ingenieur
wissenschaften
htw saar

Hochschule für
Technik und Wirtschaft
des Saarlandes
University of
Applied Sciences

Dokumentation

Multiplexing in C mit select()

vorgelegt von

Alexander Huber 367065, B.Sc.

Saarbrücken, 23.09.2019

Inhaltsverzeichnis

1 Allgemeines

1.1 Grundlagen

Die Grundlage für das Multiplexing in C mit select() ist unter anderem dem Lebenszyklus und den Herausforderungen einer Server-Client-Lösung geschuldet:

- Server sehen sich oft mit der Belastung konfrontiert, Anfragen von Clients zeitkritisch zu beantworten.
- Steigende Belastung bei steigenden Anfragen, führen zu großen Wartezeiten bei einer normalen sequenziellen Server-Client-Lösung
- Andere Art der Prozessverwaltung in Beispielanwendungen wie Chat-Programmen oder Webservern um für geringe Latenz zu sorgen.

Aus diesem Grund gibt es mehrere alternative Konzepte um das Problem zu lösen. Diese sollen nun vorgestellt werden.

1.2 Einflüsse

Die Wartezeit bzw. der Grad der Zunahme von Nachrichtenlatenz einer Clientanwendung ist dabei von verschiedenen Faktoren abhängig. Einige sind z.B.:

- Konzept der Client-Server-Lösung
- Anzahl abzuarbeitender Nachrichten
- Programmiersprache
- Plattform

1.3 Ziel

Auf Basis der zur Verfügung stehenden Informationen soll eine Server-Client-Architektur entwickelt werden, welche die Nutzung der Funktion select() der Programmiersprache C mit der Thread-Klasse aus Modern C++ vergleicht.

2 Konzept

In diesem Kapitel soll die Architektur des Projekts vorgestellt werden. Dazu besteht es aus mehreren Anwendungen, einem Client und zwei Servern. Die beiden Serveranwendungen unterscheiden sich lediglich in der Art ihrer Prozessausführung. Während ein Server das Multiplexingverfahren in der Programmiersprache C mit der select()-Funktion verwendet, führt der andere Server dieselbe Aufgabe nur eben Multithreaded in der Programmiersprache C++ mit der thread-Klasse aus. Der Client wird dabei ebenfalls in C programmiert. Die beiden Server sollen jeweils Services anbieten, die unter anderem vom Client ausgeführt werden können. Wichtig ist, dass sich die Server nach außen hin gleich verhalten, weswegen sie auf gemeinsame Schnittstellen im System zugreifen. Die angebotenen Services sollen die Anwendungsmöglichkeiten mit Deskriptoren unter Beweis stellen. Für Interprocess Communication (IPC) wird ein Service mit einer Named Pipe bereit gestellt. Transmission Control Protocol (TCP) wird mittels eines Echo-Services veranschaulicht und User Datagram Protocol (UDP) soll anhand einer Babyphone-Service Implementation dargestellt werden.

2.1 Multiplexing Server

Der Multiplexing Server hat anders als der Multithreading Server lediglich einen Thread in Ausführung. Zu aller erst werden die jeweiligen Services mit den jeweils nötigen Parametern initialisiert und bei einem Fehler die Ausführung verlassen. Nach der erfolgreichen Initialisierungsphase ist es dem Server gestattet die Kommunikation über die jeweiligen File-Deskriptoren aufzunehmen. Dazu wird jeder in Ausführung befindliche Deskriptor einem Set hinzugefügt, welches durch die select() Funktion überwacht wird. Sollte nun eine I/O-Operation auf einem der im Set befindlichen Deskriptoren stattfinden, wird das select() darüber informiert und die Prozessausführung entsprechend der Implementierung umgelenkt. Folgende Abbildung 2.1 veranschaulicht vereinfacht die Idee hinter dem Ansatz des Multiplexings.

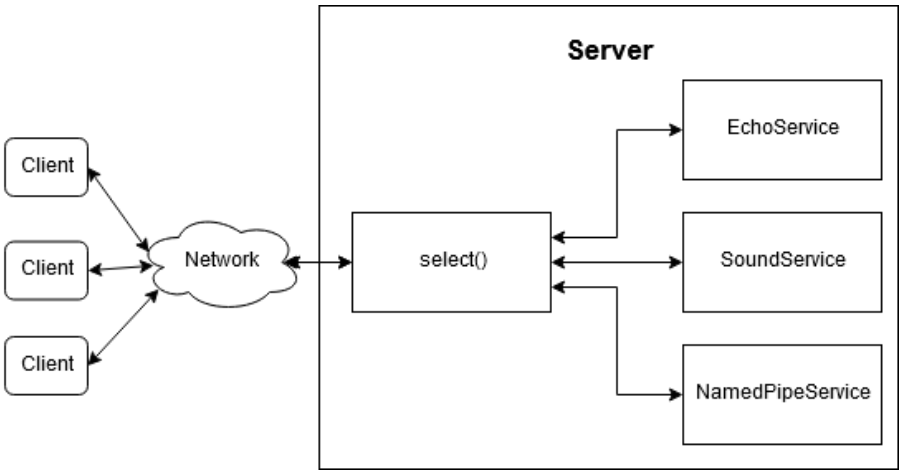

Abbildung 2.1: Konzept Prozessverteilung Multiplexing Server

2.2 Multithreading Server

Anders als der Multiplexing Server führt der Multithreading Server jeden Service in einem eigenen Thread aus. Bei drei Services werden somit mindestens 3 + main()-Threads erzeugt. Die Prozesse im Programm können also nun von mehreren Kernen in Ausführung sein und sind nicht mehr nur auf einen beschränkt. Dem Server stehen verschiedene Interaktionsmöglichkeiten im main()-Thread zur Verfügung um die Ausführung der Threads zu verändern. Folgende Abbildung 2.2 veranschaulicht das Threading-Konzept anschaulich vereinfacht.

