Praxis 1
Webserver

Fachgruppe Telekommunikationsnetze (TKN)

30. Oktober 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 wie den zu verwendenden Port als Parameter beim Aufruf des Programms übergeben. Beispielsweise soll beim Aufruf `webserver 1234` Port 1234 verwendet werden.

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.

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.

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.

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

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"
```

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.

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

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.

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.

Praxis 2: Statische DHT

Fachgruppe Telekommunikationsnetze (TKN)

14. Dezember 2023

Formalitäten

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

In dieser Abgabe erweitern wir unseren Webserver aus der letzten Aufgabenstellung. Dieser hat Ressourcen bisher einfach im Speicher vorgehalten, dies wollen wir nun durch ein (statische) Distributed Hash Table (DHT)-Backend ersetzen (siehe Abb. 1).

Eine DHT ist eine verteilte Datenstruktur die key-value-Tupel speichert. DHTs sind P2P Systeme und damit selbstorganisierend. Dadurch können Funktionen bereitgestellt werden die sich dynamisch skalieren lassen, ohne explizites oder zentrales Management. Im Detail wurden DHTs in der Vorlesung am Beispiel von Chord besprochen. Auf diesem Protokoll basiert auch diese Aufgabenstellung. Unsere DHT erlaubt Zugriff auf die gespeicherten Daten via HTTP, welches in der vorigen Aufgabenstellung implementiert wurde. Sie können also Ihre bisherige Implementierung weiter verwenden und erweitern. Sollten Sie die vorige Aufgabe nicht vollständig gelöst haben, stellen wir eine Vorlage bereit, die sie stattdessen verwenden können.

Diese DHT speichert die Ressourcen des Webservers, entsprechend sind Keys Hashes der Pfade und Werte der jeweilige Inhalt. Im Detail kann die Aufgabenstellung jedoch alle relevanten Details vor und ist bei widersprüchlichen Definitionen für die Lösung zu beachten.

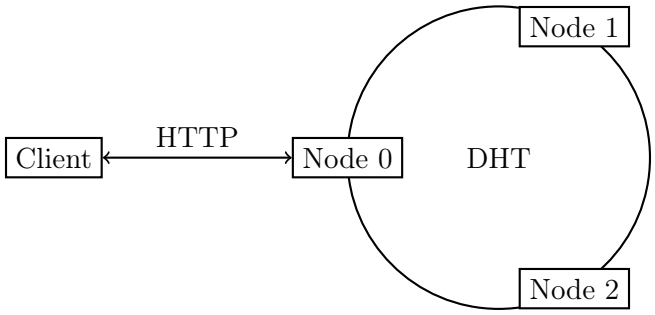
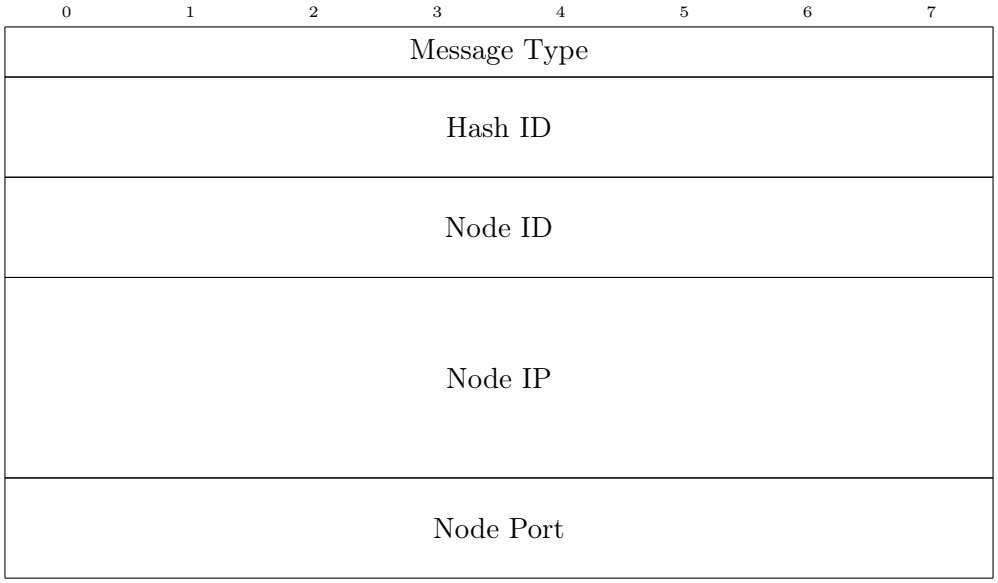


Abbildung 1: Schaubild des Gesamtsystems

Wir beschränken uns in dieser Aufgabenstellung zunächst auf eine auf eine statische DHT. Die Struktur der DHT verändert sich also nicht, weder treten Nodes dem Netzwerk bei, noch verlassen sie es. Die einzelnen Nodes bekommen Ihre jeweiligen Nachbarschaft beim Start übergeben.

Innerhalb der DHT kommunizieren die Nodes miteinander um zu herauszufinden, welche konkrete Node für eine angefragte Resource verantwortlich ist und verweisen den Client an diesen. Dafür verwenden Sie das Chord-Protokoll, welches hier via User Datagram Protocol (UDP) Nachrichten mit dem folgenden Format austauscht:



Der Message Type nimmt für die verschiedenen Nachrichtentypen verschiedene Werte an, und bestimmt so wie die weiteren Daten interpretiert werden. Beispielsweise entspricht 1 einem Reply. Details zu den konkreten Nachrichtentypen werden im Folgenden an den relevanten Stellen erläutert. Alle Werte werden stets in Network Byte Order kodiert.

1. 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 der Aufgabe. 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. Die Projektvorgabe enthält einen *Wireshark Dissector* in der Datei `rn.lua`. Wireshark verwendet diesen automatisch wenn er im

`Personal Lua Plugins`-Verszeichnis (`Help` → `About Wireshark` → `Folders`) abgelegt wird.

1.1. UDP Socket

Innerhalb der DHT tauschen die Peers Nachrichten aus, um deren Funktion bereitzustellen. Während HTTP auf TCP setzt um eine zuverlässige Verbindung zwischen zwei Kommunikationspartnern herzustellen, verwendet die DHT UDP. UDP ist nicht verbindungsorientiert und erlaubt daher, einzelne Nachrichten (Datagramme) mit beliebigen Peers der DHT über einen Socket auszutauschen. Im Detail wird UDP im Verlauf der Vorlesung besprochen.

Um per UDP zu kommunizieren, soll Ihr Programm nun einen UDP-Socket öffnen und auf ankommende Daten warten. Dabei soll der gleiche Port sowohl für TCP, als auch UDP verwendet werden. Beispielsweise soll beim folgenden Aufruf Port 1234 sowohl für den TCP-Socket, als auch für den UPD-Socket verwendet werden:

```
#./build/webserver <IP> <Port>
./build/webserver 127.0.0.1 1234
```

> **!** Öffnen Sie zusätzlich zum TCP-Socket einen UDP-Socket auf dem gleichen Port.

1.2. Hashing

In einer DHT bestimmt der Hash der Daten die Node auf der diese gespeichert sind. Ein Hash ist ein aus einer beliebigen Datenmenge abgeleiteter Wert, der häufig verwendet wird ein Datum zu indentifizieren, oder diese auf Veränderungen zu testen. Typische Hashes haben eine Länge von 256 B, für unsere Zwecke ist ein deutlich kleinerer Namensraum von 16 bit ausreichend.

Da unsere DHT HTTP-Resourcen speichert, verwenden wir hierfür den Hash des Pfads der Resource. Diese beschreiben die Identität einer Resource, unabhängig von ihrem Inhalt. Konkret verwenden wir die ersten zwei Byte des SHA256 Hashs (Dies ist für `/hashhash` beispielsweise 29 380/`0x72c4`[1]). Um diesen zu berechnen bietet es sich an, OpenSSL zu verwenden. Diese Bibliothek ist auf quasi jedem Rechner vorhanden und implementiert diverse Crypto-Primitve, insbesondere auch Hash Funktionen. Anhang C beschreibt die Verwendung in mehr Detail.

> **!** Berechnen Sie den Hash der angefragten Resource bei jeder Anfrage.

Nun wollen wir unsere Implementierung so ändern, dass jede Node basierend auf diesem Hash entscheidet, ob sie für eine Resource verantwortlich ist, oder nicht. Wir erinnern uns, dass Nodes in Chord für alle Keys verantwortlich sind, die zwischen ihrer ID, und der ihres *Vorgängers* liegen. Entsprechend müssen Nodes ihren Nachfolger kennen um dies entscheiden zu können.

[1] `echo -n '/hashhash' | sha256sum | head -c4`

Um die zukünftige Implementierung zu vereinfachen, sollen Nodes auch ihren Vorgänger kennen. Diese „Nachbarschaftsbeschreibung“ soll jeder Node unserer statischen DHT beim Start via Umgebungsvariablen übergeben werden. Umgebungsvariablen können mit der Funktion `getenv` abgefragt werden. Darüber hinaus soll ein dritter Kommandozeilenparameter einer Node ihre ID übergeben. Die folgenden Aufrufe beispielsweise starten eine DHT, die aus zwei Nodes besteht:

```
PRED_ID=49152 PRED_IP=127.0.0.1 PRED_PORT=2002 SUCC_ID=49152 SUCC_IP=127.0.0.1
    SUCC_PORT=2002 ./build/webserver 127.0.0.1 2001 16384
PRED_ID=16384 PRED_IP=127.0.0.1 PRED_PORT=2001 SUCC_ID=16384 SUCC_IP=127.0.0.1
    SUCC_PORT=2001 ./build/webserver 127.0.0.1 2002 49152
```

> **i** Da der Namensraum einer DHT einen Kreis bildet sind die beiden Nodes in einer DHT mit zwei Mitgliedern ihre jeweiligen Vorgänger *und* Nachfolger.

> **!** Entnehmen Sie den beim Aufruf übergebenen Umgebungsvariablen und Kommandozeilenparameter die Informationen über die Node selbst, sowie ihren Vorgänger und Nachfolger in der DHT.

In einer solchen, minimalen DHT bestehend aus zwei Nodes kann kann jede Node bestimmen, welche Node für ein angefragtes Datum verantwortlich ist: Wenn es sie selbst nicht ist, muss es die jeweils andere Node sein. Daher kann eine Node in diesem Fall Anfragen entweder selbst beantworten, oder auf die andere Node verweisen. HTTP sieht für solche Verweise Antworten mit Statuscodes 3XX vor, die das neue Ziel im `Location`-Header enthalten. Wir verwenden in diesem Fall `303: See Other`:

```
# curl -i localhost:2001/hashhash
HTTP/1.1 303 See Other
Location: http://127.0.0.1:2002/hashhash
Content-Length: 0
```

> **!** Beantworten Sie Anfragen wie bisher, wenn die angefragte Node für die Resource verantwortlich ist, andernfalls mit einer Weiterleitung.

> **i** HTTP-Weiterleitugen werden für verschiedene Zwecke eingesetzt, z.B. um sicherzustellen, dass Resourcen via `https` statt `http` angefragt werden. Clients folgen diesen Weiterleitungen typischerweise automatisch. `curl` hingegen tut dies nur, wenn das `-L`/`--location`-Flag gesetzt ist.

1.3. Senden eines Lookups

Im allgemeinen kann eine Node nicht direkt entscheiden, welche Node für ein Datum verantwortlich ist. In diesem Fall sendet sie eine `Lookup`-Anfrage in die DHT (d.h. ihren Nachfolger) um die verantwortliche Node zu erfahren. Diese Anfrage folgt dem oben beschriebenen Format, und enthält die folgenden Werte:

- Message Type: 0 (Lookup)
- Hash ID: Hash der angefragten Resource
- Node ID, IP, und Port: Beschreibung der anfragenden Node

> **!** Senden Sie ein `Lookup` an den Nachfolger, wenn die Node den Ort einer angefragten Resource nicht bestimmen kann.

Parallel zu diser Anfrage erwartet der Client natürlich eine Antwort. Um die Implementierung zu vereinfachen halten wir sie zustandslos: wir vertrösten den Client für den Moment, bis wir eine Antwort erhalten haben. Hierfür können wir eine `503: Service Unavailable`-Antwort mit einem `Retry-After`-Header senden:

```
# curl -i localhost:2002/path-with-unknown-hash
HTTP/1.1 503 Service Unavailable
Retry-After: 1
Content-Length: 0
```

Auf diese Weise müssen wir keine Verbindungen zu Clients offen halten und Antworten aus der DHT diesen zuordnen.

> **!** Beantworten Sie Anfragen mit einer `503`-Antwort und gesetztem `Retry-After`-Header, wenn die Node den Ort einer angefragten Resource nicht bestimmen kann.

1.4. Lookup Reply

Nun implementieren wir die Empfängerseite eines Lookups. Wir erweitern unsere Implementierung, sodass Sie Lookups beantwortet, wenn diese die Antwort kennt. Dafür überprüft die Node, ob sie selbst für den angefragten Pfad verantwortlich ist und sendet eine Antwort an den Anfragesteller, falls dies der Fall ist. Das Reply folgt dem eingangs beschriebenem Nachrichtenformat, mit den folgenden Werten:

- Message Type: 1 (Reply)
- Hash ID: ID des Vorgängers der antwortenden Node
- Node ID, IP, und Port: Beschreibung der verantwortlichen Node

> **!** Senden Sie ein `Reply` an die anfragende Node, wenn ein Lookup empfangen wird und die empfangende Node verantwortlich ist.

1.5. Weiterleiten eines Lookups

Wenn wir ein Lookup empfangen, das wir nicht beantworten können, muss die entsprechende Node „hinter“ uns in der DHT liegen. Entsprechend leiten wir dieses unverändert an unseren Nachfolger weiter.

> **!** Leiten Sie ein empfangenes `Lookup` an ihren Nachfolger weiter, wenn die Anfrage nicht beantwortet werden kann.

1.6. Ein komplettes Lookup

In den vorigen Tests haben Sie haben die einzelnen Aspekte eines Lookups implementiert: Senden, Weiterleiten und Beantworten. Nun muss ihre Implementierung lediglich Antworten verarbeiten, sich also die enthaltenen Informationen merken. Da unsere DHT nicht produktiv eingesetzt können Sie annehmen, dass nicht allzu viele Anfragen parallel gestellt werden. Es genügt also hier, wenn Sie eine kleine Liste von 10 Antworten speichern, und die jeweils älteste verwerfen, wenn eine neue empfangen wird.

! Verarbeiten Sie empfangene Antworten und speichern die relevanten Informationen.

In einer DHT von mindestens drei Nodes sollte Ihre Implementierung nun in der Lage sein, den Ort von Ressourcen zu bestimmen:

```
# curl -i localhost:2002/path-with-unknown-hash
HTTP/1.1 503 Service Unavailable
Retry-After: 1
Content-Length: 0

# curl -i localhost:2002/path-with-unknown-hash
HTTP/1.1 303 See Other
Location: http://127.0.0.1:2017/path-with-unknown-hash
Content-Length: 0

# curl -i localhost:2017/path-with-unknown-hash
HTTP/1.1 404 Not Found
Content-Length: 0
```

Beachten Sie bei der letzten Anfrage, dass `Not Found` sich auf die Existenz der Ressource bezieht. Mit den passenden Flags führt `curl` die wiederholte Anfrage, und das Folgen der Weiterleitung automatisch aus:

```
# curl -iL --retry 1 localhost:2002/path-with-unknown-hash
HTTP/1.1 503 Service Unavailable
Retry-After: 1
Content-Length: 0

Warning: Problem : HTTP error. Will retry in 1 seconds. 1 retries left.
HTTP/1.1 303 See Other
Location: http://127.0.0.1:2001/path-with-unknown-hash
Content-Length: 0

HTTP/1.1 404 Not Found
Content-Length: 0
```

1.7. Statische DHT

Herzlichen Glückwunsch! Ihr Code sollte nun eine statische DHT implementieren. Testen Sie das ganze nun mit mehreren Nodes und verschiedenen `GET`, `PUT`, und `DELETE` Anfragen.

Denken Sie beim Starten der Nodes daran, die Umgebungsvaribalen entsprechend zu setzen! Dafür können Sie die folgenden Anfragen auf einen Pfad ausgeführt:

- `GET`: Erwartet `404: Not Found`
- `PUT`: Erwartet `201: Created`
- `GET`: Erwartet `200: Ok`
- `DELETE`: Erwartet `200: Ok/204: No Content`
- `GET`: Erwartet `404: Not Found`

Dieser Anfragen entsprechen dem letzten Test des ersten Aufgabenzettels. Allerdings wird jede Anfrage an eine andere Node der DHT gestellt. Durch die bisherigen Tests ist sichergestellt, dass die Anfragen trotzdem konsistent beantwortet werden.

Ihrer lokalen. Derartige Unterschiede, beispielsweise in den vorhandenen Bibliotheken, können im Zweifel auch abweichendes Verhalten der Tests zur Folge haben.

C. OpenSSL

Um OpenSSL zu verwenden muss Ihre Implementierung gegen die Bibliothek gelinkt werden. Dies ist im vorgegebenen Code schon der Fall, die relevanten Zeilen in der `CMakeLists.txt` sind:

```
find_package(OpenSSL REQUIRED)
target_link_libraries(webserver PRIVATE ${OPENSSL_LIBRARIES} -lm)
```

> Unter macOS ist OpenSSL nicht standardmäßig installiert. Sie können es via `brew` nachinstallieren. Gegebenenfalls müssen Sie CMake den Ort an den es installiert wurde via `OpenSSL_DIR` übergeben: `cmake -Bbuild -DOpenSSL_DIR=<path>`.

Die Funktionen um SHA Hashes zu berechnen sind im `openssl/sha.h`-Header zu finden und können wie folgt verwendet werden:

```
uint16_t hash(const char* str) {
    uint8_t digest[SHA256_DIGEST_LENGTH];
    SHA256((uint8_t *)str, strlen(str), digest);
    return htons(*((uint16_t *)digest)); // We only use the first two bytes here
}
```