ASCII-Transfer

Paul Storch

10.01.2021

Inhaltsverzeichnis

1 Thematik

Mittels einem Client und Server Programm sollen ASCII-Zeichen von dem Server an den Client geschickter werden. Diese werden mittels einer zweidimensionaler Parität versehen, sodass eine Fehlererkennung möglich ist. Die beim Client ankommenden binäre Blöcke werden dann mittels future und promis Paar übergeben und eine Fehlererkennung durchzogen. Wenn diese fehlerfrei ist, werden die Blöcke wieder in ASCII-Zeichen umgewandelt.

2 Server-Client

2.1 Server

Für die Server-Client-Kommunikation wurde "asio.hpp" verwendet. Mithilfe dieser Bibliothek wurde ein einfacher Server erstellt der standardmäßig auf Port "8888" horcht und nach dem erfolgreichen Aufbau einer Verbindung die gewünschten Daten übermittelt.

```
 if (strm){
    spdlog::log(spdlog::level::level_enum::info, "Sending ASCII-Block to client!");
...
} else {
    spdlog::log(spdlog::level::level_enum::err, strm.error().message());
    spdlog::log(spdlog::level::level_enum::err,
    "Error while establishing connection with client!");
```

Falls dies nicht geschieht, wird je nach Situation eine entsprechende Fehlermeldung ausgegeben. Leider war er nicht möglich den Server in einen eigenen Thread zu geben da werden der Laufzeit immer die Fehlermeldung "Core Dumped" kam und eine Online Recherche nichts ergab.

2.2 Client

```
tcp::iostream strm{ipaddress, port};
```

Auf der Client Seite gibt er die Möglichkeit eine IPv4 Adresse und ein Port zu konfigurieren die dann für den Verbindungsaufbau verwendet werden. Falls nichts ausgewählt wird, werden die Standards "localhost" und der Port "8888" verwendet. Wie beim Server wird bei Erfolgreicher sowie bei erfolgloser Verbindung die entsprechende Information order Error Meldungen ausgegeben. Bei erfolgreicher Verbindung erhält der Client eine variierende Anzahl an binäre 80-Bit große Blöcke.

3 Parität

3.1 Eindimensionale Parität

Der ASCII-Text wird zunächst in Bits umgewandelt. Dazu wurde "Bitset" verwendet. Die 7-Bit ASCII-Zeichen haben zunächst durch die eindimensionale Parität ein weiteres Bit erhalten.

3.2 Code-Beispiel Eindimensionale Parität

```
vector<bitset<8>> onedParitaet{};
for(auto bits: asciibits){
    int countOnes{0};
    for(auto bit: bits.to_string()){
        if(bit == '1'){
            countOnes++;
        }
    }
    if(countOnes % 2 == 1){ **
        bitset<8> temp(bits.to_string() + bitset<1>(0x1).to_string());
        onedParitaet.push_back(temp);
    }else {
        bitset<8> temp(bits.to_string() + bitset<1>(0x0).to_string());
        onedParitaet.push_back(temp);
    }
}
**Falls die Summe der positiven Bits ungerade ist,
wird diese durch den Algorithmus mit einem positiven Bit zu einer geraden Summe gebracht.
```

3.3 Zweidimensionaler Parität

Um dies dann auf die Zweidimensionale Parität zu erweitern werden je 9 mit eindimensionale Parität versehenen ASCII-Zeichen zusammengefasst und mittels einem Algorithmus 8-Bits hinzugefügt. Dies Ermöglicht dann dem Client mittels eines Algorithmus eine Fehlererkennung durchzuführen.

Zweidimensionale Parität:

0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1
1	0	1	0	0	1	0	1
0	0	1	0	0	1	0	0

Mithilfe

der zweidimensionalen Parität können Bitfehler von 1-3 Bits sowie die meisten 4-Bitfehler erkannt werden.

4 future und promis Paar

4.1 Code-Beispiel future/promis

```
while(true){
    message.push_back(async(launch::async, getStream, ref(strm)));
    status = message.at(counter).wait_for(std::chrono::seconds(2));
    if(status == std::future_status::ready){
        spdlog::log(spdlog::level::level_enum::info,
                    "Receiving ASCII-Block: " + to_string(counter));
        counter++;
    } else {
        break;
    }
}
```

Durch das future und promis Paar konnte man die Übertragung der binären Blöcke elegant lösen. Mittels der

```
wait_for()
```

Funktion ist es möglich nach einer 4 Sekunden andauernden Sendepause den Stream zu beenden ohne zusätzlicher Kommunikation.

5 Verwendung

5.1 Build

Für die korrekte Erstellung braucht man Meson. Erstellen Sie zuerst im Projektordner ein Ordner mit den Namen “Build”. Danach gehen sie in den Ordner und geben sie “meson” ein. Nachdem das Kommando ausgeführt wurde geben sie “ninja” ein:

```
mkdir build
cd build
meson
ninja
```

5.2 Grundlegende Verwendung

Nach dem Korrekten erstellen sollten die Dateien “server” und “client” in dem Build-Ordner sein. Diese können dann einfach gestartet werden wobei man darauf achten sollte das Serverprogramm zuerst zu starten.

Starten des Servers:

```
./server
```

Starten des Clients:

```
./client
```

5.3 Erweiterte Verwendung

Starten des Servers mit eigenem ASCII-Text

```
./server das ist mein ASCII-Text
```

Falls der Client auf einen Server zugreifen muss, der nicht lokal gehostet wird, kann man mit -i oder mit –ipaddress eine IPv4-Adresse angeben.

```
.client -i 1.1.1.1
```

Alle weiteren funktionen findet man, wenn man –help eingibt:

```
./server --help
./client --help
```