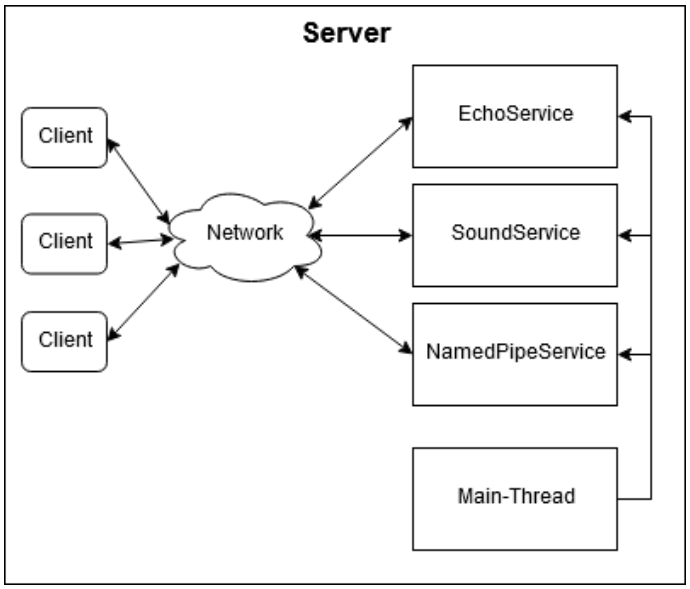

Abbildung 2.2: Konzept Prozessverteilung Multithreading Server

2.3 Client

Der Client dient als leichtgewichtige Kommunikationseinheit mit dem Server. Zwar ist es auch möglich ihn mit anderen Programmen wie beispielsweise Putty oder eigenen Implementierungen anzusprechen, gerade aber für den Sound Service ist eine gewisse Vorkonfiguration notwendig um akzeptable Ergebnisse zu liefern. Dies soll dem Anwender am besten erspart bleiben. So ist der Client einerseits befähigt UDP Pakete an den Server zu senden, als auch TCP Pakete über den Echo Service.

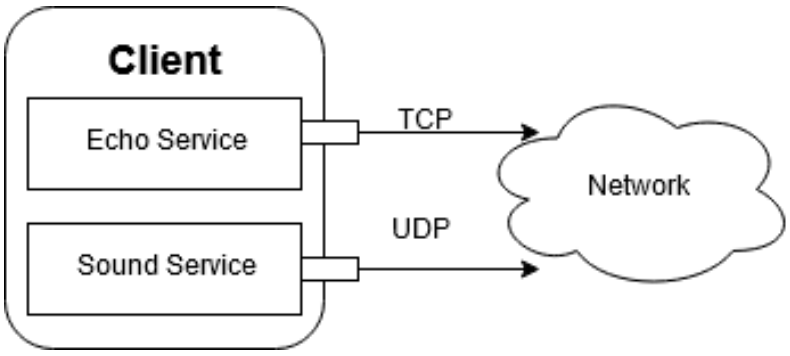

Abbildung 2.3: Client-Konzept

2.4 Services

Jeder Service wird im Projekt in einem eigenen Modul definiert. Dabei läuft er je nach Art verschiedene Phasen durch.

2.4.1 Echo Service

Der Echo Service dient als gängiges Client-Server Beispiel und veranschaulicht eine TCP Kommunikation über ein Netzwerk. Clients können sich über dieses Modul beim Server anmelden und Nachrichten über die TCP-Schnittstelle austauschen, welche vom Server verwaltet wird. Der Echo Service muss, da er über eine TCP-Verbindung läuft, zu aller erst eine Verbindungsanfrage annehmen, bestätigen und kann dann über die ihm zur Verfügung gestellte Schnittstelle mit dem Client kommunizieren. Bei Verbindungsabbrüchen muss der Server den Kanal ordnungsgemäß schließen. Das folgende Sequenzdiagramm 2.4 veranschaulicht dies.

