Jonathan Klinger

5BHIF

Einfacher HTTP 1.1 Client

A short introduction to HTTP 1.1, it's requests
and a simple Client written in C++

Abteilung Informatik
HTBLuVA Wiener Neustadt

Jänner 2021

Contents

1 HTTP

HTTP steht für Hyper Text Transfer Protocol. Es handelt sich um ein Client/Server Protocol das ursprünglich zum Abrufen von statischen Informationen (z.B. Browser ruft eine Website ab) gedacht war. Das Protokoll ist in der Anwendungsschicht (Application Layer) des OSI-Modells angesiedelt. Außerdem wird auf darunterliegenden Schichten mit TCP gearbeitet, was es zuverlässig macht.
HTTP wurde von der Internet Engineering Task Force (IETF) und dem World Wide Web Consortium standardisiert.

1.1 Eigenschaften

1.1.1 (Relativ) Einfach

Das Protokoll arbeitet mit einfachen Anfragen und Antworten (Requests und Responses). D.h. der Client (z.B. wieder ein Webbrowser) schickt eine Anfrage, der Informationen über die Ressource die er zugeschickt bekommen möchte an den Server, der mit einer standardisierten Antwort, die im besten Fall die Ressource enthält, antwortet. Auf die verschiedenen Arten von Request und die Arten von Response Status Codes werde ich später eingehen.

1.1.2 Zustandslos

HTTP ist Zustandslos, was bedeutet, dass alle Anfragen unabhängig voneinander sind. Informationen aus vorherigen Anfragen gehen verloren. Mittels Cookies in den Header-Informationen (siehe unten) können allerdings Status-Informationen zugeordnet werden.

1.1.3 Nicht auf Hypertext beschränkt

Auch wenn es ursprünglich dafür gedacht war, können immer mehr beliebige Daten versendet werden.

1.1.4 Verhandlung der Datenräpresentation

Die Content Negotiation wird verwendet um eine Abstimmung der Inhalte einer Ressource auf die Möglichkeiten des Clients ermöglicht.

1.2 HTTP 1.1

Diese Version des Protokolls wurde 1999 im RFC2616 publiziert. Eine der größten Erweiterungen war, dass der Client den Wunsch äußern kann (per Header Eintrag, siehe unten) den Verbindungsabbau zu überspringen, um dieselbe später wieder nutzen zu können. Seitdem kann auch eine TCP Verbindung für mehrere Anfragen verwendet werden können was die Ladezeiten extrem verkürzt da diese gerade beim Verbindungsaufbau langsam sind.
Außerdem wurde der MIME-Typ multipart/replace hinzugefügt.

1.3 Funktionsweise

Wie oben schon angeschnitten, Funktioniert HTTP mittel Anfrage und Antwort. Egal ob man im Browser eine URL eingibt oder ob man selbst mittels verschiedenen Technologien Anfragen versendet läuft der Ablauf zwischen Client und Server gleich ab.

1.3.1 Header

Die notwendigen Informationen für eine erfolgreiche Kommunikation werden im Header hinterlegt.
Anfrage Header:

```
GET /maps HTTP/1.1
HOST: www.google.com
Connection: close
...
```

Notwendig sind nur die 1. Beiden Zeilen. "GET" spezifiziert den Anfrage-Typ (siehe unten) danach folgt der Pfad am Server zur Ressource und die verwendete HTTP-Version. Die nächste Zeile spezifiziert den Host (und Portnummer) mittels URL.
Antwort Header:

```
HTTP/1.1 200 OK
Content-Type: text/html
Date: Fri, 16 Apr 2021 00:00:00 GMT
Server: Apache
Set-Cookie: key=value
...
```

Der Header der Antwort ist meistens um einiges Umfangreicher als der der Anfrage. In der Ersten Zeile ist wieder die verwendete HTTP Version sowie der Status Code und die Status Message. Es folgen ein Auszug der wichtigsten Antwort-Header-Felder.
Der Content-Type spezifizert wie die Daten die nach dem Header folgen interpretiert werden sollten. Date enthält das Datum und die genaue Zeit wann die Antwort gesendet wurde. Das Feld Server zeigt an welche Software der Server verwendet. Und mit Set-Cookie werden die bereits erwähnten Cookies gesetzt.

