

NVS- Projekt 2020/21, 5AHIF

# C++ Networking Sliding Window Maurice Putz



11. April 2021

## Inhaltsangabe

<b>1</b>	<b>Beschreibung</b>	<b>1</b>
1.1	Sliding-Window Algorithmus Zusammenfassung . . . . .	1
<b>2</b>	<b>Bedienanleitung</b>	<b>3</b>
2.1	Befehle . . . . .	4
<b>3</b>	<b>Programmumsetzung</b>	<b>7</b>
3.1	Umsetzung Client . . . . .	8
3.2	Umsetzung Server . . . . .	9
<b>4</b>	<b>Verwendete Externe Klassen</b>	<b>11</b>
<b>5</b>	<b>Quellen</b>	<b>11</b>

# 1 Beschreibung

Das Programm ConnectSim dient dazu, eine Übertragung von Daten mittels einem Sliding Window Algorithmus darzustellen. Weiters bietet das Programm noch die Funktionen, drei Fehlerarten im Networking darzustellen. Diese drei Fehler wären: Paketverlust, Paketverfälschung (Datenmanipulation) und Fehler bei der Paketreihenfolge. Vom Sender aus werden zufällige ASCII-Zeichen, deren Menge optional angegeben werden kann, in derer dezimalen Darstellung, mittels einer Verbindung zu einem Server übertragen. Diese Verbindung ist vordefiniert. Der Server wird vor der Ausführung des Programms manuell vom User gestartet, dieser wartet dann auf einkommende Verbindungsanfragen und startet den Algorithmus bei erfolgreicher Verbindung.

Die Größe des Sliding Window Fensters beträgt Standardgemäß eins, ist aber per Kommandozeile vom User aus konfigurierbar. Die Daten werden vom Client zum Server übertragen. Zuerst wird die Anzahl, aller zu Datenframes, dann die Größe des Sliding Window Fensters dem Server mitgeteilt. Danach beginnt die eigentliche Übertragung der Daten. Der Server nimmt ein Datenframe, addiert den Wert zu einer Checksumme, welche pro Fenster berechnet wird und sendet die Daten, als AKN zurück. Wurde die maximale Anzahl der Daten entgegengenommen (die Anzahl ist gleich der Größe des Sliding Windows), sendet der Server einen AKN an den Client mit der gesamten Checksumme des Fensters und alles beginnt wieder von vorne, bis alle Daten auf diese Art übertragen worden sind.

Am Ende überträgt der Server nochmals die maximale Summer der zu Übertragenden Daten pro Fenster zum Client. Die Summe hat sich der Client selbst vor dem Abschicken der Daten gebildet. Diese wird nun verglichen ob sie dieselbe ist.

Das Programm gibt diesen Vorgang in einer Logdatei für den Server, und einer Logdatei für den Client aus. Der Name und Pfad der Logdatei für den Client ist frei konfigurierbar. Es gibt noch einige weitere Funktionen, wie auch die Simulation von Netzwerkfehlern. Diese werden im Abschnitt "Bedienanleitung", genauer beschrieben.

## 1.1 Sliding-Window Algorithmus Zusammenfassung

Das Sliding Window Protokoll wird für die Übertragung von Informationen benutzt. Eine Beschreibung ist dazu im RFC 1180 zu TCP/IP unter Absatz 7 TCP (Transmission Control Protocol). Im Grunde genommen bewirkt dieser Algorithmus, dass ein Client mehrere Datenframes auf einmal versenden kann, ohne das er auf ein AKN des Servers warten muss. Eine genauere grafische Darstellung, wie das Fenster über die Daten "schiebt", zeigen folgende Abb. 1 bis inklusive Abb. 3.

