Praxis 1
Webserver

Fachgruppe Telekommunikationsnetze (TKN)

16. November 2023

Formalitäten

Diese Aufgabenstellung ist Teil der Portfolioprüfung. Beachten Sie für Ihre Abgaben unbedingt die entsprechenden Modalitäten (siehe Anhang A).

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

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

1. 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. Im Zweifel können Sie die Aufgaben auf den EECS-Systemen bearbeiten.

2. Wir verwenden CMake als Build-System für die Abgaben. Um Ihre Entwicklungsumgebung zu testen, erstellen Sie ein CMake Projekt, oder verwenden Sie das vorgegeben Projektskelett. Ergänzen Sie das Projekt um ein minimales C Programm namens `webserver`.

   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:

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

   Das von uns bereitgestellte Projektskelett enthält zusätzlich Tests, die ebenfalls in Ihrem Projektordner vorhanden sind.

2. Tests

Die im folgenden beschriebenen Tests dienen sowohl als Leitfaden zur Implementierung, als auch zur Bewertung. Beachten Sie hierzu insbesondere auch die Hinweise in Anhang B.

Jeder einzelne Test kann als Teilaufgabe verstanden werden. In Summe führen die Tests zu einer vollständigen Implementierung des oben beschriebenen Webservers. Teilweise werden spätere Tests den vorigen widersprechen, da sich die Aufgabenstellung im Verlauf weiter konkretisiert. Die Tests sind daher so geschrieben, dass diese auch die weitere Entwicklung Ihrer Lösung als korrekt akzeptieren. Für das Debugging Ihres Programms empfehlen wir zusätzlich zu einem Debugger (z.B. `gdb`) auch `wireshark` zu verwenden.

2.1. 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 wir den zu verwendenden Port als Parameter beim Aufruf des Programms übergeben. Darüber hinaus soll die Adresse auf die der Server den socket bindet übergeben werden. Beispielsweise soll beim Aufruf `webserver 0.0.0.0 1234` Port 1234 auf allen Interfaces verwendet werden.

> **i** Programmen, die Dienste im Netzwerk bereitstellen (Server) wird beim Start meist eine IP übergeben. Diese beschreibt das Netzwerk aus dem Anfragen angenommen werden sollen. `0.0.0.0` bezeichnet beispielsweise das gesamte Internet, `127.0.0.1` erlaubt nur Anfragen von `localhost`.

Verwenden Sie hierfür die socket API: `socket`, `bind` und `listen`. Um die übergebene Adresse zu parsen bietet es sich an `getaddrinfo` zu verwenden (siehe Anhang C).

2.2. 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. Dafür müssen Sie eine Verbindung annehmen (`accept`, Daten empfangen (`recv`) und eine Antwort senden (`send`).

2.3. 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 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:

```
1 GET / HTTP/1.1\r\n
2 Host: example.com\r\n
3 \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 heraus zuerkennen. 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:

```
1 Reque
2 Request1\r\n\r\nR
3 Request1\r\n\r\nRequest2\r\n\r
4 Request1\r\n\r\nRequest2\r\n\r\nRequest3
5 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\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.

> ! Sie können Ihren Server mit Tools wie `curl` oder `netcat` testen und die versandten Pakete mit `wireshark` beobachten.

2.4. Syntaktisch korrekte Antworten

HTTP sieht eine Reihe von Status Codes 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 `400 Bad Request` Statuscode antwortet.

> ! HTTP ist ein komplexes Protokoll. Dieses vollständig zu implementieren sprengt den Rahmen dieser Aufgabe bei weitem. Sie können daher eine Reihe sinnvoller Annahmen treffen (siehe Anhang D) die die Implementierung vereinfachen.

2.5. Semantisch korrekte Antworten

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

- 400: inkorrekte Anfragen
- 404: 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
- eine Methode die einen Payload verwendet keinen `Content-Length` Header mitsendet.

