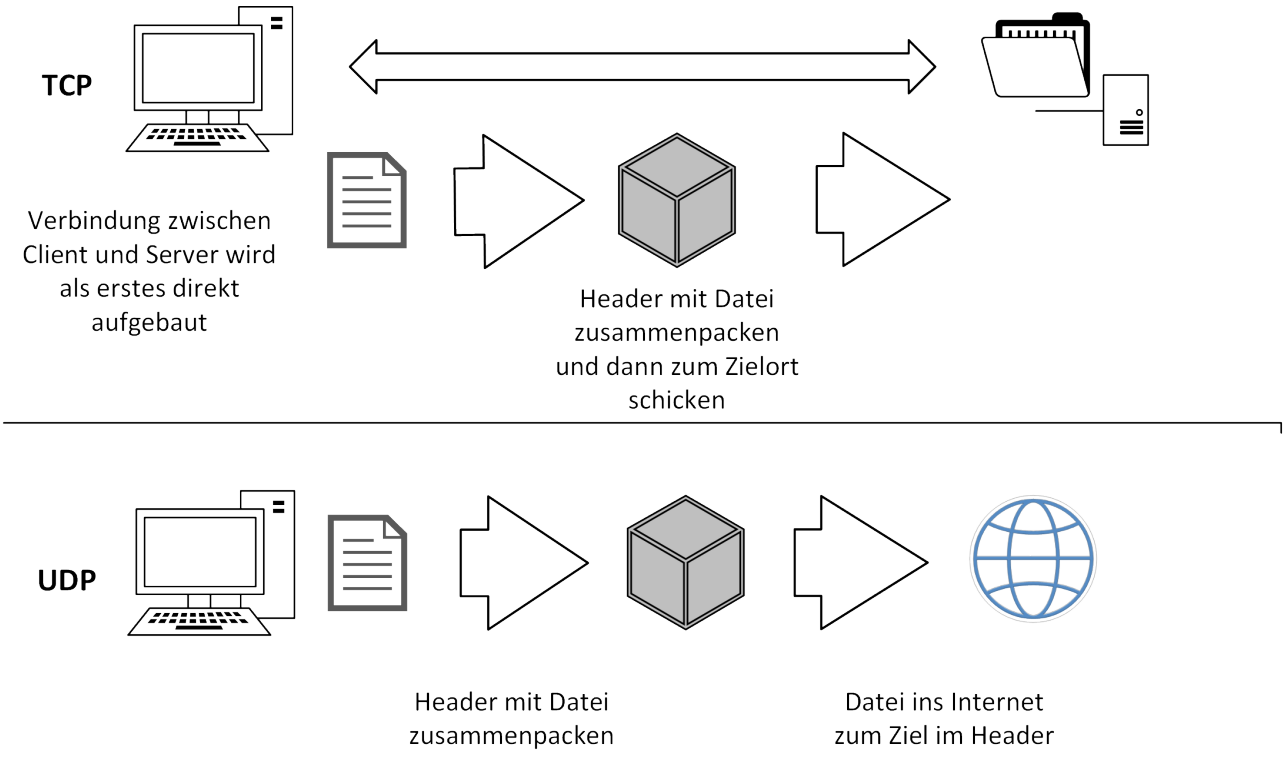

Abbildung 2.1: UDP und TCP

3 Libc Verbindungsaufbau

In diesem Kapitel werde ich nun zeigen wie eine Verbindung mit der normalen cLib erstellt wird.

3.1 Verbindungsaufbau ohne GLib (nur libc)

3.1.1 Headerdateien

Ersteinmal müssen die Headerdateien eingefügt werden, folgende Headerdateien habe ich eingefügt.

Listing 3.1: Headerdateien

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

3.1.2 Adressinformationen

Damit wir ein Socket erstellen können, müssen wir zuerst die Adressinformationen auslesen. Dies ist mithilfe eines structs möglich, es kann mit allen relevanten Daten des Netzwerks befüllt werden.

