Technische Universität Berlin

TKN

Praktikum

Rechnernetze

Praktikum 1:

How to code your webserver

Für die erste Abgabe implementieren Sie einen Webserver. In den folgenden Aufgaben erstellen und erweitern sie sukzessive ihre Implementierung und überprüfen im Anschluss deren Funktion mit den vorgegebenen Tests. Ihre finale Abgabe wird automatisiert anhand der Tests bewertet.

Hinweise

Die praktischen Aufgaben sind in Kleingruppen von bis zu vier Personen zu lösen. Reichen Sie deshalb Ihren Quelltext bzw. Lösungen dieser Aufgaben bis zum **25.11. 23:59 Uhr per ISIS** ein. Aufgaben werden automatisch sowohl auf Plagiate, als auch auf Korrektheit getestet, halten Sie sich deshalb genau an das vorgegebene Abgabeformat.

Auf der ISIS-Seite zur Veranstaltung finden Sie zusätzliche Literatur und Hilfen, insbesondere den *Beej's Guide to Network Programming Using Internet Sockets*[1], für die Bearbeitung der Aufgaben!

Projektsetup

Die Praxisaufgaben des Praktikums sind in der Programmiersprache C zu lösen. C ist in vielen Bereichen noch immer das Standardwerkzeug, speziell in der systemnahen Netzwerkprogrammierung mit der wir uns in diesem Praktikum beschäftigen. Die Aufgaben können Sie mithilfe eines Tools Ihrer Wahl lösen, es gibt diverse Editoren, IDEs, Debugger, die bei der Entwicklung hilfreich sein können. Als IDE können wir `CLion` empfehlen, welches zwar proprietär, aber für Studierende kostenlos verfügbar ist. Grundlegend ist

[1] https://beej.us/guide/bgnet

allerdings ein einfacher Texteditor wie `kate`, ein Compiler (`gcc`), und ein Debugger (`gdb`) völlig ausreichend.

a) Richten Sie Ihre Entwicklungsumgebung auf Ihrem PC ein.

   Installieren Sie dazu Compiler, Debugger, und Editor. Die konkreten Schritte dazu hängen von Ihrem System ab. Eine minimale Umgebung können Sie auf einem `apt`-basierten Linux (Debian, Ubuntu, ...), oder unter Verwendung des *Windows Subsystem for Linux* (WSL) via `apt install` installieren.

   Aufgrund der vielfältigen Landschaft von Betriebssystemen können wir Sie bei diesem Schritt leider nur eingeschränkt unterstützen. Die meisten Probleme lassen sich zwar mit vergleichsweise schnell mithilfe einer Suchmaschine lösen, trotzdem stellen wir Ihnen eine virtuelle Maschine mit einer funktionierenden Entwicklungsumgebung auf ISIS bereit.

b) Wir verwenden CMake[2] als Build-System für die Praktikumsabgaben, sowie für Tests. Um Ihre Entwicklungsumgebung zu testen, erstellen Sie ein CMake Projekt und compilieren Sie Damit ein simples C Programm.

   Ihr Projekt sollte folgende Struktur aufweisen:

```
Praxis1
├── CMakeLists.txt
└── webserver.c
```

   Sie können Ihren Code in der Konsole mit den folgenden Befehlen compilieren:

```
$ cmake -B build -DCMAKE_BUILD_TYPE=Debug
$ make -C build
$ ./build/webserver
```

   Durch die CMake Variable `CMAKE_BUILD_TYPE=Debug` wird eine ausführbare Datei erstellt die zur Fehlersuche mit einem Debugger geeignet ist. Sie können das Programm in einem Debugger wie folgt ausführen:

[2] https://cmake.org/

```
gdb ./build/webserver
```

Das von uns bereitgestellte Projektskelett enthält zusätzlich Tests, die ebenfalls in Ihrem Projektordner vorhanden sind. Diese können Sie mit dem target `test` ausführen:

```
make -Cbuild test
```