2.6. Statischer Inhalt

Zum jetzigen Zeitpunkt haben Sie einen funktionierenden Webserver implementiert, wenn auch einen wenig nützlichen: Es existieren keine Resourcen[1]. 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"
```

[1] `https://de.wikipedia.org/wiki/Uniform_Resource_Identifier`

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

Da der implementierte Server dem HTTP Standard folgt, können Sie mit diesem mit Standardtools wie `curl`, oder Ihrem Webbrowser interagieren. Entsprechend sollten Sie via `curl localhost:1234/static/foo` die statischen Resourcen abfragen können, nachdem Sie den Server auf Port 1234 gestartet haben. Mit dem `--include`/`-i`-Flag gibt `curl` Details zur empfangenen Antwort aus. Alternativ wird auch Ihr Webbrowser den Inhalt anzeigen, wenn Sie zum entsprechenden URI (`http://localhost:1234/static/foo`) navigieren.

2.7. 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 Resource 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 Resourcen: der mit versandte Inhalt wird unter dem angegebenen Pfad verfügbar gemacht. Dabei soll, entsprechend dem HTTP Standard, mit dem Status `Created` (201) geantwortet werden, wenn die Resource zuvor nicht existiert hat, und mit `No Content` (204), wenn der vorherige Inhalt mit dem übergebenen überschrieben wurde.

`DELETE` Anfragen löschen Resourcen: die Resource unter dem angegebenen Pfad wird gelöscht. Eine `DELETE` Anfrage auf eine nicht existierende Resource wird analog zu `GET` mit `Not Found` beantwortet, das erfolgreiche Löschen mit `No Content`.

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.

> **i** Auch wenn angefragte Pfade Webservern Dateien im Dateisystem entsprechen, ist dies im Allgemeinen nicht der Fall. Im HTTP-RFC wird diese falsche Annahme explizit erläutert. Auch in dieser Aufgabenstellung sollen Sie keine Dateien lesen oder schreiben. Die Resourcen sind nur im Speicher des Programms vorhanden.

Dieses Verhalten können Sie ebenfalls mit `curl` testen. Die folgenden Befehle beispielsweise führen zu in etwa den gleichen Anfragen, welche auch unsere Tests durchführen:

```
curl -i localhost:1234/dynamic/members # -> 404
curl -siT members.txt localhost:1234/dynamic/members # Create resource -> 201
curl -i localhost:1234/dynamic/members # -> 200
curl -iX "DELETE" localhost:1234/dynamic/members # -> 204
curl -i localhost:1234/dynamic/members # -> 404
```

3. Freiwillige Zusatzaufgaben

Ihre Implementierung ist nun ein funktionierender Webserver! In dieser Aufgabenstellung haben wir einige Annahmen erlaubt, um die Implementierung zu erleichtern. Unter erlaubt dies, Anfragen seriell zu bearbeiten. Für die kommenden Aufgaben wird Ihr Programm parallel mit mehreren Sockets interargieren müssen. Als Vorbereitung dafür können Sie Ihre Lösung so erweitern, dass sie dies bereits beherrscht. Hierfür bietet sich die Verwendung von `epoll(7)` mit dem eine Liste von Filedescriptors auf verfügbare Aktionen abgefragt werden kann, oder die Verwendung von Threads via `pthread`, wobei Sie auf die Synchronisierung der Threads acht geben müssen.

A. Abgabeformalitäten

Die Aufgaben sollen von Ihnen in Gruppenarbeit mit jeweils *bis zu drei Kursteilnehmern gelöst* werden. Um die Verwaltung auf ISIS einfach zu halten muss *jedes Gruppenmitglied seine Abgabe* hochladen. Fügen Sie Ihrer Abgabe eine Datei `group.txt` hinzu, die Name und Matrikelnummer aller Gruppenmitglieder enthält. Dadurch bleibt die reguläre Gruppenarbeit bei unserem Plagiarismuscheck unbeachtet.

Ihre Abgaben laden Sie auf ISIS bis zur entsprechenden *Abgabefrist* hoch. Beachten Sie bei der Abgabe, dass die Abgabefrist fix ist und es *keine Ausnahmen für späte Abgaben* gibt. Planen Sie also einen angemessenen Puffer zur Frist hin ein um Eventualitäten, die Ihre Abgabe verzögern könnten, vorzubeugen. In Krankheitsfällen kann die Bearbeitungszeit angepasst werden, sofern diese ärztlich belegt sind.

Abgaben werden nur im `.tar.gz`-Format akzeptiert. Sie können ein entsprechendes Archiv erstellen, indem Sie das folgende Snippet an Ihre `CMakeLists` anhängen und im Build-Ordner `make package_source` ausführen:

