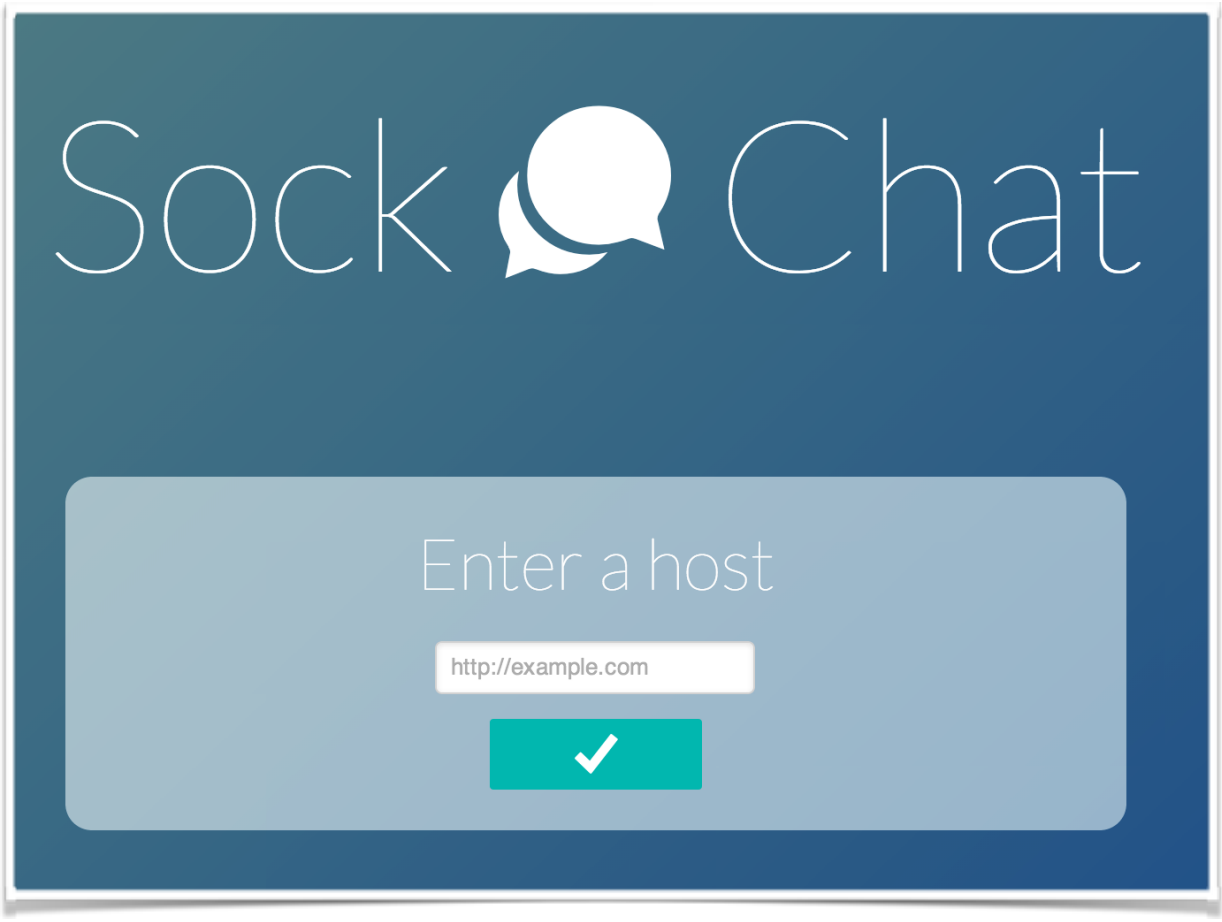



# C Websocket Chat Server

Johann Hofmann - 785841

## Zielstellung

Implementierung eines Chat Servers in der Programmiersprache C und eines Chat Client, der in den üblichen Web-Browsern ausgeführt werden kann. Der Server startet Verbindungen auf einer gewünschten Anzahl von Ports. Der Client soll Nachrichten versenden und empfangen können, sowie erhaltene Nachrichten persistieren.