1.4 HTTP Authentifizierung

Grundsätzlich gibt es mehrere Arten der Authentifizierung bei diesem Protokoll. Hier werde ich nur auf die Basic Authentication eingehen. Ist eine Ressource geschützt fordert der Server eine Benutzername/Passwort-Kombination an. Wird diese nicht geliefter sendet der Server einen Fehlercode (401, siehe unten).
Die Kombination wird im Header folgendermaßen angegeben:

```
Authentication: Basic YmVudXR6ZXI6cGFzc3dvcnQ=
```

Der hier zu sehende Text ist *nicht* verschlüsselt. Es handelt sich lediglich um die Kombination in Base64 codiert. Die Kombination muss als "benutzername:passwort" codiert werden.
Da es sich wie erwähnt nur um Base64 handelt ist diese Methode sehr unsicher!

1.5 HTTP Status Codes

Jede Anfrage wird vom Server mit einem passenden Statuscode und einer Message beantwortet. Es kann z.B. über den Erfolg oder etwaige Fehler informiert werden.
Hier ein Überblick über die möglichen Antworten:

- 1xx - Info z.B.: 100: Continue
- 2xx - Erfolg z.B.: 200: OK
- 3xx - Umleitung z.B.: 300: Multiple Choice
- 4xx - Client Fehler z.B.: 401: Unauthorized
- 5xx - Server Fehler z.B.: 500: Internal Server Error

Mithilfe der Fehlercodes kann der Client auf verschiedene Antworten passend reagieren und gegebenfalls eine Fehlersuche erleichtern.

1.6 HTTP Anfragen

1.6.1 GET

Ist die am meisten verwendete Methode. Es wird eine Ressource vom Server angefordert. Es besteht die Möglichkeit Argumtente im URI zu übermitteln. Laut Standard sollte mit GET nur Ressourcen abgerufen werden, sonst nichts. Der Request enthält keinen Body und somit auch keine Header Elemente wie Content-size.

1.6.2 DELETE

Löscht die angegebene Ressource auf dem Server. Der Request enthält ebenfalls keinen Body und somit auch keine Header Elemente wie Content-size.

1.6.3 POST

Schickt unbegrenzte Mengen an Daten zur weiteren Verarbeitung an den Server. Diese Daten befinden sich im Body der Nachricht. Dieser Inhalt kann einfacher Text, Key-Value-Paare oder ähnliches sein. Die Daten werden üblicherweise nicht gecached.

1.6.4 PUT

Liefert die gleiche Funktionalität wie POST mit dem Unterschied, dass wenn bereits eine Ressource bereits existiert ersetzt und sonst neu erstellt wird. Der Vorgang ist somit idempotent.

2 Das Programm

2.1 Aufgabenstellung

Die Aufgabenstellung:

"Funktionsfähiger HTTP 1.1-Client zum Herunterladen/Speichern von 3 beliebigen Dateien (GET, PUT, POST, DELETE, Basisfunktionalität; Steuerung über Kommandozeile; inkl. HTTP Basic-Authentication, nicht auf Header vergessen und inkl. Cookies, http-parser kann ver- wendet werden) "

2.2 Meine Lösung

Ich habe meine Arbeit in C++ programmiert. Es gibt einige Lösungsansätze. Für Übungszwecke habe ich mich auf die Verwendung von asio und spdlog konzentriert.

2.2.1 Asio

Asio ist eine Plattform-übergreifende Bibliothek für Netzwerkprogrammierung. Die Bibliothek stellt eine Reihe an synchronen und asynchronen Funktionen zur Verfügung.
Synchrone Kommunikation basiert auf blockierenden Operationen, während Asynchrone auf nicht blockierenden Operationen und Callbacks basiert.

Asio bietet die Möglichkeit mit Fehlercodes oder auch Exception die Fehler handzuhaben.

2.2.2 Spdlog

Bei spdlog handelt es sich um eine header only Bibliothek die das Logging erleichtert. Die Vorteile liegen darin das es sehr schnell arbeitet und eine schöne, farbige Formatierung ermöglicht, die auch individuell angepasst werden kann.

2.2.3 CLI11

Hier handelt es sich um einen Kommandozeilen-parser. Es ist wie auch spdlog eine schnelle Header-only-Bibliothek. Ein großer Vorteil ist das diese Bibliothek platformübergreifend funktioniert. Es lassen sich auch schöne Fehlermeldungen bilden.