Für die Fehlersuche kann es hilfreich sein den Server selbst im Debugger auszuführen und nicht von den Tests starten zulassen. Dafür könnt ihr die Tests manuell ausführen und das `--no-spawn` Flag verwenden:

```
gdb --args ./build/webserver 4711  # Das --args flag instruiert gdb
                                   # Parameter an die ausführbare
                                   # Datei zu übergeben anstatt sie
                                   # selbst zu interpretieren
```

Und in einer anderen Shell:

```
./test/test.py --port 4711 --no-spawn
```

Hinweis: Wenn Sie eine integrierte Entwicklungsumgebung verwenden wird gegebenenfalls ein anderes Build-Verzeichnis generiert.

Listen socket

Zunächst soll Ihr Programm einen auf einen TCP-Socket horchen. `HTTP` verwendet standardmäßig Port 80. Da Ports kleiner als 1024 als privilegierte Ports gelten und teils nicht beliebig verwendet werden können, werden wie den zu verwendenden Port als Parameter beim Aufruf des Programms übergeben. Beispielsweise soll beim Aufruf `webserver 1234` Port 1234 verwendet werden.

Anfragen beantworten

Der im vorhergehenden Schritt erstellte Socket kann nun von Clients angesprochen werden. Erweitern Sie Ihr Programm so, dass es bei jedem empfangenen Paket mit dem String

“Reply” antwortet.

Pakete erkennen

TCP-Sockets werden auch als Stream Sockets bezeichnet, da von Ihnen Bytestreams gelesen werden können. HTTP definiert Anfragen und Antworten hingegen als wohldefinierte Pakete. Dabei besteht eine Nachricht aus:

- Einer Startzeile, die HTTP Methode und Version angibt,
- einer (möglicherweise leeren) Liste von Headern (`Header: Wert`),
- einer Leerzeile, und
- dem Payload.

Hierbei sind einzelne Zeilen die mit einem CRLF-Token[3] terminiert. Nachrichten die einen Payload enthalten kommunizieren dessen Länge über den `Content-Length` Header.

Eine (minimale) Anfrage auf den Wurzelpfad (`/`) des Hosts `example.com` sieht entsprechend wie folgt aus:

```
GET / HTTP/1.1\r\n
Host: example.com\r\n
\r\n
```

Sie können einen Request mit `netcat` wie folgt testen: `echo -en 'GET / HTTP/1.1\r\nHost: example.com\r\n\r\n' | nc example.com 80`.

Ihr Webserver muss daher in der Lage sein diese Pakete aus dem Bytestream herauszuerkennen. Dabei können Sie nicht davon ausgehen, dass beim Lesen des Puffers exakt ein Paket enthalten ist. Im Allgemeinen kann das Lesen vom Socket ein beliebiges Präfix der gesandten Daten zurückgeben. Beispielsweise sind alle folgenden Zeilen mögliche Zustände Ihres Puffers, wenn `Request1\r\n\r\n`, `Request2\r\n\r\n`, und `Request3\r\n\r\n` gesendet wurden, und sollten korrekt von Ihrem Server behandelt werden:

```
Reque
```

[3] https://en.wikipedia.org/w/index.php?title=CRLF

```
Request1\r\n\r\nR
Request1\r\n\r\nRequest2\r\n\r
Request1\r\n\r\nRequest2\r\n\r\nRequest3
Request1\r\n\r\nRequest2\r\n\r\nRequest3\r\n\r\n
```

Erweitern Sie Ihr Programm so, dass es auf jedes empfangene HTTP Paket mit dem String “Reply\r\n” antwortet. Gehen Sie im Moment davon aus, dass die oben genannte Beschreibung von HTTP Paketen (Folge von nicht-leeren CRLF-separierten Zeilen die mit einer leeren Zeile enden) vollständig ist.