## Inhalt

# KONZEPT

## Herausforderungen und Lösungsansätze

- Das Server-Programm sollte ohne mehrfach ausgeführt zu werden mehrere Server auf verschiedenen Ports gleichzeitig (und unabhängig voneinander) laufen lassen. Für jede Port-Verbindung wird ein Socketserver als Subprozess gestartet. Der Prozess wartet auf Verbindungen und delegiert diese an Threads.

- Ein einzelner Server sollte eine große Anzahl an gleichzeitigen Client-Verbindungen parallel behandeln können, ohne dass ein Client auf den anderen warten muss. Hierfür wird vom Prozess für jeden Client ein Thread geöffnet, der die Verbindung mit dem Client verwaltet, Nachrichten empfängt und weiterleitet. Bei Verbindungsabbruch muss der Thread die Verbindung zum Client schließen, allokierte Ressourcen freigeben und sich selbst terminieren. Dies ist äußerst wichtig und muss präzise und vollständig erledigt werden, damit der Server im Idealfall ohne Unterbrechung laufen kann, ohne mit der Zeit immer mehr Ressourcen zu belegen.

- Die Verbindung auf Serverseite wird durch Sockets geregelt. Eine generische Socket-Verbindung ist in herkömmlichen Browsern aus Sicherheitsgründen nicht implementiert. Um trotzdem eine Verbindung mit dem Browser herstellen zu können, wird in der Aufgabenstellung die Benutzung eines PHP-Programms als “Middleman” empfohlen. In meiner Implementierung habe ich mich dagegen entschieden, da dies in meiner Empfindung eine unschöne Barriere zwischen Client und Server setzt. Des weiteren ist die Verbindung nun nicht mehr wirklich in “Echtzeit”. Glücklicherweise können Sockets in modernen Browsern durchaus direkt angesprochen werden, unter Verwendung des Websocket-Protokolls.

## Roadmap

- Serverprogramm zum Auslesen der Nutzerparameter und zum Starten der Prozesse.

- Implementierung der Subprozesse zum Öffnen einer Socketverbindung, die auf neue Clients warten und Threads starten und verwalten.

- Implementierung des Thread-Codes zum Verwalten der Client-Verbindung, Empfangen, Verarbeiten und Weiterleiten von Nachrichten.