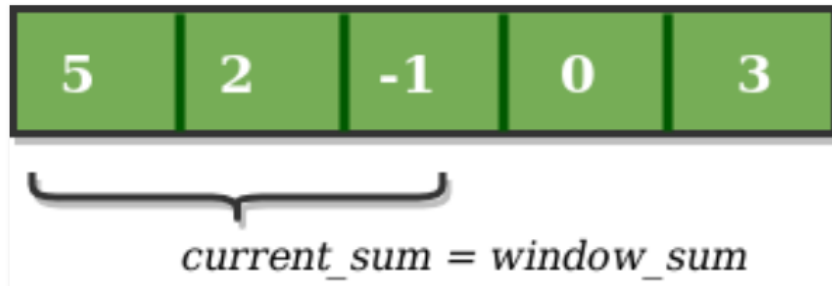


Abb. 1: Ausgangsposition mit Fenstergröße 3

Das Fenster hat die immer die Größe 3, die Summe aller Daten im dargestellten Fenster beträgt 6. Diese Zwischensumme wird in einer Variable `max_sum` zwischengespeichert, nämlich setzen wir diese gleich dem Wert in `current_sum`, dies ist die Summe welche 6 beträgt. Weiter geht es mit dem Schritt in Abbildung 2.

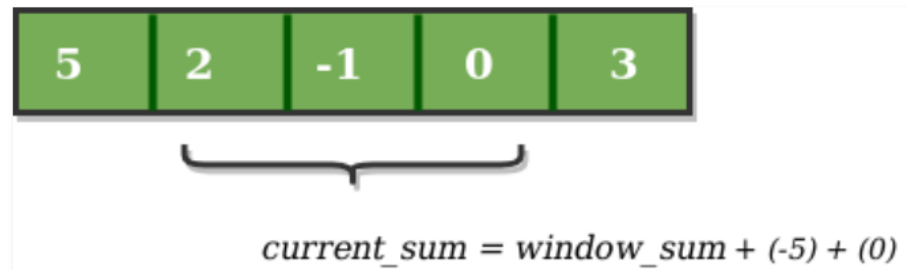


Abb. 2: Sliding Window um einen Index weitergewandert

Nun wischt das Fenster um einen Index weiter nach rechts, in Abb. 2 zu sehen. Der Wert aller Datenframes in dem Bereich des Fensters beträgt nun insgesamt 1. Dies ist kleiner als der Wert in `max_sum`. Daher wird der Wert nicht geupdated und das Fenster wischt nochmals weiter, in Abb. 3 zu sehen.

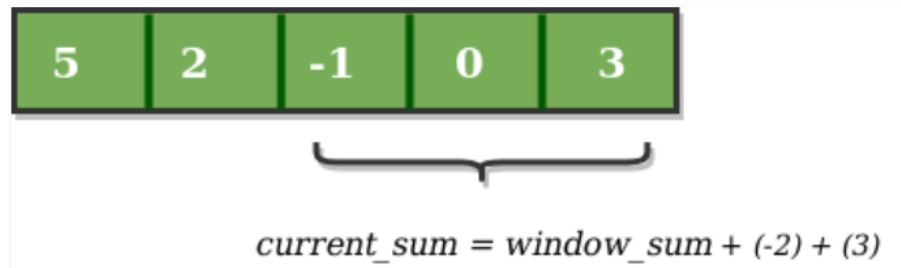


Abb. 3: Sliding Window um einen Index weitergewandert

Nachdem das Fenster wieder um einen Index weitergewandert ist, wird abermals die Summe des Fenster ermittelt, in Abb. 3 beträgt die Fenstersumme 2. Nun wird die Summe wieder mit dem Wert in der Variable `max_sum` verglichen. Da der Wert wieder kleiner ist, als der in `max_sum` gespeicherte, bleibt `max_sum` wie sie ist und unsere maximale Fenstergröße von unserem gesamten Array ist 6.

## 2 Bedienung

```
ConnectSim
Usage: ./connectsim [OPTIONS] [input_characters]

Positionals:
  input_characters TEXT    Given characters will be random times send to server    Example: "./connectsim asdf"

Options:
  -h, --help                Print this help message and exit
  -w, --windowsize TEXT:POSITIVE
                           Given number will be the size of the window of sliding window algorithm used for data transmission
Example: "./connectsim -w 3"
  -s, --set_logpath TEXT    Change name of logfile or local path for client    Example: "./connectsim -s logging/log_cs.txt
  -p, --packageloss          Simulates Package loss while sending data to server and otherwise
  -d, --datamanipulation     Simulates manipulation of some data packages.
  -r, --packagerow           Simulates behaviour if sended data have wrong order.
  -a, --allowed              Show allowed character for input
  -l, --logpath              Returns path of logfile with details on console
```

Abb. 4: Übersicht über die Komandozeilenparameter. Befehl: `connectsim -h`

Option **-h, --help**:

Zeigt die Hilfe für das Programm an, liefert das Menü wie in Abb. 4 zurück.

