TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur Professur Rechnernetze

BACHELORARBEIT

PROTOKOLLOPTIMIERUNG IN EINER PROKURAKOMMUNIKATION

Alrik Geselle

Matrikel-Nr.: 3680287

Betreuer: Dipl.-Inf. Tenshi Hara, Dr.-Ing. Thomas Springer

verantwortlicher Hochschullehrer: Alexander Schill

Termin der Abgabe: 22.08.2014

SELBSTSTÄNDIGKEITSERKLÄRUNG

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde bisher weder im Inland noch im Ausland in gleicher oder ähnlicher Form einer anderen Prüfungsbehörde vorgelegt und ist auch noch nicht veröffentlicht worden.

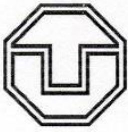
Dresden, 22.08.2014

…………………………

Alrik Geselle

Kurzfassung

Diese Arbeit befasst sich mit der Optimierung des Kommunikationsprinzips im MapBiquitous-Projekt hinsichtlich des Overheads und der Effizienz. MapBiquitous ist eine Indoor-Navigationssoftware, die über HTTP mit verschiedenen Servern kommuniziert. Zunächst werden verschiedene Optimierungsansätze für HTTP untersucht. Danach werden Alternativen wie WebSockets näher betrachtet. Bei der Implementierung wird der Prototyp als eigenständige Anwendung entwickelt, um Unterschiede zwischen HTTP und WebSockets genauer vergleichen zu können. Die Evaluation und die Auswertung der Ergebnisse zeigen, dass WebSockets sehr viele Vorteile gegenüber HTTP bieten, jedoch für die momentane Version von MapBiquitous ungeeignet sind. Ein Umbau zu Gunsten von WebSockets ist aber möglich.



TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur Lehrstuhl Rechnernetze

AUFGABENSTELLUNG FÜR DIE BACHELORARBEIT

THEMA: Protokolloptimierung in einer Prokurakommunikation

Name, Vorname:	Geselle, Alrik	Studiengang:	Medien-Inf. (Bachelor)
Matrikel-Nummer:	3680287	Projekt/Schwerpunkt:	MapBiquitous/Mobile
Erster Gutachter:	Dipl.-Inf. Tenshi Hara	Zweiter Gutachter:	Dr.-Ing. Thomas Springer
Beginn am:	30.05.2014	Einzureichen bis:	22.08.2014

ZIELSTELLUNG

SANE ist eine Crowdsourcing-basierte, verteilte Proxy-Plattform zur Bereitstellung von Crowdsourcing-bezogenen Diensten. Im Rahmen des MapBiquitous-Projektes, einem integrierten, ortsbezogenen Dienstes für den Innen- und Außenbereich, wurde erfolgreich die Machbarkeit nachgewiesen. Das Konzept beruht auf einer verteilten Infrastruktur von mobilen Clients, prokurierenden Zwischenentitäten (den SANEs) und Crowdsourcing-Servern, die beliebige Daten und Informationen kommunizieren. Die prokurierte Kommunikation wurde prototypisch implementiert und weist diversen Protokolloverhead auf.

Ziel der Bachelorarbeit ist die Untersuchung von Kommunikationsprotokollprinzipien und die anschließende Bewertung der in SANE verwendeten Kommunikationsprotokolle. Auf Basis der Bewertung soll dann die SANE-Kommunikation optimiert werden, insbesondere in Bezug auf Overhead und Effizienz. Im Anschluss sollen die relevanten Aspekte der Optimierungsvorschläge prototypisch implementiert und evaluiert werden. Die Kompatibilität zur unterliegenden Basisarchitektur muss *nicht* gewährleistet bleiben, wenn eine solche Kompatibilitätserhaltung der Optimierung im Wege stünde.

SCHWERPUNKTE

- Recherche verwandter Arbeiten zu Protokolldesign/-optimierung
- Bewertung des Ist-Zustandes
- Konzeption von Optimierungsvorschlägen (einer oder mehr)
- Prototypische Umsetzung eines Optimierungsvorschlages
- Erarbeitung einer Evaluationsmethodik
- Evaluation und Bewertung der Ergebnisse

Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill
(verantwortlicher Hochschullehrer)

INHALTSVERZEICHNIS

Inhaltsverzeichnis I

Abbildungsverzeichnis V

Tabellenverzeichnis VII

Abkürzungs- und Symbolverzeichnis IX

1 Einleitung 1

1.1 Motivation 1

1.2 Zielstellung und Vorgehensweise 2

2 Grundlagen und verwandte Arbeiten 2

2.1 HTTP 3

2.1.1 Begriffserklärung 3

2.1.2 Entstehung 3

2.1.3 Aufbau 4

2.1.4 Funktionsweise 6

2.1.5 Statuscode-Definitionen 8

2.1.6 Request-Methoden 10

2.1.7 Idempotent 17

2.1.8 Sicher 17

2.1.9 Übersicht 17

2.2 WebSocket 18

2.2.1 Begriffserklärung 18

2.2.2 Aufbau 18

2.2.3 Funktionsweise 21

3 Bewertung der SANE-Kommunikation 24

4 Konzeptuelle Betrachtungen 28

4.1 Nachrichtenübertragung durch HTTP 28

4.1.1 Übertragung per HTTP-GET 28

4.1.2 Übertragung per HTTP-POST 29
4.1.3 Parameternamen ersetzen 29
4.1.4 Cookies als Zwischenspeicher 32
4.1.5 Nachrichten komprimieren 33
4.1.6 Nachrichten bündeln 36
4.2 Nachrichtenübertragung durch WebSockets 41
4.3 Alternativen zu HTTP 42
4.4 Weitere Vorgehensweise 43
4.4.1 Erwartungen an den Prototyp 44
5 Implementierung 44
5.1 Voraussetzungen 44
5.2 Verwendete Software 44
5.3 Vorgehensweise 45
5.4 Erstellen des Servers 45
5.4.1 WebSocket-Server 45
5.4.2 HTTP-Server 47
5.5 Erstellen des Clients 48
5.5.1 WebSocket-Client 48
5.5.2 HTTP-Client 49
5.6 Probleme 50
6 Evaluation und Auswertung 51
6.1 Verwendete TestKonfiguration 51
6.2 Ergebnisse der Messungen 52
6.2.1 Durchsatz 52
6.2.2 Overhead 53
6.2.3 Übertragungsdauer 54
6.2.4 Latenz 55
6.3 Auswertung 57