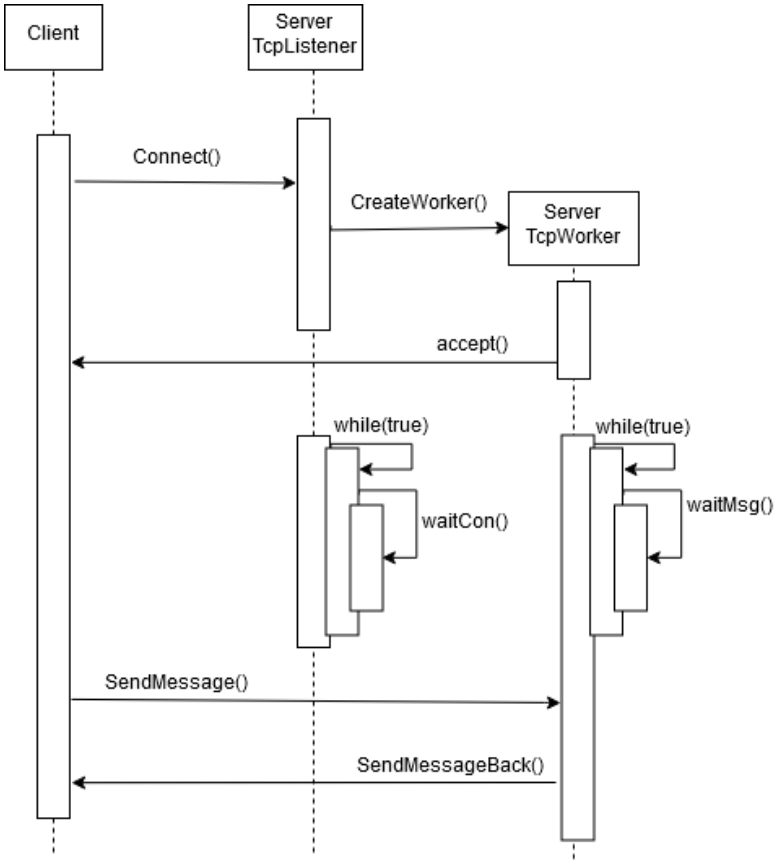

Abbildung 2.4: Sequenzdiagramm einer Verbindung und Nachrichtenübertragung

2.4.2 Sound Service

Der Sound Service ist die einfache Darstellung eines Kommunikationsbeispiels über UDP. Hierbei handelt es such um einen Audio-Streaming dienst, genauer gesagt ein Babyphone. Dieses nimmt Clientseitig einerseits Geräusche über ein Mikrofon auf und sendet diese mit UDP-Paketen über ein Netzwerk. An der Empfängerseite lauscht der Server und gibt die erhaltenen UDP-Pakete über den Lautsprecher aus. Folgende Abbildung 2.5 veranschaulicht die Kommunikation des Services.

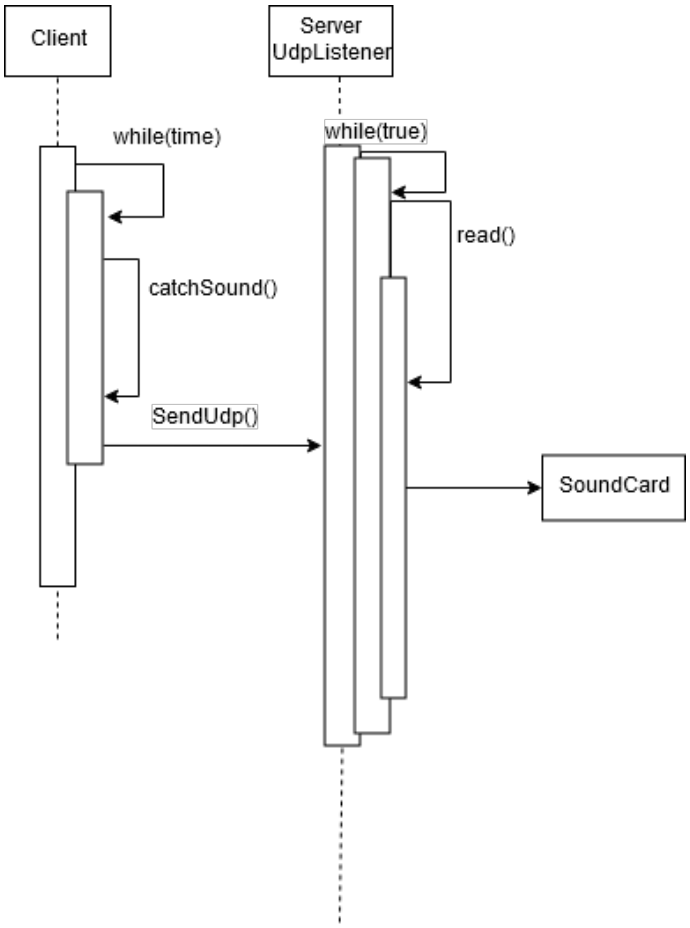

Abbildung 2.5: Sequenzdiagramm Soundhandle

2.4.3 Named Pipe Service

Der Named Pipe Service veranschaulicht IPC zwischen Prozessen innerhalb eines Rechners. So kann durch dieses Modul an einer beliebigen Stelle im System eine Named Pipe mit Rechten erstellt werden und auf diese lesend sowie schreibend zugegriffen werden. Der Server soll solange von der Pipe lesen, bis er aufgefordert wird schreibend darauf diese zugreift.

2.5 Zusätzliche Module

Zusätzliche Module werden benötigt um nicht immer dieselben Befehle in jedes Modul zu implementieren, so wird darauf geachtet, dass das SRP erhalten bleibt.

2.5.1 Sockets

Um der Anwendung eine gewisse Flexibilität zu geben werden die Funktionen zur Socket Generierung und Verwaltung in einem eigenständige Modul abstrahiert. Dieses ist zentral für alle Anwendungen, da diese jeweils Socketoperationen wie Create, Bind, Accept, Listen oder Close ausführen müssen.