```
# Packaging
set(CPACK_SOURCE_GENERATOR "TGZ")
set(CPACK_SOURCE_IGNORE_FILES ${CMAKE_BINARY_DIR} /\\..*$ .git .venv)
set(CPACK_VERBATIM_VARIABLES YES)
include(CPack)
```

B. Tests und Bewertung

Die einzelnen Tests finden Sie jeweils in der Vorgabe als `test/test_praxisX.py`. Diese können mit `pytest` ausgeführt werden:

```
pytest test # Alle tests ausführen
pytest test/test_praxisX.py # Nur die Tests für einen bestimmten Zettel
pytest test/test_praxis1.py -k test_listen # Limit auf einen bestimmten Test
```

Sollte `pytest` nicht auf Ihrem System installiert sein, können Sie dies vermutlich mit dem Paketmanager, beispielsweise `apt`, oder aber `pip`, installieren.

Ihre Abgaben werden anhand der Tests der Aufgabenstellung automatisiert bewertet. Beachten Sie, dass Ihre Implementierung sich nicht auf die verwendeten Werte (Node IDs, Ports, URIs, ...) verlassen sollte, diese können zur Bewertung abweichen. Darüber hinaus sollten Sie die Tests nicht verändern um sicherzustellen, dass die Semantik nicht unbeabsichtigt erweise verändert wird. Eine Ausnahme hierfür sind natürlich Updates der Tests, die wir gegebenenfalls ankündigen um eventuelle Fehler zu auszubessern.

Die Bewertung führen wir auf den Ubuntu 20.04 Systemen der EECS durch, welche auch für Sie sowohl vor Ort, als auch via SSH zugänglich sind. Daher können Sie Ihre Implementierung vor der Abgabe in der gleichen Umgebung testen. Wir empfehlen dies, um sicher zu gehen, dass es keine subtilen Unterschiede zwischen der Testumgebungung und Ihrer lokalen. Derartige Unterschiede, beispielsweise in den vorhandenen Bibliotheken, können im Zweifel auch abweichendes Verhalten der Tests zur Folge haben.

C. Adressen parsen

Die folgende Funktion kann verwendet werden um eine menschenlesbare Adresse für die Benutzung mit einem Socket zu parsen:

```c
/**
 * Derives a sockaddr_in structure from the provided host and port information.
 *
 * @param host The host (IP address or hostname) to be resolved into a network address.
 * @param port The port number to be converted into network byte order.
 *
 * @return A sockaddr_in structure representing the network address derived from the host and port.
 */
static struct sockaddr_in derive_sockaddr(const char* host, const char* port) {
    struct addrinfo hints = {
        .ai_family = AF_INET,
    };
    struct addrinfo *result_info;

    // Resolve the host (IP address or hostname) into a list of possible addresses.
    int returncode = getaddrinfo(host, port, &hints, &result_info);
    if (returncode) {
        fprintf(stderr, "Error parsing host/port");
        exit(EXIT_FAILURE);
    }

    // Copy the sockaddr_in structure from the first address in the list
    struct sockaddr_in result = *((struct sockaddr_in*) result_info->ai_addr);

    // Free the allocated memory for the result_info
    freeaddrinfo(result_info);
    return result;
}
```

D. Vereinfachende Annahmen für HTTP

Für die Lösung können Sie die folgenden Annahmen treffenden:

- Anfragen und Antworten sind stets kleiner als 8192 B.
- Anfragen und Antworten enthalten nie mehr als 40 Header.
- Header Namen und Werte sind je maximal 256 B lang.
- Ihr Server muss nicht mehr als 100 Resourcen speichern.
- Der `Connection`-Header kann ignoriert werden. Die Verbindung zum Client soll im Fehlerfall, bei ungültigen Anfragen, oder durch den Client geschlossen werden.

- Verwenden (und interpretieren) Sie ausschließlich den `Content-Length`-Header um die Größe des Payloads zu kommunizieren.