Systemnahe Programmierung SS24

Projekt Webserver

Abgabetermin: 19.Mai 2024

Erreichbare Punkteanzahl: 80

Das Projekt kann als Einzelperson oder als Team von zwei Personen durchgeführt werden, dabei sind unterschiedliche Aufgaben zu implementieren.

Aufgabenstellung

Es ist ein Webserver zu implementieren, der einfache Requests abarbeiten kann. Wir verwenden eine vereinfachte Version des HTTP Protokolls in der Version 1.1.

Der Webserver soll in der Lage sein, mehrere Requests gleichzeitig abzuarbeiten und Dateien aus einem festgelegten Verzeichnis zu lesen und an den Client zu senden.

Das Projekt ist in mehreren Stufen aufgebaut, die schrittweise zum fertigen Webserver führen. Für Zweier-Teams sind mehr dieser Stufen zu implementieren als für Einzelpersonen.

Stufe 1

Implementieren sie einen grundlegenden Netzwerk-Server, der Verbindungsanfragen entgegen nimmt, einen eigenen Prozess dafür startet, die empfangenen Daten auf `stdout` ausgibt und bei Ende der Kommunikation beendet wird.

Lesen sie die Basiskonfiguration aus einer Konfigurationsdatei ein und verwenden sie die externe Bibliothek `libconfig` dafür. Das Installationsverzeichnis (Toplevel-Verzeichnis) wird durch eine Umgebungsvariable festgelegt (z.B.: WEBBY_ROOT). Lesen sie diese Umgebungsvariable ein. Die Konfigurationsdatei muss sich in dem durch die Umgebungsvariable bestimmten Verzeichnis befinden.
Beispiel:

> Umgebungsvariable definieren:
>
> `export WEBBY_ROOT=/home/user/webby`
>
> Konfigurationsdatei befindet sich in: `/home/user/webby/webby.cfg`

> **Tip**
>
> Im Makefile des Templates wird bereits die Umgebungsvariable `WEBBY_ROOT` definiert, passen sie das Verzeichnis nach ihren Wünschen an.

Folgende Parameter sollen aus der Konfigurationsdatei gelesen werden:

Parameter		Default

www-root	Das Verzeichnis, in dem die HTML und sonstige Dateien enthalten sind, die vom Webserver gelesen werden. Alle Request URIs (Ausnahme: Services, siehe Stufe 8) sind relativ zu diesem Verzeichnis. Dieser Parameter muss in der Konfigurationsdatei enthalten sein, wenn nicht, dann ist eine Fehlermeldung auszugeben und das Programm zu beenden.
	Beispiel:
	`www-root : /home/user/html`
	Request URI: `forms/kontakt.html`
	Aufgelöster Dateiname:
	`/home/user/html/forms/kontakt.html`
		Kein

error_page	Zeigt auf eine HTML Seite, die im Fehlerfall als Response gesendet wird. Ist relativ zu `www-root`.	error.html
notfound_page	Zeigt auf eine HTML Seite, die im Fall, dass eine Datei/ein Service nicht gefunden wird, als Response gesendet wird. Ist relativ zu `www-root`.	404.html
port	Port, über den der Webserver erreichbar ist.	17000

> **Note**
>
> `error_page`, `notfound_page` und `port` sind optional, d.h. wenn sie nicht in der Konfigurationsdatei enthalten sind, dann sollten sie mit dem angeführten Defaultwert hinterlegt werden. Falls in der Konfigurationsdatei Einträge für `error_page` und `notfound_page` vorhanden sind, muss auf jeden Fall noch geprüft werden, ob die angegebenen Dateien tatsächlich existieren und lesbar sind!

3. Implementieren sie ein sauberes Signalhandling, indem sie mindestens diese Signale behandeln:

Signal	Behandlung
SIGCHLD	Einen Signalhandler einrichten, der entsprechend auf beendete Child-Prozesse reagiert.
SIGTERM	Einen Signalhandler einrichten, der den Serverprozess sauber beendet.
SIGINT	Einen Signalhandler einrichten, der den Serverprozess sauber beendet.
SIGHUP	Ignorieren
SIGPIPE	Ignorieren

Stufe 2

Lesen sie die empfangenen Daten in einen Buffer und analysieren sie den empfangenen HTTP Request. Extrahieren sie alle notwendigen Daten (Request-Metod, Request-URI, Header, Body) und verifizieren sie diese.
Als "Request-Method" müssen sie nur `GET` und `POST` akzeptieren, die anderen Methoden können sie ignorieren.