3 Struktur der Algorithmen

Der Ablauf eines Servers verläuft in mehreren voneinander abhängigen Schritten.

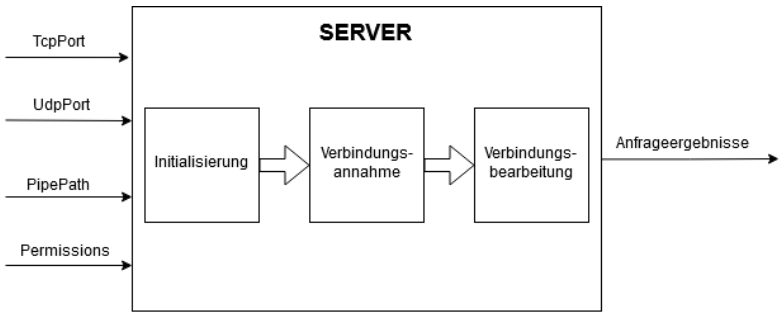

Abbildung 3.1: Generelle Funktionsweise des Servers

3.1 Parameter bzw. Kenndaten

Um den Server ausführen zu können, müssen folgende Parameter anwendungsspezifisch definiert werden.

Typ	Beschreibung	Datentyp
tcpPort	Listen Port für TCP Kommunikation	Integer
udpPort	Listen Port für UDP Kommunikation	Integer
path	Absoluter Pfad der Named Pipe	String
permission	Zugriffsrechte der Pipe	Integer

Tabelle 3.1: Grundlegende Parameter für den Server

3.2 Initialisierung

Zu aller erst werden die jeweiligen Sockets mit ihren zugehörigen Ports initialisiert. Darauffolgend wird die Pipe mit den Parametern path und permission erstellt. Zum Schluss wird die Soundkarte mittels verschiedener fest festgelegter Parameter eingerichtet. Sobald eine der Initialisierungsfunktionen einen negativen Wert ausgibt, wird das Programm abgebrochen. Der Server wird dann nicht ausgeführt. Dieser Fall tritt ein, wenn die angegeben Ports bereits belegt sind und das System keine Möglichkeit hat diese zu binden oder den TCP-Socket in den listen Modus zu versetzten. Außerdem wird das Programm abgebrochen, wenn es nicht in der Lage war, die Soundkarte richtig zu konfigurieren.

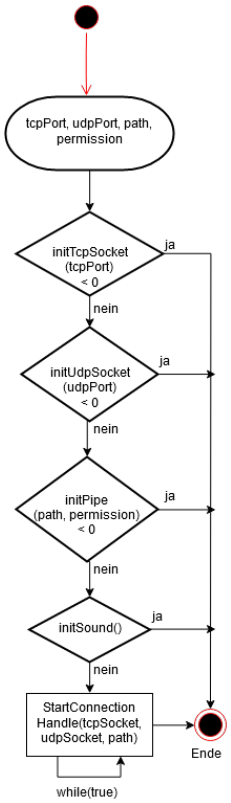

Abbildung 3.2: Initialisierungsphase des Servers zum vorbereiten für Kommunikationswünsche

3.3 Verbindungsannahme

Die Verbindungsannahme ist jeweils für die beiden Servertypen unterschiedlich und wird in ihren eigenem Abschnitt erläutert.

3.3.1 Multiplexing

Die Verbindungsannahme in dem Multiplexing Server läuft wie folgt ab. In jeder Schleifeniteration prüft FD_ISSET, ob eine I/O-Operation an dem jeweiligen Deskriptor vorhanden ist. Dabei wird zu aller erst der Deskriptor der stdin überprüft um mögliche User befehle entgegen zu nehmen. Der User hat dort zur Auswahl ob er das Programm beenden will, über die Named Pipe kommunizieren möchte oder ob eine Echo-Service Verbindung mit einem spezifischen Client abgebrochen werden soll. Ist keines davon der Fall läuft das Programm ungehindert weiter. Bei einem Verbindungswunsch über TCP bereitet der Server die Kommunikation so vor, das der Listener-Deskriptor das accept() an den nächst freien Worker-Deskriptor weiterleitet. Dieser übernimmt dann die Kommunikation mit dem jeweiligen Client und der Listener bleibt offen für den nächsten Verbindungswunsch. Sollten alle Worker belegt sein, wird keine weitere Verbindung mehr angenommen. Sollte eine I/O-Operation auf den beiden Deskriptoren udpListener und pipeListener passieren, so werden ihre jeweiligen handels ausgeführt. Der Sound Service gibt demnach die empfangenen UDP-Pakete auf den Lautsprechern aus und der Pipe Service schreibt den gelesenen Input auf der Pipe auf die stdout des Serverprozesses. Siehe dazu 3.3 und 3.4.