7 Zusammenfassung und Ausblick 58

7.1 Zusammenfassung 58

7.2 Ausblick 59

Literaturverzeichnis 61

ABBILDUNGSVERZEICHNIS

Abbildung 1 Aufbau einer HTTP-Nachricht (Vgl. Schröder, et al., 1999 S. 201) 4
Abbildung 2 Grundlegende Funktionsweise von HTTP (Wilde, 1999 S. 60) 7
Abbildung 3 HTTP unter Einbeziehung eines Proxys (Vgl. Wilde, 1999 S. 61) 8
Abbildung 4 HTTP unter Einbeziehung eines Tunnels (Vgl. Wilde, 1999 S. 61)....................... 8
Abbildung 5 Methodenbeispiel für CONNECT 10
Abbildung 6 Methodenbeispiel für DELETE 11
Abbildung 7 Methodenbeispiel für GET 12
Abbildung 8 Methodenbeispiel für HEAD 13
Abbildung 9 Methodenbeispiel für OPTIONS 13
Abbildung 10 Methodenbeispiel für POST 14
Abbildung 11 Methodenbeispiel für PUT 15
Abbildung 12 Verlauf eines TRACE-Requests vom Client über die Zwischenstationen, bis hin zum Server und wieder zurück. (HTTP: Request-Methoden - HTMLWORLD, o. J.) 16
Abbildung 13 Methodenbeispiel für TRACE 16
Abbildung 14 Protokollverlauf eines WebSocket Handshakes (Fette, et al., 2011) 22
Abbildung 15 WebSocket Framing Format (Fette, et al., 2011) 23
Abbildung 16 MapBiquitous Architektur (Hara, 2012) 24
Abbildung 17 Unterschiede der Gebäudeinformationsanfragen des HSZ ohne weitere HTTP-Header 25
Abbildung 18 Aufzeichnung der HTTP-Transaktionen mit Wireshark 26
Abbildung 19 Gebäudedarstellung im MapBiquitous-Client 27
Abbildung 20 Anfrage der benötigten SANE-Server 27
Abbildung 21 Beispielnachricht im POST-Body 29
Abbildung 22 Beispielnachricht im POST-Body mit gekürzten Parameternamen 30
Abbildung 23 Beispielnachricht im POST-Body mit fortlaufender Nummerierung 31
Abbildung 24 Ablaufdiagramm einer komprimierten Nachricht 36
Abbildung 25 Drei-Wege-Handschlag beim Aufbau einer TCP-Verbindung (Sajal, 2011) 37
Abbildung 26 HTTP-POST Methode mit gebündelten Bodys 39
Abbildung 27 Implementierung des WebSocket-Servers 46
Abbildung 28 Aufbau der onMessage-Methode 47
Abbildung 29 Einfacher HTTP-Server, der alle erhaltenen Nachrichten zurück sendet 48
Abbildung 30 WebSocket-Client, herstellen einer Verbindung 49
Abbildung 31 HTTP-Client zum Senden und Empfangen auf Android 50

Abbildung 32 Durchsatz pro Nachrichtenlänge 52
Abbildung 33 Overhead pro Nachricht 53
Abbildung 34 Overheadanteil pro Nachrichtenlänge einer einzelnen Nachricht 54
Abbildung 35 Gesamtdauer der Nachrichtenübertragung pro Nachrichtenblock 55
Abbildung 36 Latenzverhalten zwischen WebSockets und HTTP 56
Abbildung 37 Latenzverhalten einzelner Nachrichten zwischen WebSockets und HTTP 56

TABELLENVERZEICHNIS

Tabelle 1 Methodeneigenschaften 18
Tabelle 2 Overhead beim Laden der Gebäudeinformationen 25
Tabelle 3 Parameternamenkürzung 30
Tabelle 4 Übersetzungstabelle für Methodennamen 32
Tabelle 5 Verwendete Testkonfiguration 51

ABKÜRZUNGS- UND SYMBOLVERZEICHNIS

CPU	Central Processing Unit
HTTP	Hypertext Transfer Protocol
SANE	Server Access Network Entity
JSON	JavaScript Object Notation
MIME	Multi-purpose Internet Mail Extensions
PHP	PHP: Hypertext Preprocessor
TLS	Transport Layer Security
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
RPC	Remote Procedure Call

1 EINLEITUNG

In der heutigen Gesellschaft ist die Unterstützung durch elektronische Geräte nicht mehr wegzudenken. Sie erleichtern uns den Alltag in vielerlei Hinsicht. So helfen sie uns beim Bewältigen komplexer Rechenaufgaben innerhalb kürzester Zeit, sorgen für mehr Sicherheit, unterstützen uns beim Einkauf oder bei der Fortbewegung. Ohne sie könnten wir all diese Dinge nicht in so kurzer Zeit und in so hoher Geschwindigkeit erledigen. Auch die Kommunikation über kurze sowie lange Strecken untereinander spielt eine immer wichtigere Rolle.

Auch bei der Software MapBiquitous spielt Kommunikation und Unterstützung eine wichtige Rolle. MapBiquitous ist ein System für ortsabhängige Dienste basierend auf GPS und Kartendaten zur Navigation. Das Besondere daran ist, dass es sich hier vor allem um eine Indoor-Navigationssoftware handelt. Das heißt, dass sich die Navigation nicht nur auf Außenbereiche beschränkt, wie man es von den heutigen GPS-Empfängern gewohnt ist. Denn GPS funktioniert nur im Außenbereich, da Gebäude den Empfang von Satellitensignalen stören. Das System basiert auf einer dezentralen Infrastruktur von Gebäudeservern, die Informationen über Kartendaten, Positionsbestimmung und Navigation bereitstellen. Damit Gebäudeinformationen unabhängig voneinander aktualisiert werden können, wird eine Crowdsourcing-basierte Struktur eingesetzt. Um die Navigation zwischen dem Benutzer mit seinem mobilen Android-basierten MapBiquitous-Client und den Gebäudeservern zu ermöglichen, wird eine verteilte Proxy-Plattform (SANE), die Crowdsourcing-bezogene Dienste bereitstellt, verwendet.