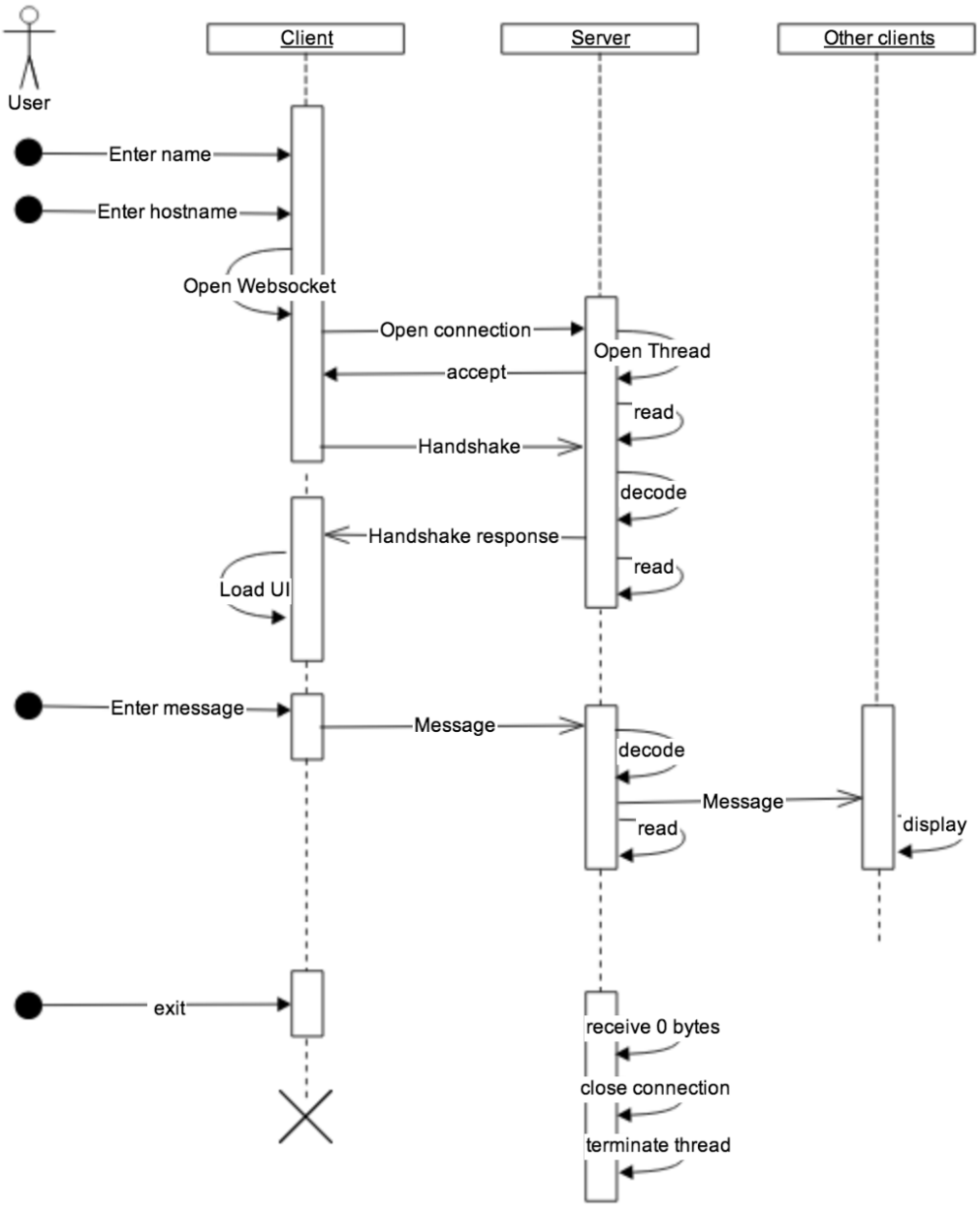
- Vollständige Implementierung des Websocket-Protokolls auf Server-Seite. Einschließlich Handshake, Auslesen der Protokoll-Header und Dekodieren von Nachrichten. (Zum Zeitpunkt meiner Implementierung gab es noch nicht die Möglichkeit, sich der vom Dozenten bereitgestellten Websocket-Bibliothek zu bedienen).

- (Parallel dazu) Implementierung einer Webseite, die sich mittels Websockets mit dem Server verbindet und dem Nutzer ein Chat-Interface bietet.

# ABLAUF

## Client-Server Kommunikation

Darstellung des Kommunikationsablaufs zwischen Nutzer, Client und Server.



# IMPLEMENTIERUNG

Hier zeige ich einige, wie ich meine, interessante Details der Implementierung. Der vollständige Quellcode kann in der ZIP-Datei der Abgabe gefunden werden.

## Client Framework

Ich verwende **React** (http://facebook.github.io/react/) als Frontend-Framework. React ist ein View-Framework, mit dem Klassen als wiederverwendbare View-Templates und gleichzeitig als Controller für alle untergeordneten Views implementiert werden können. Eine Message sieht in meiner Implementierung so aus:

```
148.   var Message = React.createClass({
149.     render: function () {
150.       return (
151.         <div className={"message " + this.props.origin}>
152.           <span className="message-text">{this.props.message}</span>
153.           <span className="message-author">{this.props.author}</span>
154.         </div>
155.       );
166.     }
167.   });
```

connect.jsx

Das vielleicht ungewöhnlichste an React ist das direkt ins Javascript eingebettete Pseudo-HTML, das so natürlich nicht kompiliert. Diese für React entwickelte Templatesprache auf XML-Basis heißt JSX und muss mit einem Precompiler in natives JS übersetzt werden.