2.2.4 Ablauf

Das Programm kann 3 Anfragen pro Aufruf durchführen. Diese werden per CLI11 aufgenommen. Die Parameter jeder Anfrage werden jeweils in einem vector mit strings gespeichert.

```
//Interface
CLI::App app{"Einfacher HTTP1.1 Client"};

//Vektoren f+r bis zu 3 Anfragen
vector<string> req1;
vector<string> req2;
vector<string> req3;

//CLI11 - Optionen
app.add_option("-1", req1, "First Request");
app.add_option("-2", req2, "Second Request");
app.add_option("-3", req3, "Third Request");

CLI11_PARSE(app, argc, argv);
```

In diesem Ausschnitt sieht man die Initialisierung der Drei Vektoren in denen Parameter wie URL, Port, usw. gespeichert werden. Die Reihenfolge muss stimmen (siehe unten). mittels "app.add_option()" wird die Benutzereingabe den jeweiligen Vektoren zugewiesen. Es wird solange der input in einem Vektor gespeichert bis ein neues Schlagwort erkannt wird.

Diese Benutzereingabe wird jetzt so gut als Möglich auf Fehler überprüft. In der main-Funktion wird sie hinsichtlich des Anfragetyps getestet.

```
if (req1[0] == "GET" || req1[0] == "DELETE") {
    send_GET_DELETE(req1, "1");
  } else if (req1[0] == "POST" || req1[0] == "PUT") {
    send_POST_PUT(req1, "1");
  } else {
    spdlog::error("1. Request: Typ nicht erkannt");
  }

  if (req2.size() > 0) {
    if (req2[0] == "GET" || req2[0] == "DELETE") {
      send_GET_DELETE(req2, "2");
    } else if (req2[0] == "POST" || req2[0] == "PUT") {
      send_POST_PUT(req2, "2");
    } else {
      spdlog::error("2. Request: Typ nicht erkannt");
    }
  }

```

```
  if (req3.size() > 0) {
    if (req3[0] == "GET" || req3[0] == "DELETE") {
      send_GET_DELETE(req1, "3");
    } else if (req3[0] == "POST" || req3[0] == "PUT") {
      send_POST_PUT(req3, "3");
    } else {
      spdlog::error("3. Request: Typ nicht erkannt");
    }
  }
```

Es wird davon ausgegangen das zumindest eine Anfrage durchgeführt wird. Die anderen Vektoren werden geprüft ob sie gesetzt sind, da sonst der Fehler "segmentation fault (core dumped)" aufgerufen wird da versucht wird mit nicht gestztem Speicher zu arbeiten. Wird kein passender Anfragetyp erkannt, wird mit spdlog eine Fehlermedlung angezeigt. Auch wenn andere Anfragen fehlerhaft sind, werden trotzdem die anderen durchgeführt.

Die Restliche Eingabe wird in den Funktionen "send_GET_DELETE()" und "send_POS_PUT()" analysiert und verarbeitet.
Die variable

```
unsigned int file_place = 4;
```

zeigt die Position an der der Dateiname steht in dem die Antwort auf eine GET Anfrage gespeichert werden soll. Bei POST erübrigt sich das, da nicht in Dateien gespeichert wird.

Dieser if/else Teil analysiert anhand der länge der Eingabe welche weiteren Header Felder gestzt und die variable file_place angepasst werden muss.

```
if (input.size() == 8) {
    spdlog::info(n + ". Request: Cookies erkannt");
    spdlog::info(n + ". Request: HTTP Basic Authorization erkannt")
    ;

    file_place = 7;

    string auth = input[5] + ":" + input[6];
    string base_auth = base64(auth);

    req_string = req_string +
    "Authorization: Basic " + base_auth + "\r\n"
    "Cookie: " + input[4] + "\r\n\r\n";
} else if (input.size() == 7) {
    spdlog::info(n + ". Request: HTTP Basic Authorization erkannt")
    ;

    file_place = 6;

    string auth = input[4] + ":" + input[5];
    string base_auth = base64(auth);

    req_string = req_string + "Authorization: Basic " + base_auth +
     "\r\n\r\n";
} else if (input.size() == 6) {
```

```
    spdlog::info(n + ". Request: Cookie erkannt");
    file_place = 5;

    req_string = req_string + "Cookie: " + input[4] + "\r\n\r\n";
} else {
    req_string = req_string + "\r\n";
}
```

Die Hier gezeigte Funktion "base64()" kümmert sich um die oben beschriebene codierung von benutzer:passwort. Da es keine C++ Funktion dafür gibt und ich in keiner der zur Verfügung stehenden Bibliotheken eine passende funktion gefunden habe, habe ich auf die Linux eigene bash Funktion mithilfe eines System-calls zugegriffen. Das funktioniert folgendermaßen:

```
string base64(string str) {
  string command = "printf " + str + " | base64 > base64.txt";
  system(command.c_str());

  ifstream ifs("base64.txt");
  string ret{ std::istreambuf_iterator<char>(ifs), std::
    istreambuf_iterator<char>() };
  ifs.close();
  return ret;
}
```

Mithilfe des Systemcalls wird der codierte text in eine Datei geschrieben die ich dann auslesen kann. Leider Funktioniert das Programm damit nur auf Linux-Betriebssystemen, was aber in diesem Fall kein Problem ist.