1.1 MOTIVATION

Wie bei vielen elektronischen Geräten ist eine Kommunikation von Daten erwünscht oder auch notwendig. So benötigt der MapBiquitous-Client Verbindungen zwischen den SANE, Gebäudeserver und weiteren Servern bzw. Diensten. Mit dem ständig steigenden Bedarf nach immer schnellerer und intensiverer Kommunikation von Daten, steigen auch der Aufwand und vor allem die Kosten, diese zu verwalten und zu ermöglichen. Deshalb ist es wichtig, die zu übertragenden Daten möglichst gering zu halten und effizient zu übertragen.

Die MapBiquitous Architektur verwendet derzeit HTTP-basierte Kommunikationsprotokolle. HTTP wurde eigentlich dafür entwickelt, Informationen von Webseiten zu laden. Für den reinen Datenaustausch ist dieses Protokoll nur bedingt geeignet, da mitunter sehr viel

Overhead entsteht. Mehr Overhead bedeutet mehr Datenverkehr und somit mehr Kosten und Stromverbrauch. Oft muss bei der Kommunikation die Verbindung erneut aufgebaut werden, was zu einer Verringerung der Geschwindigkeit führt.

1.2 ZIELSTELLUNG UND VORGEHENSWEISE

Nachdem die Probleme in Abschnitt 1.1 aufgezeigt wurden, sollen nun in der weiteren Arbeit Möglichkeiten untersucht werden, die diese Probleme beseitigen bzw. verringern. Die Kommunikation zwischen Client und SANE bzw. Client und Gebäudeserver soll optimiert werden. Dabei soll der Overhead möglichst reduziert und die Effizienz gesteigert werden. Die vorhandene Struktur muss nicht erhalten bleiben, wenn sie der Optimierung im Wege stünde. Das heißt, dass auch Alternativen zu HTTP in Betracht gezogen werden dürfen.

Um diese Ziele zu erreichen wird im Folgenden die Vorgehensweise erläutert. Zunächst sollen Kommunikationsprotokollprinzipien untersucht werden. Dabei werden im Kapitel 2 zuerst allgemeine Grundlagen zu den zu untersuchenden Aspekten dargelegt. Danach erfolgt in Kapitel 2 die Bewertung der in SANE verwendeten Kommunikationsprotokolle. Im Anschluss werden in Kapitel 4 allgemeine Optimierungsansätze diskutiert und grob ausgewertet. Danach werden die zu implementierenden Optimierungsansätze ausgewählt. Dabei soll anhand der Bewertung der SANE-Protokolle entschieden werden, inwieweit sich die SANE-Kommunikation optimieren lässt. In Kapitel 5 erfolgt die Implementierung der ausgewählten Verfahren, welche dann im darauffolgenden Kapitel 6 evaluiert und ausgewertet wird. Als Letztes werden die Ergebnisse zusammengefasst und mögliche zukünftige Ideen erwähnt.

2 GRUNDLAGEN UND VERWANDTE ARBEITEN

In diesem Kapitel werden die Grundlagen für die weiteren Teile der Arbeit etwas näher erklärt. Einige Begriffe, die aus dem Alltag schon bekannt sein sollten, werden hier nicht weiter vertieft. Die entsprechenden Stellen sind jedoch mit Hinweisen zu weiterführender Literatur markiert. Nachfolgend werden die Themengebiete HTTP inklusive Aufbau, Funktionsweise, Statuscode-Definitionen etc. und WebSockets näher betrachtet. Diese Erläuterungen basieren vor allem auf den Requests for Comments (RFC), welche eine durchnummerierte Folge von Dokumenten darstellen. Sie beschreiben unterschiedliche Gebräuche mit Bezug zum Internet, die von der Internet Engineering Task Force (IETF) herausgegeben werden. Bis

ein fertiges RFC veröffentlicht wird, können einige Jahre vergehen, da Forschung und Entwicklung zu den jeweiligen Themengebieten viel Zeit in Anspruch nehmen. Somit lässt sich ein RFC mit einer wissenschaftlichen Arbeit einer universitären Einrichtung vergleichen.

2.1 HTTP

2.1.1 Begriffserklärung

Das Hypertext Transfer Protocol ist ein Protokoll zur Übertragung von Daten zwischen einem Server und einem Client. (Vgl. Wöhr, 2004 S. 8) Die beiden Kommunikationspartner werden im Abschnitt 2.1.4 näher beschrieben. Das Protokoll bildet die Grundlage zum Laden von Webseiten mit einem Webbrowser im World Wide Web. Da für die Übertragung eine zuverlässige Datenverbindung vorausgesetzt wird, basiert es auf TCP. Durch TCP wird sichergestellt, dass Datenpakete zuverlässig und in richtiger Reihenfolge beim Verbindungspartner ankommen. HTTP ist im ISO/OSI-Referenzschichtenmodell in der Anwendungsschicht einzuordnen, welches der Schicht 5 entspricht. Die meistgenutzte Anwendung, welche dabei zum Einsatz kommt, ist der Webbrowser. HTTP ist ein zustandsloses Protokoll. Das heißt, dass alle Transaktionen zwischen Server und Client unabhängig voneinander behandelt werden. Um dennoch Sitzungsinformationen auszutauschen, können bei jeder Anfrage eine sogenannte Session-ID übertragen werden. Sie enthält eine vom Server generierte Zahlen- und Buchstabenfolge, die dazu genutzt wird, um den Client wiederzuerkennen. HTTP wurde ursprünglich für die Übertragung von Hypertext entwickelt. Mit der Weiterentwicklung des Protokolls ist es mittlerweile auch möglich jede beliebige Art von Daten zu Übertragen. Die Erweiterung der Header-Informationen und Anfragemethoden bilden auch die Grundlage für WebDAV, einem für Dateiübertragung spezialisierten Protokoll.