Dekodieren sie mindestens diese Request-Header und speichern sie deren Information:

- "Content-Length"
  Länge der Daten im Body-Abschnitt (0 falls nicht vorhanden)
- "Host"
  Hostname des Clients
- "User-Agent"
  Bezeichnung des Client-Programms (z.B.: `curl/7.81.0`)
- "Accept"
  Welche Mime-Typen der Client akzeptiert (normalerweise `*/*`)
- "Content-Type"
  Mime-Type des Inhalts im Body-Abschnitt (falls vorhanden)

Content-Type kann folgende Formen annehmen:

> Mime-Type oder
>
> Mime-Type; Charset

Das Feld "Charset" können sie ignorieren, der Server muss jedoch damit umgehen können, wenn es mitgesendet wird.
Beispiel:

> `text/html`
> `text/html; charset=utf-8`

Die Werte der Header müssen vorerst nicht interpretiert werden, sondern nur erkannt und im Logging angezeigt werden! Speichern sie die Werte im Request für eine spätere Verwendung.

> **Note**
>
> Request-Header sind grundsätzlich optional, der Client muss keine Header mitsenden. Die oben genannten Header müssen sie jedoch interpretieren und speichern, falls sie im Request vorhanden sind. Je nach Client-Programm werden noch wesentlich mehr Request-Header mitgesendet. Sie können diese mit dem Logging auch anzeigen und so den gesamten Request sehen.

Request-URI

Die Request-URI kann mit URL-Encoding codiert sein, deshalb muss sie vor der Verwendung decodiert werden. Dazur können sie die im Template enthaltene Funktion `decodeUri()` verwenden.

```
1  #include <url.h>
2
3  char *decodeUri(char *uri);
```

`decodeUri()` retourniert einen Zeiger zur dekodierten Version von uri. Im Fehlerfall wird NULL zurück gegeben.

Wichtig: der durch die Funktion allozierte Speicher ist nach Benutzung freizugeben.

Response

Senden sie eine einfache Textmeldung (Mockup) im Response-Body mit dem Code 200 zurück. Im Fehlerfall senden sie eine Fehlermeldung und den Code 500 zurück.

Stufe 3

Implementieren sie das Senden statischer Files.

Die Request-URI ist grundsätzlich ein Dateiname oder ein Verzeichnis, relativ zum `www-root` Verzeichnis. Wenn es sich um ein Verzeichnis handelt, können sie es in dieser Stufe ignorieren und einen Mockup-Response zurücksenden.
Mockup-Response bedeutet: im Response-Body wird ein einfacher, statischer Text (z.B.: "Ist Verzeichnis") gesendet.

Sie können das vereinfachte URI Format (ohne Fragment) verwenden:

```
path[?parameter]
```

wobei sie in dieser Stufe die Parameter (also alles nach `?`) ignorieren können. Die Komponente `path` interpretieren sie als Dateiname oder Verzeichis relativ zum `www-root` Verzeichnis. Der Pfad beginnt immer mit einem '/'.

Vor dem Lesen und Senden der Datei prüfen sie ob die Datei vorhanden und lesbar ist. Im Fehlerfall senden sie einen entsprechenden Error-Response (`404 Not found` oder `403 Forbidden`) und beenden sie die Verbindung und den Prozess. Ansonsten lesen sie das File ein und senden den Inhalt als Response-Body an den Client zurück.

Senden sie jetzt einen vollständigen Response an den Client zurück.
Das bedeutet: eine vollständige Response-Line, alle unten angeführten Response-Header und einen entsprechenden Inhalt im Response-Body.