3.3.2 Multithreading

Anders als im Multiplexing Server werden hier gleich zu beginn des Connection Handles 3 Threads in Ausführung gestartet. Diese Threads bewerkstelligen jeweils für sich die Aufgabe, die im Multiplexing zuvor bei jedem FD_ISSET-Vergleich vorgenommen wurde. Der executeTcpThread() führt das handle auf den Echo Service aus. Er erstellt bei jeder neuen Client-Verbindung einen neuen Worker-Thread durch die executeHandleEchoServiceThread()-Funktion. Dieser Thread wird solange ausgeführt, bis das Programm beendet wird, der Thread vom main()-Thread geschlossen wird oder bis die Verbindung durch den Client geschlossen wird. Der executeTcpThread() wird beendet, wenn das Programm über die Konsole oder durch einen Abbruch beendet wird. Beim executeUdpThread() wird das Handel auf der Soundkarte solange ausgeführt, wie das Programm am laufen ist. Der executePipeThreadRead() ist für die Ausführung des Handles zum lesen auf der Pipe zuständig. Falls im main()-Thread das Kommando zum Schreiben auf die Pipe kommt, wird das Handle kurzzeitig unterbrochen und dann weiter ausgeführt. Die Kommandos an den Server werden im main()-Thread ausgeführt. Siehe dazu 3.5.

3.4 Verbindungsbearbeitung

Die Verarbeitung einer Verbindung findet in den jeweiligen Service-Modulen unabhängig von der gewählten Prozessverwaltung eines Client-Server Konzept ab.

3.4.1 Echo Service Handle

Das handle eines Echo Services wird mit der handleEchoService(socket)-Funktion ausgeführt. So erwartet er eine Nachricht an seinem Socket-Deskriptor. Bei erfolgreichem Eingang einer Nachricht wird diese auf ihren Inhalt geprüft. So kann der Client durch den Inhalt der Nachricht die Verbindung beenden oder die Nachricht verarbeiten. Bei der Nachrichtenverarbeitung wird die Empfangene Nachricht überarbeitet und wieder an den Sender zurück geschickt.

3.4.2 Sound Service Handle

Da UDP eine Verbindungslose Kommunikation ist, macht es nicht viel Sinn hier von einer Verbindungsverarbeitung zu sprechen, jedoch lauscht der Sound Service unentwegt auf seinem Socket in der handleSoundService(socket, pcmHandle)-Funktion. Bei Paketeingang wird dieses Paket dann an das vorkonfigurierte Handle der Soundkarte, das pcmHandle, übergeben und verarbeitet.

3.4.3 Named Pipe Handle

Die Named Pipe besitzt im Gegensatz zu den beiden anderen Services ein Handle zum Lesen und eines zum Schreiben. Das sind zum einen handleNamedPipeServiceWrite(worker, reader, path, messagem length) und zum anderen handleNamedPipeServiceRead(reader, path). Das Write-Handle kümmert sich um mögliche Read-Handle die auf der Pipe aktiv sind, schließt diese zuerst, schreibt dann und öffnet diese wieder. Das Read-Handle im Gegenzug prüft lediglich ob in der angegebenen Pipe ein Input vorgefallen ist.