- Eingabe: `./connectsim -h`
- Ausgabe: [Siehe Abb. 4]

## 2.1 Befehle

Der Server muss unabhängig vom Programm **zuerst** gestartet werden und manuell vom User. Dies macht man mittels dem Befehl `./server` in dem Verzeichnis, wo die Datei zum starten gelegen hat. Eine Abfolge und das Ergebnis sieht man in Abb. 5.

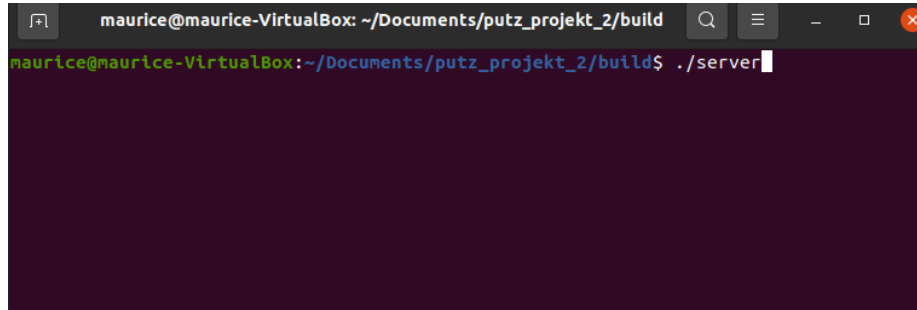


Abb. 5: Startbefehl des Servers

Durch den normalen Aufruf des Programms mit `./connectsim`, wird ConnectSim ohne speziellen Funktionen gestartet. Die Standardgröße eines Sliding Window Fensters beträgt eins, welche bei oben gezeigtem Aufruf benutzt wird, die Standardgröße von eins hat auch einen bestimmten Grund, welcher später noch genauer erläutert wird. Das Programm generiert nun zufällig oft zufällig viele (zwischen 1 und 127) ASCII-Character und beginnt die Übertragung. Der Server verarbeitet diese und antwortet. Genauer in den Abb.6-7 zu sehen. Ebenfalls werden entsprechende Ausgaben auf der Konsole der Servers getätigt und in den Logfiles des Servers und des Clients geschrieben.

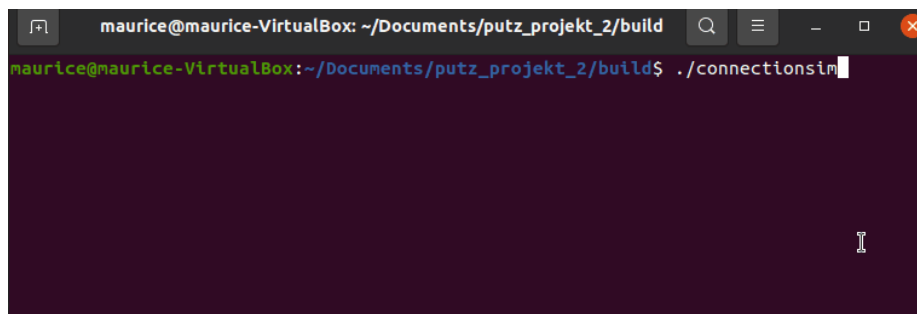
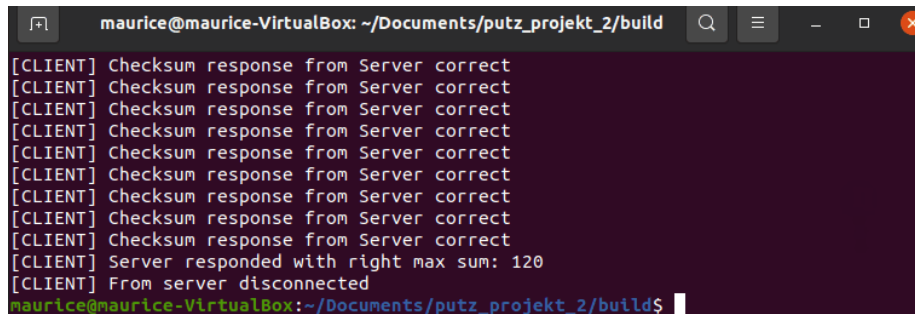