Folgende Header müssen im Response mitgeschickt werden:

- "Server"
  Bezeichnung des Servers mit Versionsnummer (Beispiel: `Webby/1.0`)
- "Date"
  Datum und Zeit zu dem der Response gesendet wurde.
  Format: `"%a,%d-%b-%y %T %z"`
  Beispiel: `Date: Thu,11-Apr-24 11:07:32 +0200`
- "Content-Length"
  Länge des gesendeten Body-Inhalts in Bytes.
- "Content-Type"
  Mime-Type des gesendeten Inhalts

Nehmen sie für "Content-Type" vorerst immer `text/html` an.

Stufe 4

Bestimmen sie zu jedem Datei-Response (aus Stufe 3) den passenden *Mime-Type* und senden sie diesen im Header Header "Content-Type" mit. Bestimmen sie den Mime-Type anhand der Dateiendung.

Falls keine Dateiendung vorhanden ist, nehmen sie `application/octet-stream` an.

Implementieren sie die Erkennung für folgende Mime-Types:

Mime-Type	Dateiendung
text/plain	.txt
text/html	.html
text/css	.css
text/javascript	.js
text/csv	.csv
image/gif	.gif
image/jpeg	.jpg
image/png"	.png
application/json	.json
application/pdf	.pdf
application/zip	.zip
application/octet-stream	

> **Important**
>
> Senden sie den Mime-Type auch für alle Fehleresponses mit! Verwenden sie `text/html` wenn es sich um eine Datei wie `error.html` oder `404.html` handelt, sonst `text/plain`.

Stufe 5

Implementieren sie das automatische Erkennen von Indexdateien in Verzeichnissen.

Wenn es sich bei der Request-URI nicht um eine Datei, sondern um ein Verzeichnis handelt, gehen sie wie folgt vor:

Ist im Verzeichnis eine Datei `index.html` vorhanden?

- Wenn ja, dann senden sie diese als Response.
- Wenn nein, dann retournieren sie einen entsprechenden Fehler-Response (404).

Beispiel:

`www_root = /home/user/html`

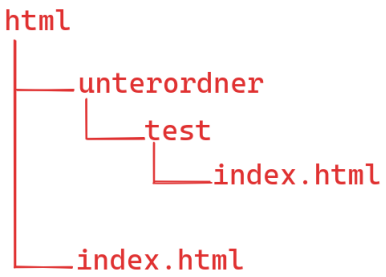
Es existieren im *www_root* die Unterverzeichnisse:

`unterordner` und `unterordner/test`

Sowie die Dateien:

`index.html` und `unterordner/test/index.html`

Verzeichnisbaum ab *www_root*:



Beispiel-Requests:

Request URI	Resultat (Response)
/	/home/user/html/index.html
/unterordner	404 Not Found
/unterordner/test	/home/user/html/unterordner/test/index.html
/unterordner/hmmpf	404 Not Found

Stufe 6

Request-Parameter werden bei GET Requests in der URI codiert indem sie an den Pfad der Ressource angehängt werden (durch '?' getrennt).

URI Syntax:

```
pfad(?(param_name=param_value)(&param_name=param_value)*)+
```

Aufgabe: Dekodieren sie die Request-URI vollständig. Erkennen sie URI Parameter und extrahieren sie deren Werte.

Beispiel:

/forms/adduser?id=3&name=mustermann&mail=muma@gmail.at

enthält folgende Parameter:

Name	Wert
id	3
name	mustermann
mail	muma@gmail.at

Geben sie die Parameter als Logging-Einträge (INFO) aus.

Beispiel: `Tue,16-Apr-24 14:31:17 +0200 | INFO | URI PARAMETER: id 3`

Stufe 7

Implementieren sie die Verarbeitung von `POST` Requests. Die Übergabe von Parametern findet hier im Request-Body statt. Das Format der Parameter ist nicht festgelegt und kann frei gewählt werden, jedoch ist der Request-Header "Content-Type" entsprechend gesetzt.

In unserem Fall gehen sie davon aus, dass die Parameter in der ersten Zeile im Body, getrennt durch Leerzeichen enthalten sind.

Das Beispiel aus Stufe 6 würde jetzt auf einen `POST` Request angewendet so aussehen:

URI: `/forms/adduser`

Body: `3 mustermann muma@gmail.at`

> **Note**
>
> Die `POST` Methode muss nur im Zusammenhang mit den in Stufe 8 vorgestellten Services implementiert werden (nicht für “normale” Inhalte).