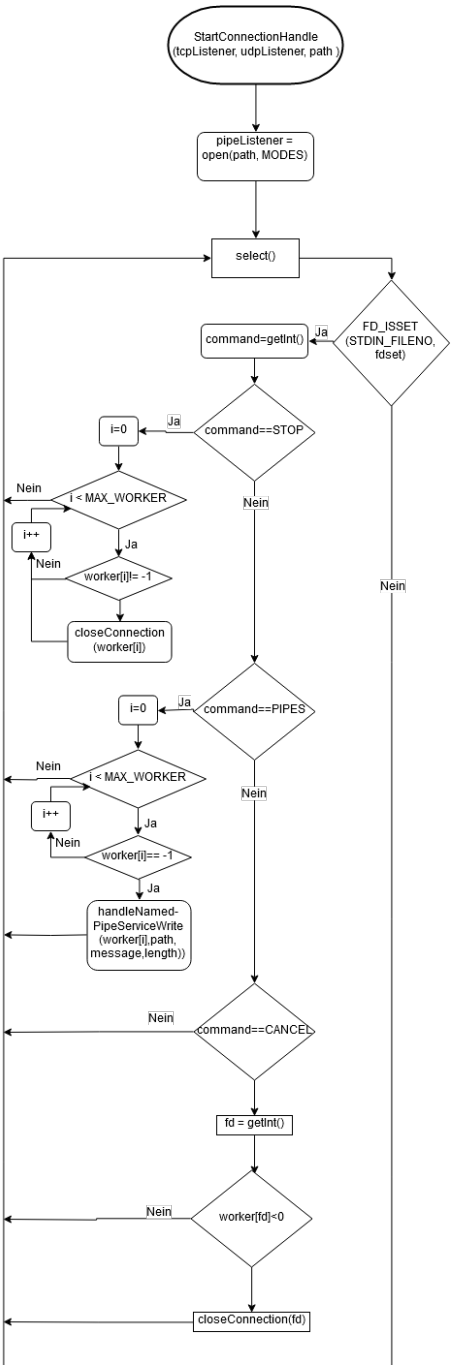

Abbildung 3.3: Verbindungsannahme Multiplexing Teil 1

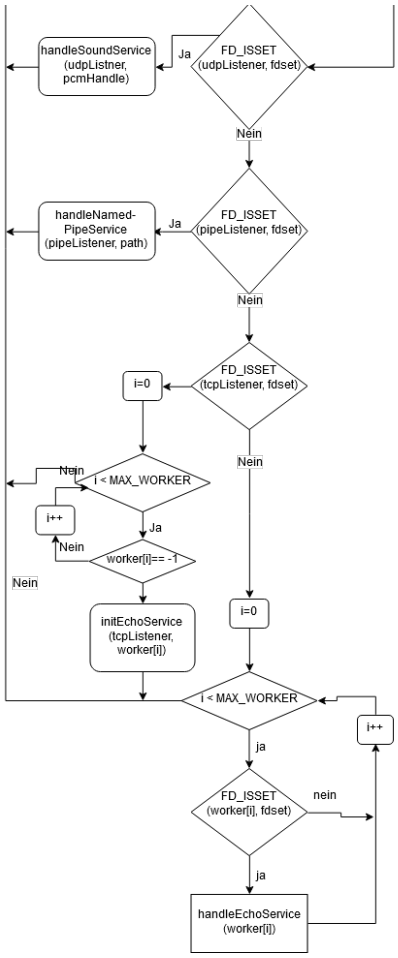

Abbildung 3.4: Verbindungsannahme Multiplexing Teil 2

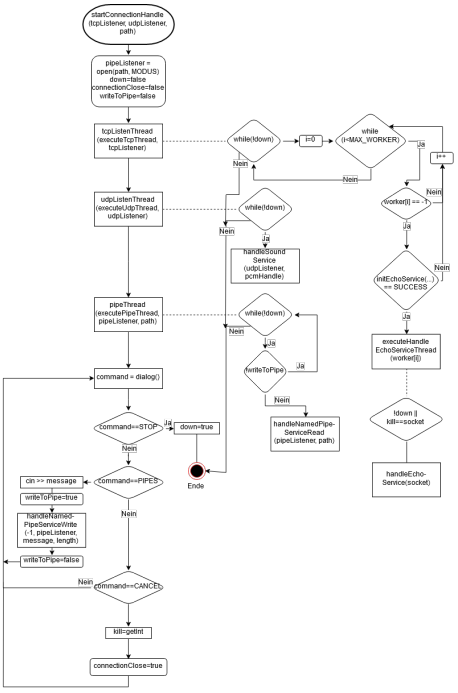

Abbildung 3.5: Verbindungsannahme Multithreading

4 Implementierung

4.1 Client

Der Client besitzt verschiedene Makros um den Dialog zu Steuern. In der Start()-Funktion wird wiederholt eine Dialoganfrage gemacht und die dialog()-Funktion aufgerufen. Der Rückgabewert der getInt()-Funktion ermittelt dabei den gewünschten Menüpunkt. So wird bei Null das Programm verlassen, bei eins eine Kommunikationsanfrage an den Echo Service über die echoServiceHandle()-Funktion aufgerufen und mit zwei der Sound Service ausgeführt.

Der Echo Service dauert so lange, bis entweder der Client oder der Server die Verbindung schließt. Dabei ist eine auf das Makro MAX_MESSAGE_SIZE an Bytes begrenzt. Sollte der Client nach 10 Sekunden immer noch keine Antwort vom Server erhalten haben, beendet er die Kommunikation.

Beim Sound Service wird noch zusätzlich die Dauer der Kommunikation in Sekunden festgelegt. Die setServerAddress()-Funktion setzt dabei die Parameter wie IP-Addresse und Port, welche für den Verbindungsaufruf gebraucht werden in den übergebenen sockaddr_in-Pointer.

Listing 4.1: Auszug aus der client.h-Datei

```
17 #define END  0
18 #define ECHOSERVER 1
19 #define SOUND 2
20 #define MAX_MESSAGE_SIZE 32
21
22 extern void start();
23 extern int dialog();
24 extern int getInt(const char* message);
25 extern void echoServiceHandle();
26 extern void soundServiceHandle();
27 extern void setServerAddress(struct sockaddr_in* server);
```

4.2 Multiplexing Server

Dem Multiplexing Server stehen definiert in seinem Makro MAX_WORKER bis zu 20 Deskriptoren zur TCP-Kommunikation zur Verfügung. Mit der definierten maximalen Puffergröße in Zeile 40 lassen sich Nachrichten der Größe 256 versenden.

Zeilen 47-50 zeigen die einzelnen init-Funktionen der Services. Nach ihrer Initialisierung lässt sich die handle-Funktion in Zeile 47 ohne Fehler ausführen. Diese beinhaltet die select()-Funktion mit der das Multiplexing vorgenommen werd.

Listing 4.2: Auszug aus der registry.h-Datei

```
28 #define MAX_WORKER 20
29 #define BUFF_SIZE 256
30 #define MAX_COMMAND_SIZE 64
31 #define STOP 1
32 #define PIPES 2
33 #define CANCEL 3
34
35 extern int startConnectionHandle(int* tcpListener, int* udpListener
       , const char* path);
36 extern int initTcpSocket(int port);
37 extern int initUdpSocket(int port);
38 extern int initPipe(const char*, int);
39 extern int initSound();
```

4.3 Multithreading Server

Der Multithreading-Server benutzt dieselbe Header-Datei zur Deklaration der Funktionen wie der Multiplexing-Server. Die Präprozessordirektiven fügen jedoch noch einige nötige Funktionen, Header und Erweiterungen hinzu.

Die wohl wichtigste Erweiterung dürften die Thread-Funktionen aus den Zeilen 42-46 sein. Diese Funktionen werden jeweils in einem einzelnen Thread ausgeführt. Wobei die executeTcpThread()-Funktion weitere executeHandleEchoServiceThread()-Funktionen aufrufen kann, bis wiederum das Maximum an möglichen Workern aus Zeile 39 erreicht ist.