Abb. 6: Startbefehl des Programms



```
maurice@maurice-VirtualBox: ~/Documents/putz_projekt_2/build
[CLIENT] Checksum response from Server correct
[CLIENT] Checksum response from Server correct
[CLIENT] Checksum response from Server correct
[CLIENT] Checksum response from Server correct
[CLIENT] Checksum response from Server correct
[CLIENT] Checksum response from Server correct
[CLIENT] Checksum response from Server correct
[CLIENT] Checksum response from Server correct
[CLIENT] Checksum response from Server correct
[CLIENT] Server responded with right max sum: 120
[CLIENT] From server disconnected
maurice@maurice-VirtualBox:~/Documents/putz_projekt_2/build$
```

Abb. 7: Ausgabe von ./connectsim

Wenn man beim Aufruf des Programms, wie zum Beispiel `./connectsim asdf` eingibt, werden die einzelnen Zeichen hinter dem Programmnamen überprüft, ob sie in ASCII enthalten sind und im positiven Fall, wird das Programm genau wie bei einem einfachen Aufruf gestartet. Nur diesmal werden bloß die ASCII-Zeichen, welche hinter dem Programmnamen stehen zufällig oft (zwischen 1 und 127 mal) übertragen.

- Eingabe: `./connectsim asdf`
- Ausgabe: Wie bei Abb. 7

Option **-w,--windowsize TEXT:POSITIVE:**

Startet das Programm und ersetzt die default-Sliding Window Größe auf den angegebenen Wert. Das Programm beendet sich, falls diese Zahl größer ist, als die Anzahl der zu Übertragenden Elemente. Ist dies der Fall, wird eine Nachricht ausgegeben, wie der User in dem nun weiter verfahren sollte. Zum Beispiel, falls der User eine Größe von 8 für das Fenster eingibt, aber durch das zufallsbedingte auswählen der ASCII Charaktere, nur 5 dieser erstellt werden, muss dieser Fall abgefangen werden, da die Startbedingungen für den Algorithmus nicht korrekt sind.

Erwartet eine positive ganze Zahl als Eingabe. Beispiel:

- Eingabe: `./connectsim -w 3`
- Ausgabe: [Programm startet und Vorgang wird auf der Konsole und in den Logdateien festgehalten]

Flag **-s,--set\_logpath TEXT**:

Setzt den aktuellen Pfad für die Logdatei des Clients, zu dem Angegebenen. Der Pfad wird in der Datei **settings.json**, abgespeichert. Erwartet einen gültigen Pfad/String als Eingabe. Beispiel:

- Eingabe: `./connectsim -s /new_folder/client_log.txt`
- Ausgabe: Path of Logging file successfully changed!

Flag **-p,--packageloss**:

Simuliert einen Paketverlust. Beim Senden vom Client zum Server Beispiel:

- Eingabe: `./connectsim -p`
- Ausgabe: [SERVER] Received package count does not match with received count of data packages to process.  
Please look into log files.

Flag **-d,--datamanipulation**:

Simuliert einen Paketverlust. Beim senden von Daten vom Client zum Server. Beispiel:

- Eingabe: `./connectsim -d`
- Ausgabe: [SERVER] Received package count does not match with received count of data packages to process.  
Please look into log files.

Flag **-r,--packagerow**:

Simuliert, dass die Reihenfolge der geschickten Packages, falsch ankommt. Beispiel:

- Eingabe: `./connectsim -r`
- Ausgabe: [CLIENT] Server responded with right max sum: [MAX\_SUM of ASCII Characters]  
oder es wird ein "std::system\_error" geworfen und der Server schließt den Socket, so können auf dem Socket keine Daten mehr gesendet oder empfangen werden.