Stufe 8

Implementieren sie das Ausführen von Shell-Scripts als "Services".

Jedesmal wenn eine Request-URI mit dem Pfad-Präfix `/services` beginnt, soll der Rest des URI-Pfades als Service interpretiert werden. Ein “Service“ ist ein Shell-Script, das in einem dedizierten Verzeichnis liegt. Ein Service nimmt Parameter entgegen und schreibt seinen Output (als HTML) auf `stdout`. Dieser Output wird vom Webserver gelesen und als Ergebnis eines Requests zum Client zurück übertragen.

Die Services sollen sowohl mit `GET` (Parameter als Teil der URI) als auch mit `POST` Requests (Parameter im Body) funktionieren.

Fügen sie in der Konfigurationsdatei einen Parameter `service_dir` hinzu. Dieser Parameter zeigt auf ein Verzeichnis, das die Shellscripts enthält.

Beispiel:

```
service_dir = /home/user/services
```

Request: http://localhost:17000/services/testservice?id=3&prompt=hello

Sucht in dem durch die Konfigurations-Variable `service_dir` festgelegten Verzeichnis nach der Datei `testservice`. Ist diese vorhanden und ausführbar, wird sie mit den Parametern `3` und `hello` ausgeführt.
In diesem Fall wird das Script `/home/user/services/testservice` ausgeführt, mit den Werten `3` und `hello` als Parametern (`$1` und `$2`).

Parameterübergabe:

An das Shell-Script werden nur die Werte der Parameter und nicht deren Namen in der Reihenfolge des Eintreffens übergeben. Sie stehen dann im Shell-Script als `$1 ... $n` zur Verfügung.

Beispiel für einen GET Request:

URI: http://localhost:17000/services/testservice?id=3&prompt=hello

Führt folgendes Script aus:

`/home/user/services/testservice "3" "hello"`

Beispiel für einen POST Request:

URI: http://localhost:17000/services/testservice

Body: 3 hello

Führt folgendes Script aus:

`/home/user/services/testservice "3" "hello"`

Bemerkungen:

Prüfen sie ob das angegebene Script tatsächlich vorhanden ist und sowohl lesbar als auch ausführbar ist.

Verwenden sie `popen()` um das Script mit den Parametern auszuführen und lesen sie den Output des Scripts aus der Pipe ein. Das Shell-Script muss seinen Output auf `stdout` schreiben (siehe das Beispiel testservice im Template) und wird vom Webserver über die Pipe eingelesen und an den Socket weitergegeben.

Textfelder und Pfade sollten sie mit " " klammern, damit Whitespaces keine Probleme machen.

Beispiel für eine gültige Befehlszeile: `"\\"/home/user/services/testservice\\" 3 \\"hello\\""`

Das Ergebnis eines Services hat den Mime-Type `text/html`.

Im Template finden sie dazu im Verzeichnis `services` die vorbereiteten Scripts `testservice` und `postservice`.

> **Tip**
>
> Das Shell-Script `testservice` liefert den Output des Befehls `df -k` (verfügbarer Diskspace) aufbereitet zurück. Verwenden sie es als Vorlage und zum Testen.

Gruppenarbeiten

Das Projekt kann als Einzel- oder als Gruppenarbeit durchgeführt werden. Die Gruppengröße ist auf 2 Personen beschränkt.

Stufe	Auszuführen durch
Stufe 1	Alle
Stufe 2	Alle
Stufe 3	Alle
Stufe 4	Alle
Stufe 5	Alle

Stufe	Auszuführen durch
Stufe 6	nur Gruppen
Stufe 7	nur Gruppen
Stufe 8	nur Gruppen

Rahmenbedingungen

Allgemein

- Verwenden sie das im Moodle im Abschnitt "**Projektarbeit**" hinterlegte **Template**.
- Folgende Signale müssen (wie oben beschrieben) behandelt werden:
  `SIGCHLD, SIGTERM, SUGHUP, SIGPIPE, SIGINT`

Logging

Implementieren sie ein durchgängiges Logging, dessen Meldungen auf `stderr` ausgegeben werden.

