

Praxis 2: Statische DHT

Fachgruppe Telekommunikationsnetze (TKN)

12. Januar 2024

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 Webserver, 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.

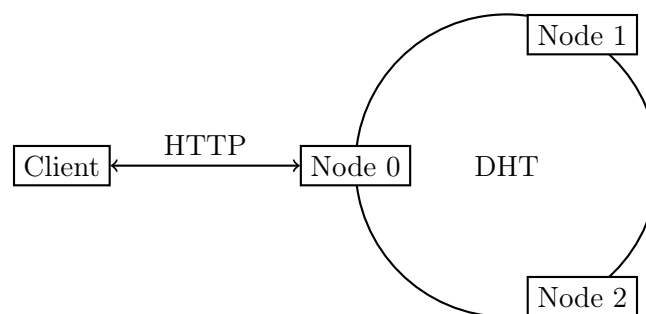


Abbildung 1: Schaubild des Gesamtsystems

Wir beschränken uns in dieser Aufgabenstellung zunächst auf eine 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 jeweilige 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:

0	1	2	3	4	5	6	7
Message Type							
Hash ID							
Node ID							
Node IP							
Node Port							

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 UDP-Socket verwendet werden:

```
1 | #./build/webserver <IP> <Port>
2 | ./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-Ressourcen 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 29380/0x72c4¹). Um diesen zu berechnen, bietet es sich an, OpenSSL zu verwenden. Diese Bibliothek ist auf quasi jedem Rechner vorhanden und implementiert diverse Crypto-Primitive, 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.

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

¹echo -n '/hashhash' | sha256sum | head -c4

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:

```
1 | 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
2 | 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
```

• 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 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:

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

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

• HTTP-Weiterleitungen werden für verschiedene Zwecke eingesetzt, z.B. um sicherzustellen, dass Ressourcen 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 dieser 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:

```
7 # curl -i localhost:2002/path-with-unknown-hash
8 HTTP/1.1 503 Service Unavailable
9 Retry-After: 1
10 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, oder ihr Nachfolger, für den angefragten Pfad verantwortlich ist und sendet eine entsprechende Antwort an den Anfrager, falls dies der Fall ist. Das Reply folgt dem zuvor beschriebenen 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 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 jetzt in der Lage sein, den Ort von Ressourcen zu bestimmen:

```
11 # curl -i localhost:2002/path-with-unknown-hash
12 HTTP/1.1 503 Service Unavailable
13 Retry-After: 1
14 Content-Length: 0
15
16 # curl -i localhost:2002/path-with-unknown-hash
17 HTTP/1.1 303 See Other
18 Location: http://127.0.0.1:2017/path-with-unknown-hash
19 Content-Length: 0
20
21 # curl -i localhost:2017/path-with-unknown-hash
22 HTTP/1.1 404 Not Found
23 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:

```
24 # curl -iL --retry 1 localhost:2002/path-with-unknown-hash
25 HTTP/1.1 503 Service Unavailable
26 Retry-After: 1
27 Content-Length: 0
28
29 Warning: Problem : HTTP error. Will retry in 1 seconds. 1 retries left.
30 HTTP/1.1 303 See Other
31 Location: http://127.0.0.1:2001/path-with-unknown-hash
32 Content-Length: 0
33
34 HTTP/1.1 404 Not Found
35 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 Umgebungsvariablen entsprechend zu setzen! Dafür können Sie die folgenden Anfragen auf einen Pfad ausführen:

- 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

Diese 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.

A. Abgabeformalitäten

Die Aufgaben sollen von Ihnen in Gruppenarbeit mit jeweils *bis zu drei Kursteilnehmern* gelöst werden. *Jedes Gruppenmitglied muss seine Abgabe* einzeln hochladen. Ohne eine eigene Abgabe auf ISIS können Sie keine Punkte erhalten! Fügen Sie Ihrer Abgabe eine Datei `group.txt` hinzu, die Name und Matrikelnummer aller Gruppenmitglieder enthält. Dadurch bleibt die reguläre Gruppenarbeit bei unserem Plagiarismuskcheck unbeachtet.

Ihre Abgaben laden Sie ausschließlich auf ISIS bis zur entsprechenden *Abgabefrist* hoch. Sollten Sie technische Probleme bei der Abgabe haben, informieren Sie uns darüber unverzüglich. Lassen Sie uns dabei auch zur Sicherheit ein Archiv Ihrer Abgabe per Mail zukommen.

Beachten Sie bei der Abgabe, dass die Abgabefrist fix ist und es *keine Ausnahmen für späte Abgaben oder Abgaben via E-Mail* 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. Senden Sie uns in diesem Fall so bald wie möglich das Attest zu.

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

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

Wir empfehlen *dringend*, nach der Abgabe Ihr Archiv einmal selbst herunterzuladen, zu entpacken, und die Tests auszuführen. Dadurch können Sie Fehler, wie leere Abgaben, fehlende Quelldateien, Tippfehler, Inkompatibilitäten, falsche Archivformate, und vieles mehr vermeiden.

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 pytest test # Alle tests ausführen
2 pytest test/test_praxisX.py # Nur die Tests für einen bestimmten Zettel
3 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

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 `webserver` mit dem Dateinamen `webserver` (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 Poolrechnern² 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:

```
1 | ./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. 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:

```
1 | find_package(OpenSSL REQUIRED)  
2 | target_include_directories(webserver PRIVATE ${OPENSSL_INCLUDE_DIRS})  
3 | target_link_libraries(webserver PRIVATE ${OPENSSL_LIBRARIES} -lm)
```

²Die 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.

•
i

Unter macOS ist OpenSSL nicht standardmäßig installiert. Sie können es via **brew** nachinstallieren. Über **brew list openssl** können Sie herausfinden, an welchem Ort die Bibliothek installiert wurde. Diesen Pfad **find_library** als HINTS übergeben, damit CMake auch dort die Bibliothek sucht. Zum Beispiel:

```
4 | find_package(OpenSSL REQUIRED HINTS /usr/local/frank/openssl@3/1.2.3_4/)
```

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

```
1 | uint16_t hash(const char* str) {  
2 |     uint8_t digest[SHA256_DIGEST_LENGTH];  
3 |     SHA256((uint8_t *)str, strlen(str), digest);  
4 |     return htons(*(uint16_t *)digest); // We only use the first two bytes here  
5 | }
```