Flag **-a,--allowed**:

Damit kann man alle erlaubten ASCII Zeichen für das Programm sich anzeigen lassen. Einige ASCII Zeichen wie zum Beispiel 1 - SOH (start of heading), in C++ nicht über die Kommandozeile gewünscht verarbeitet werden kann. Alle Zahlen und Buchstaben und die meisten welche in ASCII enthalten sind, werden unterstützt.

- Eingabe: `./connectsim -a`
- Ausgabe: Tabelle, wo auf der linken Seite Dezimal-Werte stehen und auf der rechten Seite die jeweiligen zugehörigen ASCII Zeichen.

Flag **-l,--logpath**:

Führt einen Kommandozeilenbefehl **readlink -f [LOGFILE - FILENAME]** im Hintergrund aus und liefert den absoluten Pfad der Lockdatei für den Client zurück.

- Eingabe: `./connectsim -a`
- Ausgabe: Tabelle, wo auf der linken Seite Dezimal-Werte stehen und auf der rechten Seite die jeweiligen zugehörigen ASCII Zeichen.

### 3 Programmumsetzung

Das Schreiben und lesen des Clients und des Servers, sind jeweils in zwei identischen Funktionen realisiert. Für das Schreiben, wurde die Funktion **asio::write** benutzt. Diese Funktion erwartet sich eine Referenz auf ein Socket und eine Nachricht. Beim senden mit der **write** Funktion von asio muss die mitgelieferte Nachricht eine const variable sein:

---

```
1 void send_data( asio::ip::tcp::socket& socket , const string message )
```

---

Das lesen wurde mittels der asio Funktion **asio::read\_until()** realisiert. Diese erwartet sich eine Referenz auf ein Socket, einen streambuffer und das Zeichen, bei welchem die Funktion aufhört zu lesen, bis zu diesem Zeichen liest diese aus dem Stream dann ein. In diesem Fall hier, bis zum Zeichen `/n` also einem Linebreak.

---

```
1 void receive_data( asio::ip::tcp::socket& socket )
```

---

Den MAX\_VALUE welcher für den Sliding Window Algorithmus essentiell ist, wurde mittels folgender Funktion realisiert:

---

```
1 void max_sum( vector<char> ascii_vec , int window_size , int size )
```

---

Diese Funktion bekommt den Vektor mit den zu übertragenden Daten, die Länge des Sliding Windows und die Anzahl aller Elemente im Vektor mit. Er liefert

die größte maximale Summe eines Fenster zurück, oder -1, falls das Sliding Window Fenster größer ist, als die Anzahl der ASCII Elemente.

### 3.1 Umsetzung Client

Zuerst wurde ein objektorientierter Ansatz verwendet, deswegen sind in früheren commits des Repository, noch .h Dateien und Klassen zu sehen. Diese wurden aufgrund unnötiger Komplexität und fehlenden Mehrwert im Bezug auf Aufwand verworfen.

Nach dem Parsen wird der ein **basic\_logger** von der Klasse **spdlog**, mit dem Pfad aus der Datei **settings.json**, erstellt.

Danach wird, bei einem normalen Start des System, ohne mit angegebenen Flags, ein Vektor erstellt, welcher den Typ **char**, speichern kann, es wurde ein Vektor aufgrund der Einfachheit des Datentyps gewählt und aufgrund persönlicher Präferenzen. Der Vektor wird mit der zufälligen Anzahl an zu Übertragenden Daten befüllt:

---

```
1 const vector<char> ascii_vec = create_random_ascii(input_chars);
```

---

Die Variable **input\_chars**, ist vom Typ string und enthält die einzelnen Elemente, die ein Benutzer optional mitgeben kann, um die Menge des Alphabets, aus dem ASCII-Zeichen zufällig gewählt werden, festzulegen. Dies wird in der Funktion, welche diese erstellt überprüft:

---

```
1 vector<char> create_random_ascii(string allowed_ascii_signs="")
```

---

Diese Funktion liefert einen mit chars befüllten Vektor zurück, falls der Benutzer das Alphabet eingeschränkt hat, wird **allowed\_ascii\_signs**, entsprechend gesetzt, was per default ein Leerstring ist. Insgesamt können Zeichen zwischen 1 und 127 mal in den Vektor eingefügt werden, dies hat keinen speziellen Grund, man könnte theoretisch das obere Limit an Zahlen auch bis zum maximalen Vektor-Größe Limit setzen. Jedoch wäre das nicht sinnvoll, da die Ausgabe in der Kommandozeile sehr lange dauern würde und andere Probleme mit dem Speicher auftreten würden, bei Interesse sind dazu weiter unten Links.