2.1.2 Entstehung

Tim Berners-Lee und Roy Fieldling vom CERN, dem europäischen Kernforschungszentrum in der Schweiz, entwickelten 1989 das Hypertext Transfer Protocol. (Vgl. li_service, 2011) Zunächst wurde es 1991 unter der heutigen Bezeichnung HTTP/0.9 veröffentlicht. Zu diesem Zeitpunkt wurde lediglich der Befehl GET unterstützt. (Vgl. Reibold, 2001) Schon fünf Jahre später wurde die erste Weiterentwicklung HTTP/1.0 (RFC 1945) präsentiert. Zu den Erweiterungen zum neuen Standard zählten unter anderem die Befehle HEAD und POST sowie Authentifizierung und die MIME-Unterstützung, welche Informationen über den Typ

der Daten lieferten. (Vgl. Berners-Lee, et al., 1996) 1999 wurde die bis heute noch aktuelle Version HTTP/1.1 veröffentlicht.

2.1.3 Aufbau

Im Folgenden wird der Aufbau einer HTTP-Nachricht erklärt, welche in der Abbildung 1 schematisch dargestellt ist. Eine Nachricht besteht in HTTP aus einem Nachrichtenkopf (engl. Message Header), welcher die protokollspezifischen Header-Daten enthält und einem Nachrichtenkörper (engl. Message Body), in dem sich die optional zu übertragenden Nutzdaten befinden. (Vgl. Wöhr, 2004 S. 219) Die Reihenfolge der in der Abbildung dargestellten Elemente, entspricht der zeitlichen Übertragungsreihenfolge.

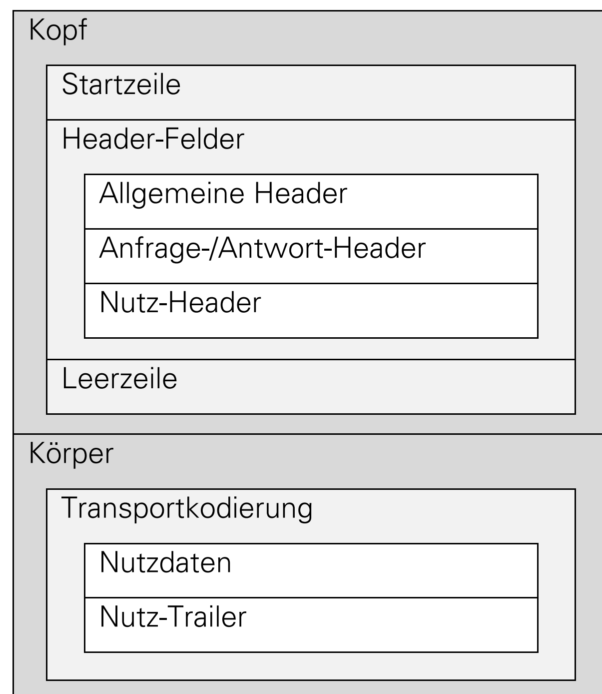


Abbildung 1 Aufbau einer HTTP-Nachricht (Vgl. Schröder, et al., 1999 S. 201)

Der Nachrichtenkopf enthält Kodierungs- und Inhaltstypinformationen über den Nachrichtenkörper, damit dieser vom Empfänger richtig interpretiert werden kann. Diese Informationen werden in kleinere Bereiche unterteilt, welche im Folgenden kurz erläutert werden.

In der Startzeile befinden sich die Anfrage-Methode bzw. Antwort mit Statuscode (genaueres im Abschnitt 2.1.5), die Adresse zur Ressource[1] (engl. Request-URL) und die HTTP-Version. Das URI-Schemata für das Hypertext Transfer Protocol lautet „http:“.

Wie sicherlich schon erkennbar ist, wird die Anfrage-Methode nur bei Anfragen vom Client zum Server genutzt. Wenn der Server auf eine Anfrage antwortet, so ist diese nicht vorhanden. Stattdessen befindet sich der Statuscode mit einer Kurznachricht an dessen Position.

Die Header-Felder werden dazu genutzt, um zusätzliche Angaben zur Nachricht zu übertragen. Es gibt jedoch auch Fälle, in denen keine Header-Felder benötigt werden. (Vgl. Wöhr, 2004 S. 219) Zu den Header-Feldern zählen die allgemeinen Header (engl. General Header), die ihre Informationen eher auf die Verbindung beziehen. Das kann zum Beispiel das Datum der Anfrage sein oder ob eine Verbindung direkt im Anschluss geschlossen werden soll.

Im Anfrage-Header (engl. Request Header) stehen weitere Informationen zur Anfrage für den Server. Deshalb werden diese Header nur in der Hinrichtung benutzt. In der Rückrichtung, also der Antwort auf eine Anfrage, kommen hingegen die Antwort-Header (engl. Response Header) zum Einsatz. Sie enthalten weitere Informationen zur Antwort vom Server. Als Letztes gibt es noch die Nutz-Header (engl. Entity Header). Sie enthalten zusätzliche Informationen zu den Nutzdaten, auch Metadaten genannt. Da Nutzdaten in beide Richtungen übertragen werden können, kommen die Nutz-Header auch in beiden Fällen zum Einsatz.

Jeder Nachrichtenkopf endet mit einer Leerzeile. Sie signalisiert das Ende des Kopfes und besteht nur aus einem Zeilenwechsel. Werden Nutzdaten übertragen, so befindet sich direkt nach dem Nachrichtenkopf der Nachrichtenkörper. Ist im Nachrichtenkopf das Header-Feld „Transfer-Encoding“ gesetzt, so können die Nutzdaten kodiert im Nachrichtenkörper vorliegen. Die Nutzdaten enthalten die angeforderten Inhalte wie zum Beispiel Dokumente, Formulardaten oder auch Fehlerbeschreibungen. Zu guter Letzt kann der Nutz-Trailer zum Beispiel Prüfsummen enthalten, welche genutzt werden, wenn eine Transportkodierung verwendet wird. Die Felder Nutz-Header, Nutzdaten und Nutz-Trailer sind zusammengefasst auch unter dem Begriff Nutzlast (engl. Pay Load) bekannt. (Vgl. Schröder, et al., 1999 S. 200 ff.)