Format:

<Datum> "|" <Typ> "|" <Meldung>

Datum: `Tag, TT-MON-YY Time TZ` Beispiel: `Tue,16-Apr-24 14:28:35 +0200`

Typ: `INFO` oder `ERROR`
Grundsätzlich ist `ERROR` für interne Fehler zu verwenden, wie Fehler beim Aufruf von Funktionen aus der Standard-Library (z.B.. open() usw.). Das sind alle Fehler, welche die interne Funktion des Servers betreffen.
Für alle anderen Meldungen soll der Typ `INFO` verwendet werden, als auch für Fehler, die sich aus dem Request ergeben (z.B.: Datei nicht gefunden usw.)

Meldung: Eine aussagekräftige Beschreibung des aufgetreten Problems.

Beispiel:

```
1
2  Tue,16-Apr-24 14:28:35 +0200 | INFO | Config: www_root =
   /home/user/webby/html
3  Tue,16-Apr-24 14:28:35 +0200 | INFO | Config: service_dir =
   /home/user/webby/services
4  Tue,16-Apr-24 14:28:35 +0200 | INFO | Config: error_page = error.html
5  Tue,16-Apr-24 14:28:35 +0200 | INFO | Config: notfound_page = 404.html
6  Tue,16-Apr-24 14:28:35 +0200 | INFO | Config: port = 17000
7  Tue,16-Apr-24 14:28:35 +0200 | INFO | Starting server: listening on port
   17000
8  Tue,16-Apr-24 14:31:17 +0200 | INFO | New Request from client: 127.0.0.1
9  Tue,16-Apr-24 14:31:17 +0200 | INFO | Method: GET
10 Tue,16-Apr-24 14:31:17 +0200 | INFO | HOST:  localhost
11 Tue,16-Apr-24 14:31:17 +0200 | INFO | URI: /test/hallo.sh
```

```
Tue,16-Apr-24 14:31:17 +0200 | INFO | URI PARAMETER: page 3
Tue,16-Apr-24 14:31:17 +0200 | INFO | URI PARAMETER: name Meier
Tue,16-Apr-24 14:31:17 +0200 | INFO | USER-AGENT:  Mozilla/5.0 (X11; Linux
x86_64; rv
Tue,16-Apr-24 14:31:17 +0200 | INFO | ACCEPT:
text/html,application/xhtml+xml,application/xml;q=0.9
Tue,16-Apr-24 14:31:17 +0200 | INFO | CONTENT-LENGTH: 0
Tue,16-Apr-24 14:31:17 +0200 | INFO | serving file: /test/hallo.html

```

Was ist zu loggen (Mindestanforderung, mehr ist möglich):

- Alle internen Fehler (Type `ERROR`)
- Informative Events (Typ `INFO`):
- Aufbau einer neuen Verbindung nach `fork()` mit der IP-Adresse des Clients!
- Eingelesene Werte aus Konfigurationsdatei
- Request:
  - Method
  - URI und URI-Parameter (Parameter nur ab Stufe 6)
  - Header
- Response: Dateipfad der im Response gesendeten (falls) Datei.

> **Tip**
>
> Nutzen sie das Logging intensiv während der Entwicklung zum Debuggen. Loggen sie alle relevanten Daten, später können sie jene Log-Einträge die sie nicht für den tatsächlichen "Betrieb" benötigen wieder entfernen.

Fehlerhandling

Im Fehlerhandling werden zwei Fälle unterschieden:

1. Eine Verbindung mit dem Client ist noch nicht zustande gekommen

Bei Problemen die auftreten, bevor eine Verbindung zustande kommt (Startup usw.) geben sie eine entsprechende Meldung im Log-System aus. Je nach Schwere des Fehlers beenden sie das Programm oder implementieren ein sinnvolles Defaultverhalten.

2. Eine Verbindung mit dem Client wurde aufgebaut

Sobald eine Verbindung mit dem Client zustande gekommen ist, müssen sie einen Response mit einer passenden Fehlerseite (error oder 404 usw.) zurücksenden. Zusätzlich wird eine Fehlermeldung im Log-System ausgegeben.
Anschließend können sie den Prozess mit *exit()* beenden (Cleanup nicht notwendig).