Die maximale Vektor Größe könnte man wie folgt feststellen:

---

```
1 std::vector<int> myvector;  
2 std::cout << "max_size: " << myvector.max_size();  
3  
4 //Moegliche Ausgabe:  
5  
6 max_size: 1073741823
```

---

Danach überprüft das Programm ob die Anzahl der Zeichen im Vektor nicht größer ist, als das angegebene Sliding Window. Falls ja wird ein entsprechender

Fehler geworfen und das Programm beendet. Falls die Bedingung jedoch stimmt, dann wird ein Socket erstellt und der Client versucht sich mit dem Endpunkt zu verbinden, auf welchem der Server vorhin gestartet worden ist.

```
asio::error_code ec;
asio::io_context context;
asio::ip::tcp::socket socket(context);
socket.connect(asio::ip::tcp::endpoint(asio::ip::address::from_string("127.0.0.1", ec), 9999));
```

Abb. 8: Client Socketverbindung

Die Wahl des Ports beruht auf einem Leitfaden zur Auswahl von Ports bei Networking-Programmierung, der Port 9999 wurde auf Seiten von TCP und UDP nicht besonders spezifiziert oder besetzt. Nachzulesen unter:

[https://en.wikipedia.org/wiki/List\\_of\\_TCP\\_and\\_UDP\\_port\\_numbers](https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers)

Danach wird geloggt, dass der Client sich mit dem Server verbunden hat und die Datenübertragung beginnt. Es werden zuerst die Größe der Fenster geschickt und die Anzahl aller Datenframes geschickt. Der Client speichert die schon vorhin beschriebene MAX\_SUM aller Daten zwischen und beginnt nun mit der Übertragung der eigentlichen ASCII-Zeichen jeweils in Anzahl der Fenstergröße auf einmal.

Der Client überprüft, ob der Server die richtigen AKN sendet und lässt so das Fenster weiter gleiten. Falls ein falscher AKN kommen, wird ein Fehler geloggt, und das Programm fährt fort. Am Ende empfängt der Client die MAX\_SUM des Server, wenn diese mit der vorhin zwischengespeicherten des Clients übereinstimmt, kommt eine Nachricht, dass die Übertragung ohne Probleme vollzogen werden konnte. Falls nicht, wird dies auch entsprechend geloggt und ausgegeben. Der gesamte Vorgang wird auf Client und Serverseite in zwei separate Logdateien geschrieben. Der Pfad für die des Clients steht in der settings.json Datei im **/build** - Verzeichnis. Die des Servers ist fix im Code angegeben, und zwar immer im aktuellen Verzeichnis, wo sich die ausführbare Datei des Servers beim Starten befindet.

## 3.2 Umsetzung Server

Wenn der Server gestartet wird, wird zuerst ein Endpoint erstellt und ein **asio::acceptor** listened auf neue Verbindungen. Danach wird der Logger erstellt wie in Abb. 9 zu sehen ist.

```
asio::error_code ec;
asio::io_context context;
asio::ip::tcp::socket socket(context);
socket.connect(asio::ip::tcp::endpoint(asio::ip::address::from_string("127.0.0.1", ec), 9999));
```