Listing 3.2: Struct addrinfo

```
1   struct addrinfo
2   {
3       int              ai_flags;
4       int              ai_family;
5       int              ai_socktype;
6       int              ai_protocol;
7       size_t           ai_addrlen;
8       struct sockaddr *ai_addr;
9       char            *ai_canonname;
10
11      struct addrinfo *ai_next;
12  };
```

Genauer erklärt stehen die einzelnen Variablen für folgendes:

- flags: AI_PASSIVE, AI_CANONNAME, etc. (gehe ich hier nicht genauer auf ein)
- family: AF_INET, AF_INET6, AF_UNSPEC (unspec für jedes Protokoll)
- socktype: SOCK_STREAM, SOCK_DGRAM (Stream oder Datagram Socket)
- protocol: am besten mit 0 befüllen damit jedes Protokoll akzeptiert wird
- addrlen: größe von ai_addr in bytes
- addr: struct sockaddr_in or _in6 (Die Adresse in IPv4 oder v6)
- canonname: full canonical hostname: Der Hostname
- next: linked list, next node

Das addrinfo struct wird am Besten mithilfe von getaddrinfo() aufgerufen und ausgefüllt. Dieses sieht so aus:

Listing 3.3: getaddrinfo

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

Auch hier nochmal genauer erklärt:

- *node = adresse (www.) oder IP
- *service = der Port (oder HTTP etc.)
- *hints = struct von addrinfo der nach laden von addrinfo an getaddrinfo übergeben wird
- addrinfo **res = Pointer zu einer linked-list von Ergebnissen, die wir wiederbekommen

Zu beachten ist, dass addrinfo eine linked list ist und man mit addrinfo *ai_next auf die nächste Node zugreifen kann.

3.2 socket()

Jetzt sehen wir uns die Erstellung des Sockets an.
Es muss nun eine Entscheidung getroffen werden, wollen wir Stream oder Datagram Socket haben. Natürlich eigentlich auch relevant ist, ob IPv4 oder IPv6 verwendet wird.

Zunächst erst einmal der Aufbau für die Erstellung eines Sockets:

Listing 3.4: Socket erstellen

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

Wie man erkennen kann, nimmt die Funktion socket() die Informationen von addrinfo und erstellt mithilfe dieser Informationen den Socket.

3.2.1 bind()

Damit wir nun einen Port an einem Socket binden können, gibt es bind(). Dies wird später bei der Funktion listen() benötigt.

Listing 3.5: bind() Syntax

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

Listing 3.6: bind() zusammen mit socket()

```
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);
```

Wie man sehen kann benötigt bind() den socket file descriptor und auch Daten von addrinfo, mit denen man dann einen Port an einem Socket gebunden hat.

3.2.2 connect()

Damit wir uns zu einem anderen Socket verbinden können, benötigt man natürlich auch eine connect() Funktion.
Die Syntax sieht wie folgendermaßen aus:

Listing 3.7: connect()

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

connect() benötigt wieder einen socket file descriptor, dann natürlich auch die Adresse des Servers zu dem man verbunden werden will und auch die Länge der Adresse.
Im Zusammenhang sieht das so aus:

Listing 3.8: connect() im Zusammenhang

```
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, &res);
6
7  sockfd = socket(res->ai_family, res->ai_socktype,
8                  res->ai_protocol);
9
10 connect(sockfd, res->ai_addr, res->ai_addrlen);
```

3.2.3 listen() + accept()

Damit eine Verbindung akzeptiert werden kann, gucken wir uns zunächst listen an:

Listing 3.9: listen()

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

Damit wir ein Socket erstellt, welcher auf eine Verbindung wartet von einem Client.
Wenn nun eine Verbindung hergestellt wird, muss der Server sie noch akzeptieren und das geht so:

Listing 3.10: accept()

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

new_fd = accept(sockfd, (struct sockaddr *) &their_addr,
                &addr_size);
```

Der Rückgabewert von accept ist int und damit erstellt er einen neuen file descriptor, welcher dann die Verbindung zu dem Client vom Server darstellt und über diesen auch kommuniziert.

3.2.4 send() + recv()

Weiterhin sehr wichtig zur Datenkommunikation sind die Funktionen send() und recv(). Die eine Funktion ist für das senden von Dateien zuständig, die andere für das erhalten. Die Syntax von send() sieht so aus:

Listing 3.11: send()

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

- sockfd : Der Socket descriptor zu dem man Daten senden will.
- msg : Dies Ist ein pointer zu den Daten die gesendet werden sollen.
- len : Ist die größe der Datei in Byte.
- flags : Bietet verschiedene Optionen an, aber generell auf 0

Der Rückgabewert ist int und dort werden die gesendeten Bytes angegeben.

Und einmal im Zusammenhang:

Listing 3.12: send() im Zusammenhang