Um nun aus allen Nachrichten aus dem Array messages Views zu erstellen, kann man einfach für jedes Element mit der JSX-Syntax einen neuen Message-View erstellen.

```
118.   var messages = this.state.messages.map(function (value) {
119.     return (
120.       <Message key={value.message + Math.random()}
121.                author={value.author}
122.                message={value.message}
123.                origin={value.origin} />
124.     );
125.   });
```

connect.jsx

React mag auf den ersten Blick seltsam erscheinen, doch es verleiht den meisten Programmen eine sehr übersichtliche Struktur. Weiterhin ist React besonders für Realtime-Anwendungen mit viel DOM-Manipulation hervorragend geeignet, weil es eine eigene “Mock DOM” verwaltet und bei Updates nur die Teile der DOM neu zeichnet, die sich tatsächlich verändert haben.

Die JSX-Quelle meines Clients findet sich in *public/connect.jsx* und die kompilierte “pure” JS-Variante in *public/connect.js*. Nur letztere wird in der index.html eingebettet. Bemerkenswert ist, wie wenig sich die beiden Dateien unterscheiden und wie die JSX-Objekte auch in reinem Javascript Sinn machen und gut nutzbar sind.

# IMPLEMENTIERUNG

## Client Connection und Threading

Nach dem Erstellen eines Sockets durch `socket()`, `bind()` und `listen()` wird in einer Endlosschleife auf die Verbindung mit einem Client gewartet.

```
168.    while(1){
169.      connfd = accept(listenfd, (struct sockaddr*)NULL, NULL);
```

socket.c

Nun wird ein neuer Thread erstellt, der sich in der Funktion `talk_to_client` um die Kommunikation mit dem Client kümmert.

```
196.      pthread_t* thread1 = malloc(sizeof(pthread_t));
197.      c->thread = thread1;
198.      if (pthread_create (thread1, NULL, (void *) talk_to_client, (void *) c)) {
199.        fprintf(stderr, "%s\n", "Fatal: Cannot create thread!");
200.        exit(1);
201.      }
202.
203.      sleep(1);
204.    }
```

socket.c

Davor wird der Client noch in einer eigenen `struct` gespeichert, wo die wichtigsten Informationen für den eigenen und andere Threads zur Verfügung stehen.

```
3.    typedef struct client {
4.      int connfd;
5.      char connected;
6.      int index;
7.      pthread_t* thread;
8.    }client;
```

socket.c

Neben dem Connection File Descriptor und dem Thread wird hier auch der Verbindungsstatus in der Variable `connected` gespeichert, wodurch andere Threads dem “eigenen” freundlich mitteilen können, dass sie keine Verbindung zum Client aufbauen konnten.

Das Feld `index` bezieht sich auf die Slotnummer des Clients im globalen Array von Clients. Über dieses Array iterieren die Threads, z.B. beim Senden von Nachrichten durch die Funktion `send_to_all`.

```
17.   void send_to_all(char* message, int size, int connfd){
18.     for (int i = 0; i < MAX_CLIENTS; i++) {
19.       if(clients[i] && clients[i]->connfd != connfd){
20.         printf("Sending to %d\n", i);
21.
22.         if(!~send(clients[i]->connfd, message, size, 0)){
23.           fprintf(stderr, "%s%d\n", "Warning: Error sending message to client ", i);
24.
25.           // politely disconnect the client
26.           clients[i]->connected = 0;
27.         }
28.       }
29.     }
30.   }
```

socket.c

# IMPLEMENTIERUNG

## Websocket Handshake

Um eine Websocket-Verbindung mit einem Browser herstellen zu können, muss ein Protokoll-gerechter Handshake durchgeführt werden. Google Chrome auf meinem System sendet dazu als erste Nachricht:

```
GET ws://localhost:3000/ HTTP/1.1
Pragma: no-cache
Origin: null
Host: localhost:3000
Sec-WebSocket-Key: HLaFMzwAWT5umbDR2aEP5Q==
Upgrade: websocket
Sec-WebSocket-Extensions: x-webkit-deflate-frame
Cache-Control: no-cache
Connection: Upgrade
Sec-WebSocket-Version: 13
```