[1] Uniform Ressource Identifier (URI) sowie URL und URN können im RFC 3986 nachgelesen werden. Siehe auch http://tools.ietf.org/html/rfc3986 (aufgerufen am 12.07.2014)

2.1.4 Funktionsweise

Möchte der Client zum Beispiel eine Website mit seinem Browser laden, so sendet der Browser einen HTTP-Request zum Server. Der Server antwortet darauf hin mit einem HTTP-Response. In HTTP sind Request und Response, welche für Anfrage und Antwort stehen, Nachrichten, welche in einer Transaktion zwischen Client und Server übertragen werden. Bevor jedoch eine Transaktion stattfinden kann, muss eine Verbindung zum Server aufgebaut werden. HTTP nutzt TCP-Verbindungen, um Nachrichten sicher zu übertragen. Somit ist HTTP ein einfaches und zuverlässiges Protokoll, welches ebenso verbindungsorientiert arbeitet. Sowohl Client als auch Server nutzen Port 80 zur standardmäßigen Kommunikation. Um eine Transaktion durchzuführen, sind folgende Schritte notwendig:

1. Aufbauen der TCP-Verbindung zum Server durch den Client
2. Senden des HTTP-Request vom Client zum Server
3. Rücksendung des HTTP-Response zum Client nach Verarbeitung durch den Server
4. Abbauen der TCP-Verbindung durch den Sever

Der Client ist ein Programm, welches eine Verbindung zum Server aufbaut, um Anfragen zu versenden. In den meisten Fällen entspricht das Programm einem Web-Browser, der zum Laden und Anzeigen von Webseiten im Internet benutzt wird. Andere Arten können zum Beispiel Schnittstellen zur Übertragung von Daten zum Server, welche keine Webdokumente sind, sein oder auch eine Suchmaschine, die Webseiten durchsucht. Das Gegenstück zum Client ist der Server, der auch ein Programm darstellt. Darunter zählen Programme, die Anfragen von Clients entgegen nehmen, verstehen, verarbeiten und darauf antworten. (Vgl. Wilde, 1999 S. 50 f.)

Nachdem die Verbindung zwischen Client und Server aufgebaut wurde, antwortet der Server nur dann mit einem HTTP-Response, wenn er zuvor eine Anfrage vom Client erhalten hat. In der Abbildung 2 ist eine vereinfachte Darstellung einer HTTP-Transaktion zwischen Client und Server zu sehen. Somit ist klar, dass ein Client sich die Daten vom Server „ziehen“ (engl. pull) muss. Das ist auch eines der Kritikpunkte zu HTTP. Es gibt jedoch eine ganze Reihe von Möglichkeiten, damit der Server dennoch Nachrichten zum Client „schieben“ (engl. push) kann. Ein paar davon sind WebSockets (siehe Abschnitt 2.2) oder Long-Polling[2]. Dieser Nachteil soll mit einer zukünftigen HTTP/2.0 Version behoben werden.

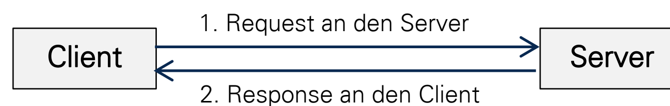


Abbildung 2 Grundlegende Funktionsweise von HTTP (Wilde, 1999 S. 60)

Zwischen Server und Client können sich noch andere Zwischenstationen befinden, die Anfragen und Antworten zum entsprechenden Empfänger weiterleiten. Das können zum Beispiel Tunnel, Proxys oder Gateways sein. Ein Proxy ist auch ein Programm, welches die Rolle eines Servers oder Clients einnehmen kann. Er vermittelt somit Nachrichten zwischen beiden Kommunikationspartnern. Alle Anfragen vom Client werden zum Server und alle Antworten vom Server zum Client weitergeleitet. Dem Proxy ist es aber auch möglich die Anfragen eines Clients zu verarbeiten. Die Antwort auf eine Anfrage könnte auf dem Proxy im Zwischenspeicher (engl. Cache) liegen. Somit werden diese Anfragen nicht zum Server weitergeleitet, sondern direkt durch den Proxy beantwortet. Abbildung 3 verdeutlicht den Verlauf einer Transaktion noch einmal. In dieser ist auch zu sehen, wie der Proxy die Rollen des Clients bzw. des Servers einnimmt. Ein Gateway ist einem Proxy ähnlich, weshalb die Darstellung in Abbildung 3 sowohl für einen Proxy, als auch für einen Gateway gelten soll. Bei einem Proxy wissen Server und Client, dass sie mit einem Proxy kommunizieren. Bei einem Gateway hingegen nicht.

[2] Long-Polling ist ein Verfahren, bei dem der Client eine Anfrage zum Server schickt, aber der Server nicht sofort antwortet. Die Verbindung bleibt solange offen, bis sich auf dem Server ein Zustand ändert. Erst dann antwortet der Sever auf die Anfrage. Nach Empfang der Nachricht beim Client, stellt dieser eine erneute Anfrage an den Server und der Vorgang wiederholt sich von vorn. Siehe auch http://stackoverflow.com/questions/11077857/what-are-long-polling-websockets-server-sent-events-sse-and-comet (aufgerufen am 02.07.2014)

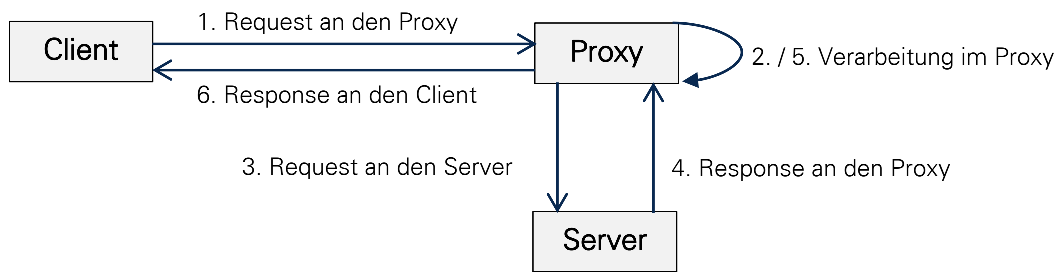


Abbildung 3 HTTP unter Einbeziehung eines Proxys (Vgl. Wilde, 1999 S. 61)

Im Gegensatz zu einem Proxy und Gateway leitet ein Tunnel HTTP-Nachrichten direkt weiter. Ankommende Anfragen oder Antworten von einem Server oder Client werden also nicht bearbeitet. Tunnel besitzen auch keinen Zwischenspeicher, das heißt, dass alle Anfragen eines Clients zum Server weitergeleitet werden. In der Abbildung 4 wird eine solche Transaktion noch einmal verdeutlicht. Man sieht auch die für die Kommunikationspartner anscheinende direkte Verbindung zueinander. Der Tunnel ist für sie sozusagen unsichtbar.

