Rechnernetze und verteilte Systeme
Praxis 1: Webserver

Fachgruppe Telekommunikationsnetze (TKN)

Beginn Bearbeitung: 04.11.2024, 00:00 Uhr
Abgabe: **08.12.2024 23:59 Uhr**
Stand: 14. November 2024

Formalitäten

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

Für die erste Praxis-Hausaufgabe implementieren Sie in C einen Webserver, den sie mit den folgenden Aufgaben erstellen und Stück für Stück erweitern. Jede Teilaufgabe baut aufeinander auf und ist mit einem vorgegebenen Test überprüfbar. Ihre finale Abgabe wird automatisiert anhand der Tests bewertet. Bei Unklarheiten und Fragen zum Projektsetup and Abagbe, schauen Sie sich bitte zuerst Praxis 0 erneut an! 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. Ein einfacher Texteditor wie `kate`, ein Compiler (`gcc`), und ein Debugger (`gdb`) sind allerdings ebenfalls völlig ausreichend.

1. 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`. Dazu müssen Sie ein entsprechendes Target `webserver` in der CMakeLists.txt konfigurieren. Diesen Prozess haben Sie bereits in Praxis 0 geübt und getestet.

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.

Bei Unklarheiten zum Projektsetup and Abgabeprozess, schauen Sie sich erneut Praxis 0 an! Im ISIS-Kurs finden Sie zusätzliche Anleitungen & Literatur, insbesondere den Beej's Guide to Network Programming Using Internet Sockets, für die Bearbeitung der Aufgaben!

2. Aufgaben

Die im folgenden beschriebenen Aufgaben dienen als Leitfaden zur Implementierung. Insgesamt führen die verschiedenen Teilaufgaben zu einer vollständigen Implementierung eines Webservers. Beachten Sie hierzu insbesondere auch die Hinweise in Anhang B.

Teilweise werden spätere Aufgaben den vorigen widersprechen, da sich die Aufgabenstellung im Verlauf weiter konkretisiert. Die Tests für die Aufgaben sind daher so konzipiert, dass frühere Tests die Implementation auch von späteren Teilaufgaben noch als korrekt akzeptieren. Für das Implementieren Ihres Programms empfehlen wir **unbedingt einen Debugger**, sowie 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 D).

Unsere Tests (siehe Anhang B) starten jeweils eine eigene Instanz Ihres Servers. Wenn der gleiche Port unmittelbar nach dem Ende eines Prozesses wiederverwendet werden soll, kann es sein, dass das Betriebssystem diesen noch nicht freigegeben hat. Im Detail werden Sie dieses Verhalten im TCP-Kapitel behandeln. Setzen Sie daher die `SO_REUSEADDR`-Socket-Option mittels `setsockopt`, um in solchen Situationen Probleme beim Verwenden des Ports zu vermeiden.

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 mit einem CRLF terminiert. Requests, die einen Payload enthalten, kommunizieren dessen Länge über den `Content-Length`-Header. Fehlt dieser Header, enthält der Request keinen Payload.

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 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:

```
Reque
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\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 E) 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
- Ihre Header nicht dem Muster `<Key>: <Value>` entsprechen.

2.6. Statischer Inhalt

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

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 Ressourcen 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 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. Dabei soll, entsprechend dem HTTP Standard, mit dem Status `Created` (201) geantwortet werden, wenn die Ressource zuvor nicht existiert hat, und mit `No Content` (204), wenn der vorherige Inhalt mit dem übergebenen überschrieben wurde.

`DELETE` Anfragen löschen Ressourcen: die Ressource unter dem angegebenen Pfad wird gelöscht. Eine `DELETE` Anfrage auf eine nicht existierende Ressource 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.

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

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 Ressourcen 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
```

A. Abgabeformalitäten

Die Aufgaben können in Gruppen aus **ein bis drei Mitgliedern** bearbeitet (und abgegeben) werden. Dazu wählen Sie jeweils in der ersten Woche der Bearbeitungszeit eine (neue) Abgabegruppe auf ISIS, auch dann wenn Sie alleine arbeiten. Die getätigte Gruppenwahl gilt jeweils für den gesamten Bearbeitungszeitraum eines Praxis-Zettel.

Ab der jeweils zweiten Woche der Bearbeitungszeit können Sie Ihre Lösung abgeben. Ab diesem Zeitpunkt kann die Gruppenwahl **nicht mehr** verändert werden! Sollten Sie zu diesem Zeitpunkt keine Abgabegruppe gewählt haben, können Sie für diesen Praxis-Zettel **leider keine Lösung abgeben!** Die Gruppenabgabe muss von einem der Gruppenmitglieder in ISIS hochgeladen werden und gilt dann für die gesamte Gruppe. Ohne eine Abgabe auf ISIS erhalten Sie keine Punkte!