Listing 4.3: Auszug aus der registry.h-Datei

```
42 extern void executeHandleEchoServiceThread(int& socket);
43 extern void executeTcpThread(int tcpListener);
44 extern void executeUdpThread(int udpListener);
45 extern void executePipeThreadRead(int& pipeListener,const char *
       path);
46 extern int getInt(const char* message);
```

4.4 Echo Service

Wie zu erwarten besitzt der Echo Service drei Funktionen. Mit der initEchoService()-Funktionen werden eingehende Kommunikationsanfragen angenommen.

Die Funktion handleEchoService() in Zeile 14 handelt unterdessen die Kommunikation bei schon bestehenden Verbindungen.

Die closeConnection-Funktion stellt sicher, dass eine Verbindung auch ordnungsgemäß geschlossen wird. Dazu fährt sie alle Teile der Kommunikation herunter und reinitialisiert den Socket, sodass erfür eine neue Kommunikation verwendet werden kann.

Listing 4.4: Auszug aus der echoService.h-Datei

```
11 #define BUFF_SIZE 256
12
13 extern int initEchoService(int* listener, int* worker, const char*
       message);
14 extern int handleEchoService(int* socket);
15 extern void closeConnection(int* socket);
```

4.5 Sound Service

Der Sound Service muss zu aller erst eine aufwendige Konfiguration des Soundkartentreibers vornehmen um das Gerät ordnungsgemäß einsetzen zu können. Die Funktion initSoundDevice() bewerkstelligt dies. Mit dem daraus resultierendem handel des Typs snd_pcm_t* lässt sich danach auf die Soundkarte zugreifen, je nach dem wie man diese Konfiguriert hat.

Die handleSoundService()-Funktion ist dann einfach nur noch mit dem empfangen der Daten und dem schreiben auf das Konfigurierte Handle beschäftigt.

Die closeSoundService()-Funktion schließt am ende das konfigurierte Gerät ordnungsgemäß.

Listing 4.5: Auszug aus der soundService.h-Datei

```
8  extern int initSoundDevice(snd_pcm_t** handle, const char*
       deviceName, int channels, unsigned int * sampleRate,
       snd_pcm_stream_t stream,
9                                  snd_pcm_format_t format,
                                      snd_pcm_access_t mode,
10                                 snd_pcm_uframes_t frames);
11 extern int handleSoundService(int socket, snd_pcm_t* pcmHandle);
12 extern void closeSoundService(snd_pcm_t* pcmHandle);
```

4.6 Named Pipe Service

Auch der Named Pipe Service besitzt eine Initialisierungsmethode, die initNamedPipe()-Funktion. Sie erstellt falls möglich eine neue Named Pipe mit den zugewiesenen Rechten auf dem System. Bei möglichen Fehlern die beim Anlegen dieser Dateiëntstehen können werden unterschiedliche Fehlercodes im Fehlerfall ausgegeben.

Die beiden Funktionen handleNamedPipeServiceRead() und handleNamedPipeServiceWrite() sind die I/O-Funktionen die auf der Pipe ausgeführt werden. während die Read-Funktion auf der Pipe ließt, schließt die Write-Funktion mögliche Reader und schreibt auf die Pipe.

Listing 4.6: Auszug aus der pipeService.h-Datei

```
13 #define MAX_BUFF_SIZE 256
14
15 extern int initNamedPipe(const char* path, mode_t mode);
16 extern void handleNamedPipeServiceRead(int* reader, const char*
       path);
17 extern void handleNamedPipeServiceWrite(int worker,int* reader,
       const char* path, const char* message, int length);
```

4.7 Socket

Das Socket Modul erleichtert die Initialisierung von Sockets auf dem System, indem es die Systemcalls aus der sockets.h libary abstrahiert und erweitert. So kann unter anderem in der acceptSocket()-Funktion ein Deskriptor den Wunsch einer Verbindung auf einen anderen Deskriptor weiterleiten und dann die Verbindungsinformationen ausgeben.

Listing 4.7: Auszug aus der linuxsocket.h-Datei

```
14 extern int createSocket(int family, int type, int protocol);
15 extern int listenSocket(int* socket);
16 extern int bindSocket(int* socket, unsigned int address, int port);
17 extern int acceptSocket(int* listener, int* worker);
18 extern int closeSocket(int* socket);
```

5 Beobachtungen

Nun zu den Unterschieden und Beobachtungen die gemacht wurden als beide Serveranwendungen getestet wurden.

5.1 Performance

Auslastungsüberwachung mit dem Programm htop haben den Verbrauch an Rechenkapazität der beiden fast identischen Anwendungen gezeigt. Die Bilder in Abbildung 5.1 und 5.2 zeigen, dass das Thread-Programm eine deutlich höhere Auslastung im Leerlauf besitzt und kontinuierlich die Ressourcen der Anderen Kerne ausnutzt. Die Anzeigen in grüner Farbe sind die in Ausführung befindlichen Threads, des in weiß angezeigten Prozesses.