Abb. 9: Server Listen

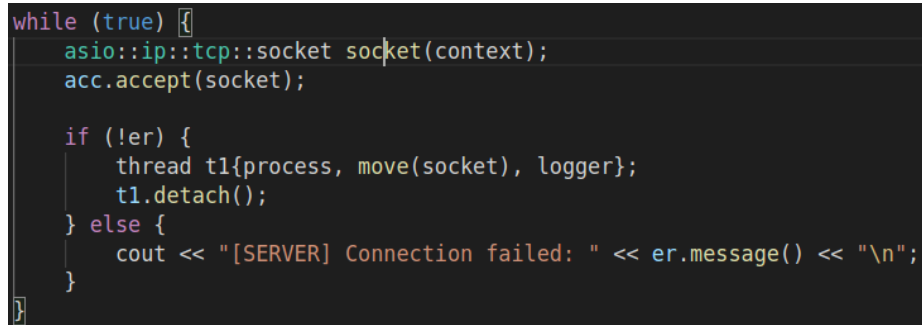
In einer Endlos-Schleife, erzeugt der Server dann mit ankommenden Verbindungen einen Socket und akzeptiert sie. Falls dies ohne Fehler funktioniert startet er für jeden Socket einen Thread, welcher die Funktion **process()** ausführen muss.

---

```
1 void process(asio::ip::tcp::socket socket ,  
2 std::shared_ptr<spdlog::logger> logger)
```

---

Diese Funktion erwartet eine ein Socket-Objekt und ein ein **shared\_pointer** vom Typ **spdlog::logger**. Dieser Ablauf ist in Abb. 10 zu sehen.



```
while (true) {  
    asio::ip::tcp::socket socket(context);  
    acc.accept(socket);  
  
    if (!er) {  
        thread t1{process, move(socket), logger};  
        t1.detach();  
    } else {  
        cout << "[SERVER] Connection failed: " << er.message() << "\n";  
    }  
}
```

Abb. 10: Server Eingehende Verbindung

In der Funktion **process()** wird zuerst die die Sliding Window Fenstergröße empfangen, dann die Anzahl aller zu erwartenden Daten. Später, aber noch vor der Verarbeitung der eigentlichen ASCII Werte, wird noch eine Uhrzeit gespeichert und bei jedem Schleifendurchlauf, wird diese mit der aktuellen Zeit am Server verglichen, der gesamte Prozess darf aus Sicherheitsgründen nicht mehr als sechs Sekunden dauern. Falls diese Zeit überschritten wird, wird beim nächsten Schleifendurchlauf das Socket geschlossen, und es können auf diesem keine Daten mehr gesendet und empfangen werden.

Der Server loggt alle Ereignisse und gibt den Ablauf auch in der Konsole aus. Jeden AKN und die aktuelle Checksumme des Algorithmus. Am Ende wird mit der **max\_sum** noch die größte Summe berechnet und übertragen. Danach beendet der Server den Socket nach einer 90 Millisekunden langen Wartezeit. Dies ist aufgrund von Performance und des speziellen Programmablaufes künstlich erzeugt worden.

## 4 Verwendete Externe Klassen

- [rang-library](#): für die farbige Ausgabe der Tabelle
- [CLI11-library](#): für die Verarbeitung von Eingabe und Options- und Funktionsargumenten
- [spdlog-library](#): für das Logging
- [tabulate-library](#): für die Formatierung der Ausgabe als Tabelle
- [asio-library](#): für das networking in C++
- [chrono-library](#): für Zeitmessungen
- [json-library](#): JSON for modern C++ für die Änderung des Logpfades und Speicherung dessen

## 5 Quellen

- <https://tools.ietf.org/html/rfc1180>
- <https://justtechreview.com/sliding-window-protocol-program-in-c-and-c/>
- <https://tools.ietf.org/html/rfc1180#section-7>
- <https://www.tutorialspoint.com/sliding-window-protocol>
- <https://www.geeksforgeeks.org/window-sliding-technique/>
- [https://www.cplusplus.com/reference/vector/vector/max\\_size/](https://www.cplusplus.com/reference/vector/vector/max_size/)
- <https://stackoverflow.com/questions/32316346/limit-on-vectors-in-c>
- [Folie distsys1.pdf](#) aus dem NVS Unterricht und Folie 26-, 27- und 28-[tcpip\\_programming.pdf](#)

**Alle benutzten Bilder sind frei für akademische Zwecke zu benutzen  
oder das Urheberrecht liegt beim Verfasser (Maurice Putz)**