Die Anfrage wird mithilfe von Asio folgendermaßen gesendet.

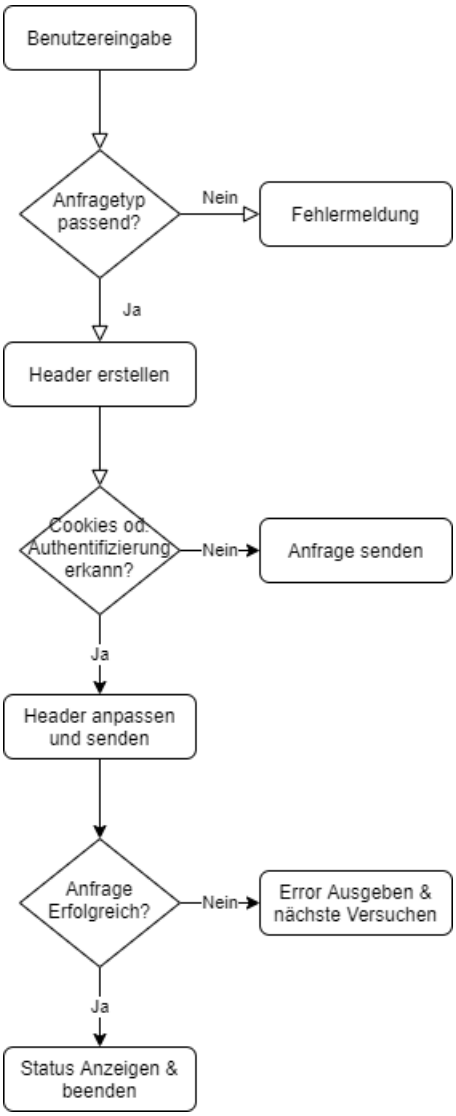
```
char reply[1000];
error_code ec;

size_t reply_length = asio::read(sock, asio::buffer(reply), ec);

//Char[] to String
string res = "";
for (unsigned int i=0; i<reply_length; i++) {
    res = res + reply[i];
}
//Only get body/status from response
string body = res.substr(res.find("\r\n\r\n") + 4);
string status = res.substr(0, res.find("\r\n"));
```

In dem character Array reply wird die Antwort des Servers gespeichert. Diese wird dann wieder in einen string umgewandelt damit man die funktion "find()" verwenden kann.

Mit den Variablen "status" und "body" wird dann eine Ausgabe mittels spdlog erstellt. Außerdem wird der Body (zumindest bei GET) in eine Datei geschrieben. Die Funktion "send_POST_PUT()" funktioniert im grunde genau wie das gezeigte Material mit einem unterschied in den Head-Feldern und das die Antwort nicht in einer Datei gespeichert wird.
Hier nochmal eine vereinfachte Zusammenfassung:



2.2.5 Bedienung

Es ist wichtig das die Parameter in der richtigen Reihenfolge angegeben werden.
Hier eine allgemeine definition:

```
./klinger_project_2 -1|-2|-3 GET|POST|PUT|DELETE url path port [cookies]
  [content-type content] [file] [username passwort]
```

Beachte: Bei POST und PUT wird kein Dateiname erwartet!
Eine gute Möglichkeit aufrufe zu testen ist für mich ptsv2.com. Diese Website dient als "Mülleimer" für GET und POST anfragen. Unter /t/jonny/post

kann man die Anfragen senden und unter ptsv2.com/t/jonny einsehen. Diese Möglichkeit sollte nach stand 16.04.2021 noch einige Tage zur verfügung stehen.

3 Sources

- TCP/IP Programmierung von Prof. Kolousek
- HTTP-Skripten von Prof. Kolousek
- https://www.w3schools.com/tags/ref_httpmethods.asps
- https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Evolution_of_HTTP
- https://www.w3.org/Protocols/rfc2616/rfc2616.html
- https://tools.ietf.org/html/rfc2616