Das die Auslastung bei Threads nicht unbedingt etwas damit zu tun haben muss, dass Threads deutlich Ressourcen hungriger sind als ihr Multiplexing Pendant kann auch einfach der Implementierung geschuldet sein. Diese ist nach aller Wahrscheinlichkeit nicht optimal gewählt worden und verbraucht demnach deutlich mehr Rechenleistung. Dies kann zum Beispiel der Designentscheidung geschuldet sein, dass die Deskriptoren in der Thread Variante nicht-blockierend sind. Dabei entsteht durch ständiges überprüfen in mehreren Threads ein deutlich größerer Overhead, ist aber von Nöten damit Threads auf externe Signale des main()-Threads reagieren können und nicht in einer Read- oder Write-funktion stecken bleiben.

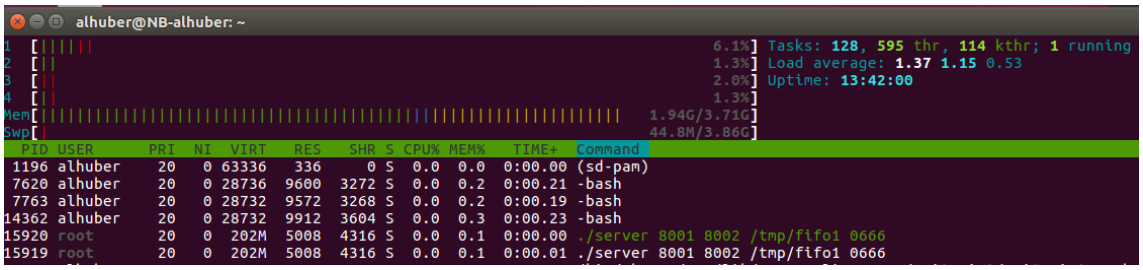

Abbildung 5.1: Prozessüberwachung Multiplexing mit htop

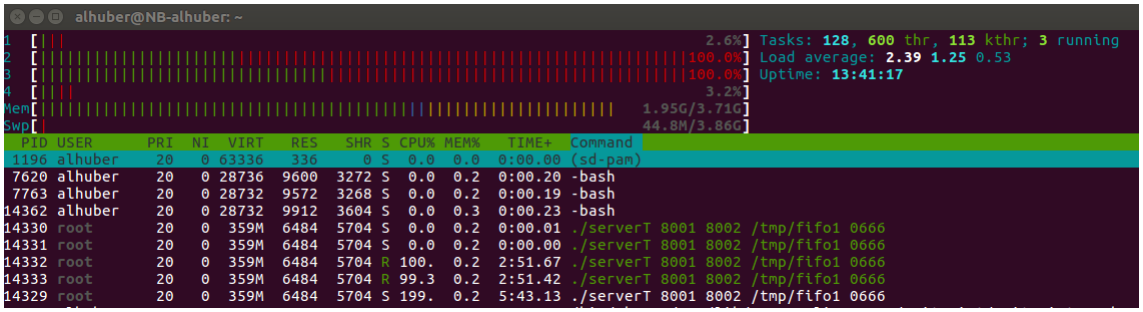

Abbildung 5.2: Prozessüberwachung Multithreading mit htop

5.2 Ausführungszeit

Eindeutig sind jedoch die Ergebnisse zumindest in der Ausführungszeit der Anfragebearbeitungen über den Echo Service. Wie in der Abbildung 5.3 zu sehen ist, benötigt die Thread-Variante eine um den Faktor 9,4 längere Zeit um dieselbe Anfrage zu verarbeiten. So benötigt die Thread-Variante im Durchschnitt 0,00084 Sekunden während das Multiplexing-Verfahren 0,00009 Sekunden benötigt.

Threads	Multiplexing	
0,000109	0,000114	
0,000912	0,000091	
0,000916	0,00009	
0,000895	0,000104	
0,000945	0,000106	
0,000964	0,00009	
0,000945	0,000108	
0,000879	0,0001	
0,000781	0,0001	
0,000901	0,000104	
0,000918	0,000084	
0,000997	0,000091	
0,000927	0,000092	
0,000856	0,00009	
0,000881	0,000084	
0,000833	0,000098	
0,00096	0,000098	
0,000719	0,0001	
0,000835	0,00002	
0,000817	0,000037	
0,0008495	0,00009005	**Mittelwert**
850 Mikrosekunden	90 Mikrosekunden	Faktor 9,4

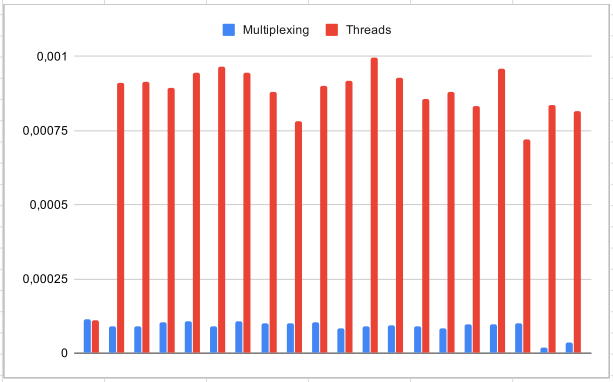

Abbildung 5.3: Vergleich Prozessausführung Thread und Multiplexing

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

IPC Interprocess Communication

UDP User Datagram Protocol

SRP Single Responsibility Principle

TCP Transmission Control Protocol