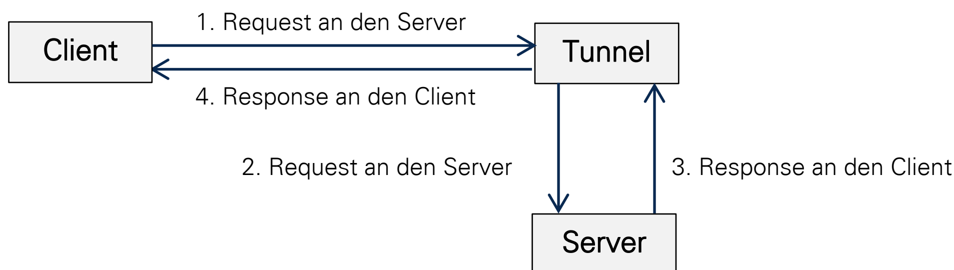


Abbildung 4 HTTP unter Einbeziehung eines Tunnels (Vgl. Wilde, 1999 S. 61)

2.1.5 Statuscode-Definitionen

Während der Kommunikation zwischen Client und Server schickt der Client immer wieder Anfragen (Requests) zum Server. Der Server antwortet auf jede Anfrage hin mit einem sogenannten Statuscode. Statuscodes bestehen aus einer dreistelligen Zahl. Dabei gibt die erste Zahl die Klasse an, wohingegen die letzten beiden Zahlen zur Kategorisierung dienen. Aktuell gibt es fünf Klassen, von denen jede einen eigenen Typ beschreibt. Statuscodes dienen der Auswertung von Fehlern oder zusätzlichen Informationen für den Client, was zum Beispiel der Browser sein kann. Jeder HTTP Client sollte Statuscodes so auswerten können, dass sie der Spezifikation von HTTP entsprechen. Die Codes können dabei in einer eigenen Anwendung beliebig erweitert werden. Manche Statuscodes liefern zu der dreistelligen Zahl an sich auch noch zusätzliche Informationen, welche die entsprechende Situation näher beschreiben. (Vgl. Schröder, et al., 1999 S. 214) Dem Client unbekannte Statuscodes sollten dabei nicht ignoriert, sondern generisch behandelt werden. (Vgl. Tilkov, 2011 S. 223)

2.1.5.1 (1xx) Informational

Der Server sendet diese Klasse von Statuscodes zurück, um den Client darüber zu informieren, dass die Anfrage empfangen wurde und nun bearbeitet wird. Das heißt, dass eine beliebige Anzahl von diesen Nachrichten vor der eigentlichen Antwort vom Server zum Client gesendet werden können. (Vgl. Fielding, et al., 2014) Da diese Art von Nachrichten erst mit HTTP/1.1 eingeführt wurden, dürfen sie allerdings nur dann an Clients versendet werden, wenn sie diese Version auch unterstützen. (Vgl. Tilkov, 2011 S. 223) Der Client darf diese Antworten ignorieren, wenn er nicht damit rechnet. Das bedeutet wiederum, dass die übertragenen Daten wirklich nur zur Information dienen sollen. Es dürfen keine Nutzdaten übertragen werden, stattdessen sind nur die Statuszeile, eventuelle Header-Felder und eine Leerzeile erlaubt.

2.1.5.2 (2xx) Successful

Diese Klasse des Statuscodes wird vom Server zurückgesendet, wenn die Bearbeitung einer Anfrage erfolgreich war. Dabei werden meistens Daten zum Client überragen. Der am häufigsten vorkommende Statuscode 200 (OK) wird zum Beispiel dann zurück gesendet, wenn der Client über seinen Browser eine Webseite laden möchte.

2.1.5.3 (3xx) Redirection

Manchmal ist es notwendig, dass der Client weitere Schritte einleiten muss, damit der Server die angeforderte Ressource zur Verfügung stellen kann. Das ist zum Beispiel dann der Fall, wenn eine angeforderte Ressource nicht mehr über die alte Adresse erreichbar ist. Der Server gibt dann über das Location-Header-Feld die neue Adresse bekannt, unter der sie in Zukunft zu erreichen ist.

2.1.5.4 (4xx) Client Error

Enthält eine Anfrage Fehler oder fehlen Informationen so wird diese Klasse verwendet. Zu jeder dieser Antworten sollte eine genauere Beschreibung mitgeliefert werden, sofern das in der verwendeten Methode erlaubt ist. Jeder, der im Internet surft, wird schon einmal die Fehlermeldung 404 gesehen haben. Diese Meldung bedeutet, dass die angeforderte Ressource vom Server nicht gefunden wurde, da der Client anscheinend einen ungültigen URI verwendet hat. Aus diesem Grunde wird diese Klasse unter Client-seitige Fehler eingeordnet.

2.1.5.5 (5xx) Server Error

Ist auf dem Server ein Fehler aufgetreten, obwohl eine Anfrage vollständig und korrekt ist, dann wir ein Statuscode dieser Klasse zurück gesendet. Auch hier gilt: Wenn es die verwendete Methode erlaubt, dann soll eine genauere Beschreibung mitgeliefert werden.

2.1.6 Request-Methoden

Im Folgenden werden die Anfrage-Request-Methoden vorgestellt, welche bei der Kommunikation zwischen Client und Server-Nachricht verwendet werden. Es existieren zwei Arten von Nachrichten: die Requests, welche der Client als Anfrage zum Server sendet und die Responses, mit welchen der Server auf eine Anfrage antwortet. Zu jeder Erklärung der Methoden wird im folgenden Abschnitt je eine Anfrage und eine Antwort als Beispiel dargestellt.