Und erwartet eine Antwort in folgendem Format:

```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Accept: tbPWmEqTLjEMrcYe2HB9TGM24i4=
```

Abgesehen vom `Sec-WebSocket-Accept` Feld sieht die Antwort immer gleich aus.

```
3.  const char* RESPONSE_TEXT = "HTTP/1.1 101 Switching Protocols\r\nUpgrade: websocket\r
    \nConnection: Upgrade\r\nSec-WebSocket-Accept: ";
```

WS.C

`Sec-WebSocket-Accept` kann durch eine Reihe von Transformationen aus dem `Sec-WebSocket-Key` berechnet werden. Zuerst wird ein festgelegter Magic String angehängt.

```
4.  const char* KEY = "258EAFA5-E914-47DA-95CA-C5AB0DC85B11";
5.
6.  char* ws_handshake(const char* receive_buffer){
7.
8.    const char* index = strstr(receive_buffer, "Sec-WebSocket-Key:") + 19;
9.    const char* stop = strstr(index, "\r");
10.   char key[1024];
11.   strncpy(key, index, stop - index);
12.   strncat(key, KEY, 37);
```

WS.C

Anschließend wird der neue String mit SHA1 gehashed und zuletzt Base64 enkodiert.

```
14.   unsigned char hashed[200];
15.   SHA1(key, strlen(key), hashed);
16.
17.   char* encoded = base64(hashed, strlen(hashed));
18.   char* response = malloc(1024);
19.
20.   // build handshake response
21.   strncat(response, RESPONSE_TEXT, strlen(RESPONSE_TEXT));
22.   strncat(response, encoded, strlen(encoded));
23.   free(encoded);
24.   strcat(response, "\r\n\r\n");
```

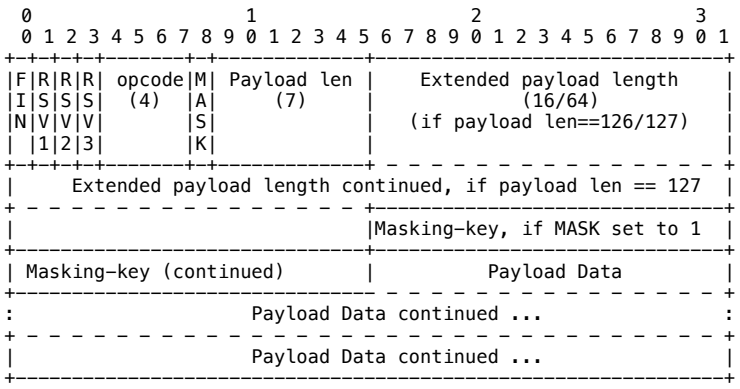
WS.C

Wenn die Antwort vom Browser akzeptiert wurde, können jetzt Nachrichten ausgetauscht werden.

# IMPLEMENTIERUNG

## Websocket-Protokoll Header

Um Websocket Nachrichten zu empfangen und weiterzuleiten, ist es notwendig, den Base Frame auszulesen, der am Anfang jeder Nachricht nach dem Handshake ankommt. Die IETF definiert den Frame folgendermaßen:




Die ersten Bits, die jede Nachricht definieren, können in C mit Bitfields auch als `struct` ausgedrückt werden:

```
7.   typedef union ws_frame{
8.     struct bit_frame{
9.       unsigned int OP_CODE : 4;
10.      unsigned int RSV1 : 1;
11.      unsigned int RSV2 : 1;
12.      unsigned int RSV3 : 1;
13.      unsigned int FIN : 1;
14.      unsigned int PAYLOAD : 7;
15.      unsigned int MASK : 1;
16.    };
17.    unsigned int raw_frame;
18.  }ws_frame;
```

ws.h

Besonders wichtig ist unter anderem die Nachrichtengröße (Payload). Um die Implementierung möglichst simpel zu halten, lasse ich zurzeit keine Nachrichtengrößen über 125 Byte zu. Dies ist bei einem Chat-Server glücklicherweise auch nicht problematisch.