Syntaktisch korrekte Antworten

HTTP sieht eine Reihe von Status Codes[4] vor, die dem Client das Ergebnis des Requests kommunizieren. Da wir zurzeit noch nicht zwischen (in-)validen Anfragen unterscheiden können, erweitern Sie Ihr Programm zunächst so, dass es auf jegliche Anfragen mit dem `Bad Request` (400) Statuscode antwortet. *Hinweis*: Sie können Ihren Server mit Tools wie `curl` oder `netcat` testen und die versandten Pakete mit `wireshark` beobachten.

Semantisch korrekte Antworten

Parsen Sie nun empfangene Anfragen als HTTP Pakete. Ihr Webserver soll sich nun mit den folgenden Statuscodes antworten:

- 400: inkorrekte Anfragen
- 204: GET-Anfragen
- 501: alle anderen Anfragen

Anfragen sind inkorrekt, wenn

- keine Startzeile erkannt wird, also die erste Zeile nicht dem Muster `$Method $URI $HTTPVersion\r\n` entspricht oder

[4] https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

- eine Methode die einen Payload verwendet keinen `Content-Length` Header mitsendet.

Statischer Inhalt

Zum jetzigen Zeitpunkt haben Sie einen funktionierenden Webserver implementiert, wenn auch einen wenig nützlichen: Alle Ressourcen[5] existieren, enthalten aber keinen Inhalt. Erweitern Sie Ihre Implementierung also um Antworten, die konkreten Inhalt (einen Body) enthalten. Gehen Sie hierfür davon aus, dass die folgenden Pfade existieren:

```
static/
├── foo: "Foo"
├── bar: "Bar"
└── baz: "Baz"
```

Entsprechend sollte eine Anfrage auf den Pfad `static/bar` mit `Ok` (200, Inhalt: `Bar`), und eine Anfrage nach `static/other` mit `Not found` (404).

Dynamischer Inhalt

Der Inhalt der in der vorigen Aufgabe ausgeliefert wird ist strikt statisch: Der vom Webserver ausgelieferte Inhalt ändert sich nicht. Das HTTP-Protokoll sieht allerdings auch Methoden vor, welche die referenzierte Ressource verändert, also beispielsweise deren Inhalt verändert, oder diese löscht.

In dieser letzten Aufgabe wollen wir solche Operationen rudimentär unterstützen. Dafür wollen wir zwei weitere HTTP Methoden verwenden: `PUT` und `DELETE`.

- `PUT` Anfragen erstellen Ressourcen: der mit versandte Inhalt wird unter dem angegebenen Pfad verfügbar gemacht.
- `DELETE` Anfragen löschen Ressourcen: die Ressource unter dem angegebenen Pfad wird gelöscht.

[5] https://de.wikipedia.org/wiki/Uniform_Resource_Identifier%5D

Passen Sie Ihre Implementierung an, sodass sie die beiden neuen Methoden unterstützt. Anfragen sollten dabei nur für Pfade unter `dynamic/` erlaubt sein, bei anderen Pfaden soll eine `Forbidden` (403) Antwort gesendet werden.

Abgabe

Die vorgegebene `CMakeLists.txt` ist so konfiguriert, dass Sie damit ein Archiv des Projekts erstellen können:

```
make -Cbuild package_source
```

Das generierte Archiv sollte nun im `build`-Ordner verfügbar sein: `RN-Praxis1-0.1.1-Source.tar.gz`
Laden Sie Ihre Abgabe bis zum **25.11., 23:59 CET** auf ISIS hoch.

Viel Erfolg :)

Freiwillige Zusatzaufgaben

Ihre Implementierung ist nun ein funktionierender Webserver! Für einen vollen Funktionsumfang fehlen jedoch noch einige Funktionen. Wenn Sie Ihre Implementierung erweitern möchten, bieten sich zum Beispiel folgende Ziele an:

- Inhalte die Dateien aus dem Dateisystem entsprechen, statt dem Speicher des Servers
- Unterstützung für weitere HTTP Verben: `OPTIONS`, `PATCH`, `POST`
- Verschieden Encodings: HTTP erlaubt Clients zu signalisieren in welchem Format sie Inhalt erwarten. HTTP-basierte APIs verwenden hier häufig `text/json` statt `text/html`.