Behandeln sie mindestens diese Fehler und senden sie die beschriebenen Responses zurück:

Fehler	Response-Code	Response Body
Ungültige Request URI	400 Bad Request	Parameter: error_page Default: error.html
Zugriff auf Datei oder Verzeichnis ist nicht möglich (fehlende Berechtigung)	403 Forbidden	Parameter: error_page Default: error.html
Verzeichnis, Datei oder Service-Script nicht gefunden oder keine Berechtigung	404 Not Found	Parameter: notfound_page Default: 404.html
Alle anderen Fehler	500 Internal Server Error	Parameter: error_page Default: error.html

Im Template finden sie zwei Vorlagen für allgemeine Fehler: `error.html` und für den "Not Found" Fehler: `404.html`. Sie können diese oder auch eigene verwenden. Wichtig ist, dass mindestens diese beiden Dateien vorhanden sind und im Fehlerfall als Response an den Client gesendet werden.

Ablauf für das Erstellen eines Fehler-Responses:

1. Lesen der Parameter `error_page`, `notfound_page` aus der Konfigurationsdatei. Falls keine Einträge vorhanden sind, verwenden sie die angeführten Defaultwerte.
2. Prüfen ob die in den oben genannten Parametern angeführten Dateien vorhanden und lesbar sind. Falls nicht, dann wird anstelle der entsprechenden Datei ein einfacher Text-Response mit einer passenden! Fehlermeldung im Body zurückgesendet.

> **Important**
>
> Testen sie beide Szenarien: Konfigurationsparameter vorhanden/nicht vorhanden.

Coding-Rules

Folgende Coding-Rules fließen in die Bewertung ein:

- Returnwerte von Funktionen aus der Standardbibliothek und anderen externen Bibliotheken auf Fehler prüfen und diese entsprechend behandeln (Loggen und/oder Fehler-Response)
- Alle Ressourcen die nicht benötigt werden, sobald wie möglich freigeben. Insbesondere auch den Heap-Speicher der alloziert wird, wieder korrekt freigegeben.
- Prüfen sie bei Strings und Zeigern als Rückgabewert von Funktionen ob sie NULL sind bevor sie sie verwenden.
- Verwenden sie idealerweise Strukturen um Requests und URIs abzubilden.
- In jeder Datei im Header den Gruppennamen anführen.

- Teilen sie den Sourcecode wo sinnvoll in mehrere Module (Dateien) auf.
  Erstellen sie zu jeder `.c` Datei die passende Header-Datei (Ausnahme: jene Datei die `main()` enthält) und verwenden sie die Struktur (`#ifdef _MODUL_NAME` ...) wie wir sie behandelt haben.

Testen

Im Template ist ein Test-Script `test/runTests.sh` enthalten, das sie mit `make test` ausführen können.

> **Important**
>
> Alle Testfälle, die im Template enthalten sind müssen fehlerfrei durchlaufen.

In der Datei `tests.dat` sind die Testfälle enthalten, die von diesem Script durchgeführt werden. Das Format der Einträge ist:

<Method>URL<Expected result>

Method: G|P .... Request-Method GET oder POST

URL: URL mit Parametern

Expected result: Text der im Response-Body vorkommen muss

Beispiel: `G$http://localhost:17000/$<!DOCTYPE html>`

Definieren sie eigene Testfälle nach Bedarf dazu.

Für jene, die den Webserver als Einzelperson implementieren, müssen die Testfälle die mit 'P' beginnen (POST) nicht durchgeführt werden (auskommentieren mit `#`).

Die Testfälle funktionieren mit den im Template enthaltenen Dateien in den `html` und `services` Verzeichnissen. Wenn sie eigene Testfälle implementieren, bzw. die in den URLs vorkommenden Dateien ändern müssen sei ev. auch die zugehörigen Testfälle anpassen (Expected result).

Abhängigkeiten

Zum Lesen der Konfigurationsdatei ist die externe Bibliothek *libconfig* zu verwenden. Falls sie auf ihrem System nicht vorhanden ist, können sie diese mit dem Befehl

`sudo apt install libconfig-dev`

installieren.