2.1.6.1 CONNECT

Connect ist eine durch die Spezifikation von HTTP/1.1 reservierte Bezeichnung, welche als Methode für SSL Proxying verwendet wird. Dabei soll ein Proxy dynamisch in den Tunnelmodus wechseln können. Das bedeutet, dass bei einer SSL-Kommunikation zwischen Server und Client einkommende Daten direkt weitergeleitet werden, ohne sie vorher zu bearbeiten. Diese Methode wird auf der Client-Seite verwendet, um dem Proxy mitzuteilen zu welchem Server der Tunnel aufgebaut werden soll. Der Proxy baut dann diese Verbindung auf und leitet alle Daten vom Server zum Client und umgekehrt weiter. (Vgl. Wilde, 1999 S. 444)

Client

```
CONNECT example.com:443 HTTP/1.1
```

Server

```
HTTP/1.1 200 Connection established
Date: ...
```

Abbildung 5 Methodenbeispiel für CONNECT

2.1.6.2 DELETE

Der Client kann durch diese Methode Dateien löschen oder sie in einen nicht zugänglichen Bereich, wie einem Papierkorb, verschieben. Die Antwort des Servers teilt allerdings dem Client nur mit, dass der Server beabsichtigt die Datei zu löschen. Das heißt, selbst wenn der

Server dies meldet, weiß der Client nicht, ob die Datei tatsächlich gelöscht wurde. Wenn die Anfrage erfolgreich durchgeführt wurde, antwortet der Server mit dem Statuscode 200 und dem Ergebnis der Anfrage. Die gleiche Antwort nur ohne Rücksendung des Ergebnisses liefert der Statuscode 204. Der Statuscode 202 wird jedoch zum Client zurückgesendet, wenn die Anfrage akzeptiert, aber noch nicht ausgeführt wurde. Somit ist nicht garantiert, ob die Datei zu einem späteren Zeitpunkt wirklich gelöscht wird. Die Header-Felder „Date", Content-Type" und „Content-Length" enthalten Informationen über das Datum mit der genauen Uhrzeit der Anfrage, sowie den MIME-Typ[3] und die Länge des Bodys in Bytes.

Client

```
DELETE /hello.htm HTTP/1.1
Host: example.com
```

Server

```
HTTP/1.1 200 OK
Date: ...
Content-type: text/html
Content-length: 17

Resource deleted!
```

Abbildung 6 Methodenbeispiel für DELETE

2.1.6.3 GET

Die wohl wichtigste und am meisten verwendete Methode GET wird vom Client genutzt, um vom Server Informationen anzufordern, die dann zum Beispiel im Browser dargestellt werden können. Meistens handelt es sich dabei um Dateninformationen wie einem Dokument, Grafiken oder Audiodateien, welche sich auf dem Server befinden. Es ist jedoch auch möglich, dass der Server die zurückzusendenden Daten erst generiert. Hierbei ist allerdings zu beachten, dass der Server entscheidet, ob es sich bei der angeforderten Ressource um Dateien oder generierte Inhalte handelt. Die Ressource wird anhand des Request-URI ermittelt, in der auch Nutzdaten zum Server übertragen werden können. Die Spezifikation sieht kein Limit für die maximale Länge der Request-URI mit den zu übertragenden Nutzdaten vor. In der Praxis existieren jedoch solche Limits, die unter anderem durch einige ältere Clients wie Browser und Proxys hervorgerufen werden können. Teilweise existieren sogar Beschränkungen auf nur 255 Bytes. (Vgl. Fielding, et al., 1999) Die Methode POST hingegen hat keine Beschränkung der Länge der Nutzdaten und ist somit mehr für die Übertragung von

[3] MIME-Typ (Multipurpose Internet Mail Extensions) wird dazu verwendet, dem Kommunikationspartner mitzuteilen, um welche Art von Daten es sich im Body handelt. Siehe auch http://tools.ietf.org/html/rfc2045 (aufgerufen am 14.07.2014)

Nutzdaten geeignet als GET. Wenn der Server die sogenannten Range-Requests unterstützt, dann kann ein Client auch nur einen Teil einer Ressource anfordern. In der Anfrage wird zum Beispiel das Header-Feld „Range“ mit dem Wert „bytes=0-999“ an den Server geschickt, um nur die ersten 1000 Bytes der Ressource anzufordern. Der Server antwortet daraufhin mit dem Statuscode 206 Partial Content, anstatt dem üblichen 200 OK und den ersten 1000 Bytes der Ressource. Das hat den Vorteil, dass ein Browser-Client bei einer unterbrochenen Verbindung nur einen Teil der Webseite neu anfordern muss, wenn er schon Daten zum Teil empfangen hat.[4] (Fielding, et al., 2014) Ein derartiger Fall einer unterbrochenen Verbindung kann häufig bei mobilen Geräten mit einer Funkverbindung auftreten. (Vgl. Ramsey, 2008) Laut Spezifikation von HTTP soll die GET-Methode nur Daten abrufen und keine weiteren Auswirkungen auf dem Server verursachen. Die Antwort einer GET Anfrage kann im Cache zwischengespeichert werden. (Vgl. Tilkov, 2011 S. 51f.)

Client

```
GET / HTTP/1.1
HOST: example.com
```

Server

```
HTTP/1.1 200 OK
Date: …
Content-Type: text/html

<!DOCTYPE …
```

Abbildung 7 Methodenbeispiel für GET

2.1.6.4 HEAD

Head verhält sich ähnlich wie die Methode GET. Der Unterschied besteht darin, dass der Server nur Metadaten, also ohne Nutzdaten, auf Anfragen zurückliefert. Das ist genau dann sinnvoll, wenn URIs auf Existenz, Gültigkeit oder Änderungen überprüft werden sollen. Des Weiteren wäre es auch möglich, die Größe einer Datei vor dem Download zu überprüfen, um dann entscheiden zu können, ob die Datei wirklich heruntergeladen werden soll. Da sich GET-Anfragen zwischenspeichern lassen, kann so durch die Überprüfung der Prüfsumme (Content-MD5) oder Inhaltslänge (Content-Length) der Cache-Eintrag als veraltet markiert werden, falls sich die angeforderte Ressource im Cache befindet. Zur effizienten Verarbeitung sollten bei HEAD-Anfragen nur die benötigen Metadaten berechnet werden anstatt wie bei einer GET

[4] Weitere Informationen über Range-Requests finden sich in RFC 2616 sowie RFC 7233. Mit dem kürzlich erschienen RFC 7233 wurde die Funktionalität erweitert.

Anfrage zu verfahren und am Ende die Nutzdaten zu verwerfen. (Vgl. Wilde, 1999 S. 76f.; Vgl. Tilkov, 2011 S. 53f.)