```
char *msg = "C-Coding ist toll!";
int len, bytes_sent;

len = strlen(msg);
bytes_sent = send(sockfd, msg, len, 0);
```

recv() sieht dagegen so aus:

Listing 3.13: recv()

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

- sockfd : Der Socket descriptor zu dem man Daten senden will.
- *buf : Der Buffer in dem die Informationen zwischengespeichert werden.
- len : Die maximale Buffergröße
- flags : Auch hier wieder verschiedene Optionen, aber i.d.R auf 0 setzen

Rückgabewert ist wieder int, aber dieses mal wird -1 bei Fehler und 0 wenn die Verbindung geschlossen wurde zurückgegeben

3.2.5 recvfrom() + sendto()

Jetzt gucken wir uns noch kurz die Funktionen für den Datentransfer über ein Datagram Socket an.

Listing 3.14: sendto()

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

Listing 3.15: recvfrom()

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

Beide Funktionen verhalten sich größtenteils gleich und benötigen auch die selben Variablen.
Der Unterschied ist, dass man, da keine Verbindung aufgebaut ist, eine Adresse immer mit angeben muss. Jeweils einmal wohin gesendet wird oder wovon empfangen wird.

3.2.6 close() + shutdown()

Damit Verbindungen auch getrennt werden können, bzw. Sockets geschlossen werden, gibt es 2 Funktionen.
Einmal close():

Listing 3.16: close()

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

Und einmal shutdown():

Listing 3.17: shutdown()

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

Wenn man die Verbindung einfach schnell schließen will und nicht soll mehr geschrieben werden, dann kann mann close() benutzen. Shutdown() hingegen bietet da mehr Optionen:
Wenn how auf 0 gesetzt wird, dann kann man nichts mehr empfangen auf dem Socket.
Wenn how auf 1 ist dann kann man nichts mehr senden.
Bei how auf 2 geht beides nicht mehr.

Aber auch bei 2, ist shutdown() nicht genauso wie close(), der Speicherplatz wird erst wieder frei wenn man close() benutzt.

3.3 Veranschaulichung

Damit man sich das ganze visuell besser vorstellen kann, hab ich hier eine Veranschaulichung eines Verbindungsaufbaus.

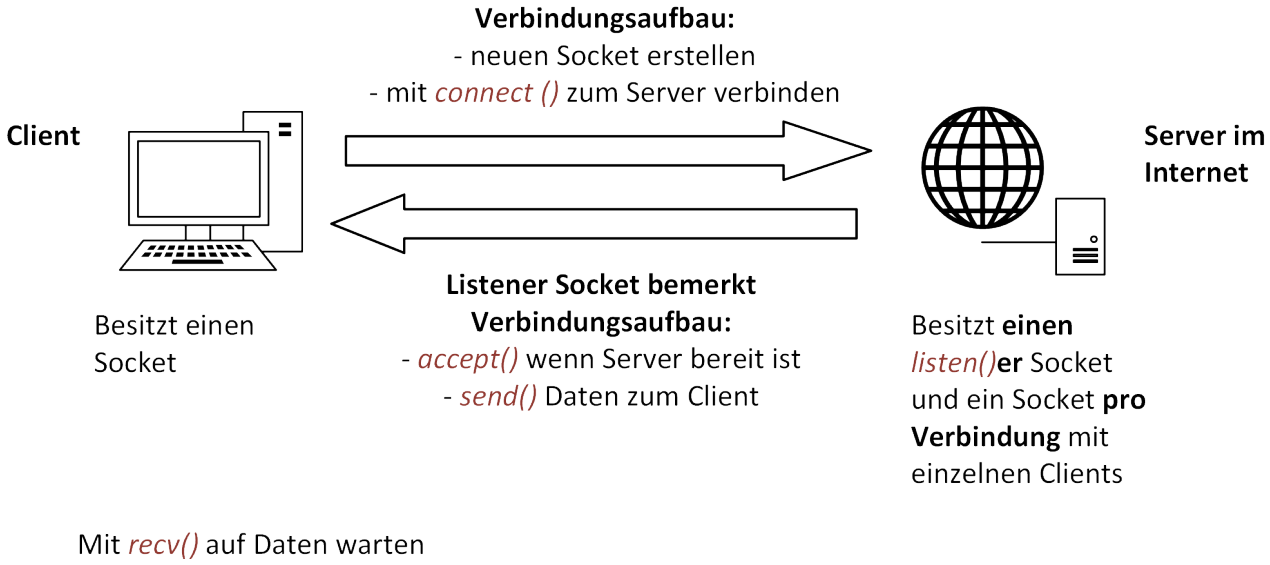

Abbildung 3.1: Verbindungsaufbau mit Libc

4 GLib Verbindungsaufbau

In diesem Kapitel wird der Verbindungsaufbau mit der GLib behandelt. Dies geht aber nicht sehr in die Tiefe und für weitere Informationen sollte man die GLib Dokumentation genauer anschauen.

4.1 Serverseite

Beginnen wir mit der Serverseite.

Listing 4.1: GLib Serververbindung Teil 1