Es werden nur mit CMake via `make -C build package_source` erstellte Abgabearchive im `.tar.gz`-Format akzeptiert. Beachten Sie, dass eine falsche Dateiendung nicht einfach umbenennen können. **Wir empfehlen dringend**, ihr so erstelltes Archiv einmal zu entpacken, und die Tests auszuführen. So können Sie viele Fehler mit ihrer Projektkonfiguration vor der Abgabe erkennen und vermeiden.

Ihre Abgaben können ausschließlich auf ISIS und bis zur entsprechenden *Abgabefrist* abgegeben werden. Sollten Sie technische Probleme bei der Abgabe haben, informieren Sie uns darüber unverzüglich und lassen Sie uns zur Sicherheit ein Archiv Ihrer Abgabe per Mail zukommen. Beachten Sie bei der Abgabe, dass die **Abgabefrist fix ist** und es **keine Ausnahmen für zu späte Abgaben oder Abgaben via E-Mail** gibt. Planen Sie einen angemessenen Puffer zur Frist hin ein. In Krankheitsfällen kann die Bearbeitungszeit angepasst werden, sofern sie uns baldmöglichst ein Attest zusenden.

B. Tests und Bewertung

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

```
1 python3 -m pytest test # Alle tests ausführen
2 python3 -m pytest test/test_praxisX.py # Nur die Tests für einen bestimmten Zettel
3 python3 -m 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. Analog müssen Sie zusätzlich auch das Paket `pytest-timout` installieren, damit Sie die Tests ausführen können. Alle Abhängigkeiten finden Sie auch in der Datei `requirements.txt`.

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 unbeabsichtigterweise verändert wird. Eine Ausnahme hierfür sind natürlich Updates der Tests, die wir gegebenenfalls ankündigen, um eventuelle Fehler zu auszubessern.

Wir stellen die folgenden Erwartungen an Ihre Abgaben:

- Ihre Abgabe muss ein `CMake` Projekt sein.
- Ihre Abgabe muss eine `CMakeLists.txt` enthalten.
- Ihr Projekt muss ein Target entsprechend der oben genannten Definition haben (z.B. `hello_world`) mit dem selben Dateinamen (z.B. `hello_world`) (case-sensitive) erstellen.
- Ihre Abgabe muss interne CMake Variablen, insbesondere `CMAKE_BINARY_DIR` und `CMAKE_CURRENT_BINARY_DIR` unverändert lassen.
- Ihr Programm muss auf den EECS Poolrechnern[2] korrekt funktionieren.
- Ihre Abgabe muss mit CPack (siehe oben) erstellt werden.
- Ihr Programm muss die Tests vom jeweils aktuellen Zettel bestehen, nicht auch vorherige.
- Wenn sich Ihr Program nicht deterministisch verhält, wird auch die Bewertung nicht deterministisch sein.

Um diese Anforderungen zu überprüfen, sollten Sie:

- das in der Vorgabe enthaltene `test/check_submission.sh`-Script verwenden:

```
./test/check_submission.sh praxisX
```

- Ihre Abgabe auf den Testsystemen testen.

Fehler, die hierbei auftreten, werden dazu führen, dass auch die Bewertung fehlschlägt und Ihre Abgabe entsprechend benotet wird.

C. Tests debuggen

Standardmäßig erstellen die Tests eine interne Instanz eines Webservers, sie nutzen also nicht ihre aktuell laufende Instanz. Um Ihre **eigene, bereits laufende** Executable zu debuggen, können Sie pytest mit dem Zusatzflag *debug_own* ausführen. Dabei müssen Sie auch die automatischen Timeouts während des debuggings deaktivieren:

```
python3 -m pytest test --debug_own --timeout=99999 --timeout_override
```

[2] 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.

D. 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;
}
```

E. 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 Ressourcen 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.
- Anfragen müssen keinen `Host`-Header setzen.

F. Freiwillige Zusatzaufgaben

Ihre Implementierung ist nun ein funktionierender Webserver! In dieser Aufgabenstellung haben wir einige Annahmen erlaubt, um die Implementierung zu erleichtern. Unter anderem erlaubt dies, Anfragen seriell zu bearbeiten. Für die kommenden Aufgaben wird Ihr Programm parallel mit mehreren Sockets interagieren müssen. Als Vorbereitung dafür können Sie Ihre Lösung so erweitern, dass sie dies bereits beherrscht. Hierfür bietet sich die Verwendung von `poll` 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.