Die Mask-Flag zeigt an, ob die Nachricht mit einem angehängten Masking-Key entschlüsselt werden muss. Browser verschlüsseln ihre Nachrichten immer, ich habe darauf verzichtet. Die Funktion `ws_get_message` aus *ws.c* zeigt meine Dekodierung von Client-Nachrichten.

```
32.    // "parse" the incoming data into a frame
33.    struct bit_frame* in = (struct bit_frame*) &receive_buffer[0];
34.    unsigned char *keys = (unsigned char*) &receive_buffer[2];
35.    unsigned char *encoded = (unsigned char*) &receive_buffer[6];
36.
37.    char* decoded = malloc(in->PAYLOAD + 1);
38.
39.    // xor-decode the message
40.    for (int x = 0; x < in->PAYLOAD; x++) {
41.      decoded[x] = (char)(encoded[x] ^ keys[x % 4]);
42.    }
43.
44.    // end the string
45.    decoded[in->PAYLOAD] = 0;
46.    return decoded;
```

ws.c

Wichtig hier ist vor allem das abwechselnde XOR jedes Bytes der Nachricht mit einem der 4 Masking-Bytes.

# IMPLEMENTIERUNG

## Verbindungsabbruch

Sollte eine Verbindung verloren gehen oder geschlossen werden, ist es auf dem Server besonders wichtig alle allokierten Ressourcen freizugeben sowie den Thread zu beenden. Das Ziel ist es, den Server potentiell unendlich lang laufen lassen zu können, ohne dass mit der Zeit mehr Ressourcen benötigt werden oder er irgendwann einmal abstürzt.

Der Thread eines Client bemerkt die geschlossene Verbindung meist, wenn die blockierende Funktion `recv` statt der erhaltenen Anzahl Bytes 0 zurück gibt.

```
76.     while (c->connected) {
77.       int read_bytes = recv(c->connfd,receive_buffer,bufsize,0);
78.       if(read_bytes <= 0){
79.         printf("Slot %d seems to have disconnected.\n", c->index);
80.         break;
81.       }
```

socket.c

In diesem Fall wird mittels des `break` Statements die Endlosschleife unterbrochen, der Server versucht nun nicht mehr, Nachrichten vom Client zu empfangen.

Nach dieser Schleife folgt nun eine Sequenz von Clean-Anweisungen, die (hoffentlich) alle Ressourcen befreien und den Thread sowie die Socketverbindung zu diesem Client schließen. Außerdem wird der Client aus dem globalen Array `clients` entfernt, damit er Platz für neue Clients macht.

```
123.    printf("Terminating thread %d\n", c->index);
124.    shutdown(c->connfd, 2);
125.    clients[c->index] = 0;
126.    free(c->thread);
127.    free(c);
128.    free(receive_buffer);
129.    pthread_exit(0);
```

socket.c

# BENUTZUNG

## Server

Den Quellcode mit Make kompilieren.

```
$    make server
```

Den Server starten. Erster Parameter ist der erste zu öffnende Port und der zweite Parameter ist die Anzahl der weiteren Ports, die nach dem ersten Port geöffnet werden sollen.

```
$    ./server 3000 5
```

## Client

Um den Client zu starten, muss die Datei *index.html* im Verzeichnis *public/* mit einem Browser geöffnet werden. Es wird ein Browser benötigt, der Websockets unterstützt.

### BROWSER MIT WEBSOCKET-SUPPORT

| **Browser** | Chrome | Firefox | Safari | Opera | Internet Explorer |
|---|---|---|---|---|---|
| **ab Version** | 14.0 | 11.0 | 6.0 | 12.1 | 10.0 |