```
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 (GSocketListener *listener,
11                                  guint16 port,
12                                  GObject *source_object,
13                                  GError **error);
```

Wichtig ist natürlich die Headerdateien einzubinden.
Dann die gLib sowie gError initialisieren mit NULL.
In der GLib erstellen wir dann einen sog. Socketservice.
Nachdem wir dies haben müssen wir auch ein Socket erstellen, der an einem bestimmten Port nach Verbindungen wartet (listener Socket)

Listing 4.2: GLib Serververbindung Teil 2

```
g_signal_connect(instance, detailed_signal,
                 c_handler, data)

g_socket_service_start (service);

gboolean
incoming_callback (GSocketService    *service,
                   GSocketConnection *connection,
                   GObject           *source_object,
                   gpointer           user_data)
```

Um eine Verbindung aufzubauen gibt es g_signal_connect()

- instance : Dort steht der Socketservice.
- detailed_ : Einen string mit Informationen zur eingehenden Verbindung
- c_handler : Die Funktion die abgerufen wird wenn jemand zum Server eine Verbindung herstellt.
- data : Daten, welche an die Funktion mitgesendet werden, beim Server eigentlich keine (NULL)

Nun sollte der Service gestartet werden.
Soabld eine Verbindung hergestellt wird, behandelt incoming_callback diese Verbindung.

4.2 Clientseite

Die Clientseite sieht logischerweise etwas anders aus:

Listing 4.3: GLib Clientverbindung Teil 1

```
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();
```

- Auch gLib initialisieren.
- Dann das GError-Objekt initialisieren.
- Neue Connection erstellen.
- Neuen Socket für einen Client erstellen mit Standardwerten.

Listing 4.4: GLib Clientverbindung Teil 2

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

Auch hier nochmal eine kurze Erklärung der Syntax.

- *client : Der eben erstellte GSocketClient
- *host_and_port : Der Name erklärt sich von selbst.
- default_port : Wenn kein Port angegeben wird dieser hier verwendet.
- *cancellable : Ob man abbrechen kann.
- **error : GError Objekt in dem Fehler gespeichert werden.

Der Input-Stream ist zum lesen da und der Output-Stream ist zum schreiben da.
Beide müssen jeweils mit der Connection initialisiert werden.

Natürlich sollten neue Variablen für die beiden Streams gewählt werden damit wir den Stream speichern können.

4.2.1 Verschiedene Funktionen

Es gibt noch verschiedene Funktionen die man mit der GLib nachdem eine Verbindung hergestellt wurde machen kann, wie z.B. in einem Stream etwas schreiben:

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

Diese Funktion schreibt Bytes von einem Buffer in den Output stream.

- * buffer : Die Daten
- count : Anzahl der Bytes die zu schreiben sind.
- cancellable : Ob die Aktion abbrechbar ist (NULL = abbrechbar).
- *error : Der GError für die Fehler.

Natürlich gibt es noch viele mehr, diese finden Sie unter anderem in der GLib Dokumentation.

Wichtig ist aber auch die Fehlerüberprüfung:

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

5 Zusammenfassung

- Verbindungsart muss gewählt werden, Stream oder Datagram socket
- TCP stellt eine gesicherte Verbindung her, vorzuziehen bei Downloads
- UDP stellt keine Verbindung her, Daten werden mit Adresse wie ein Brief ins Internet geschickt, diese Art ist schneller aber dafür nicht so sicher.
- getaddrinfo() wenn ohne GLib gearbeitet wird benutzen um Arbeit zu sparen.
- bind() um Port mit Programm zu verknüpfen, dies ist nur bei einem Server notwendig.
- Wesentlich schneller und einfacher ist allerdings eine Verbindung mit der GLib.
- Auch in meiner Ausarbeitung kann ich nicht alles behandeln, für weitere Fragen gucken Sie sich am besten meine Quellen an oder besonders bei der GLib, natürlich die Dokumentation.

6 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

Abbildungsverzeichnis

Listingverzeichnis