Client

```
HEAD / HTTP/1.1
Host: example.com
```

Server

```
HTTP/1.1 200 OK
Content-Type: text/html
Date: …
Expires: …
Last-Modified: Fri, 09 Aug 2013 23:54:35 GMT
Content-Length: 1270
```

Abbildung 8 Methodenbeispiel für HEAD

2.1.6.5 OPTIONS

Diese Methode kann der Client einsetzen, um Informationen über die verfügbaren Kommunikationsoptionen einer Ressource zu erhalten, welche bei einem Request oder Response unterstützt werden. Somit gelangt der Client an Informationen, ohne dass die Ressource vorher übertragen oder verarbeitet werden muss. Entscheidend bei der Antwort des Servers ist, ob im Request-URI das Zeichen „*“ steht oder speziell eine Ressource. Im Falle des Zeichens übermittelt der Server dem Client die allgemein unterstützten Kommunikationsoptionen, welche auf dem Server implementiert wurden. (Vgl. Wilde, 1999 S. 77) Wenn speziell die Ressource im Request-URI steht, dann übermittelt der Server nur alle Kommunikationsoptionen, die genau für die zutreffende Ressource gelten. Die Antwort des Servers mit dem Statuscode 200 (OK) auf solch eine Anfrage, sollte alle, auch optionale Features als Header-Felder enthalten. (Vgl. Fielding, et al., 1999)

Client

```
OPTIONS * HTTP/1.1
Host: example.com
```

Server

```
HTTP/1.1 200 OK
Allow: OPTIONS, GET, HEAD, POST
Content-Length: 0
Date: …
```

Abbildung 9 Methodenbeispiel für OPTIONS

2.1.6.6 POST

POST wird verwendet um Daten vom Client zum Server zu senden. Die Datenmenge kann dabei unbegrenzt groß sein. Im Gegensatz zu PUT, welches nur zum Erstellen neuer Ressourcen gedacht ist, können auf dem Server durch POST neue Ressourcen angelegt oder modifiziert werden. Jedoch entscheidet wieder der Server, welche Aktion durch das in dem Request angegebenen Request-URI ausgelöst wird. Die POST-Methode wird am häufigsten bei der Übertragung von Formulardaten verwendet. Dennoch kann diese Methode auch dafür eingesetzt werden, um bestimmte Vorgänge auf dem Server auszulösen. Das kann allerdings schnell dazu führen, dass POST als Allzweckmittel für beliebige Operationen eingesetzt wird, wodurch die Vorteile der anderen Methoden verschenkt werden. Nach einer erfolgreichen Übertragung sollte der Server wieder mit Statuscodes antworten. So sollte der Server mit dem Statuscode 200 (OK) antworten, wenn Daten wieder zurück zum Client gesendet werden, welche die Antwort beschreiben. Im Gegensatz dazu wird 204 (No Content) übertragen, falls die Antwort keine weiteren Beschreibungen erfordert. Wurde eine neue Ressource auf dem Server angelegt, so wird mit dem Statuscode 201 (Created) geantwortet.

Client

```
POST / HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 21

method=test&say=hello
```

Server

```
HTTP/1.1 200 OK
Date: …
Content-Type: text/html

<!DOCTYPE …
```

Abbildung 10 Methodenbeispiel für POST

2.1.6.7 PUT

Wie bereits erwähnt ist PUT dafür zuständig, dass neue Ressourcen auf dem Server angelegt werden können. Das Request-URI gibt dabei die Ressource auf dem Server an. Ist sie bereits auf dem Server vorhanden, dann sollten die bei der Anfrage übermittelten Daten als modifizierte Version der Ressource betrachtet werden. Wenn sie nicht vorhanden ist, wird eine neue erstellt und der Server muss danach mit dem Statuscode 201 (Created) antworten. Wenn eine vorhandene Ressource modifiziert wurde, dann antwortet der Server entweder

mit 200 (OK) oder 204 (No Content) um eine erfolgreiche Bearbeitung der Anfrage zu signalisieren. Im Falle einer nicht erfolgreichen Operation sollte der Server mit einer entsprechenden Fehlermeldung antworten. Sollte die Anfrage bereits in einem Cache zwischengespeichert sein und das Request-URI zeigt auf einen oder mehreren zwischengespeicherten Einträge, so müssen diese als veraltet markiert werden. Die Antwort vom Server auf diese Anfragen ist nicht zwischenspeicherbar. (Vgl. Fielding, et al., 1999)

Client

```
PUT /hello.htm HTTP/1.1
Host: example.com
Content-type: text/html
Content-Length: 181

<html>
<body>
<h1>Hello World!</h1>
</body>
</html>
```

Server

```
HTTP/1.1 201 Created
Date: ...
Content-type: text/html
Content-length: 17

Resource created!
```

Abbildung 11 Methodenbeispiel für PUT

2.1.6.8 TRACE

Die Methode TRACE lässt sich zur Diagnose von Anfragen nutzen. Der Server sendet eine Kopie der Nachricht unverändert an den Client zurück. So lässt sich überprüfen, wie die Anfrage letztendlich aussieht, wenn sie beim Server ankommt. In der Antwort vom Server findet sich im Header-Feld „Via“ eine Liste mit den vom Request durchlaufenen Server-Stationen. Sowohl die Zwischenstation der Hin- als auch der Rückrichtung werden vermerkt. (Vgl. HTTP Message Methods - Part 2 of Chapter 3 from HTTP: The Definitive Guide (3/4) | WebReference, 2003) So lässt es sich einfacher zurückverfolgen, ob die Nachricht verändert wurde, falls sie einen oder mehrere Proxys durchläuft, welche als Zwischenspeicher agieren können. Jeder Proxy, der durchlaufen wird, muss dabei einen Vermerk zu den Header-Daten hinzuzufügen. Der Client kann über das Header-Feld Max-Forwards bestimmen, wie viele Proxys maximal durchlaufen werden sollen. Jede Zwischenstation dekrementiert den Wert

von Max-Forwards um 1. Wenn der Wert 0 erreicht ist, wird die Anfrage im aktuellen Zustand zurück zum Client gesendet. Falls die Anfrage schon den Zielserver erreicht hat ohne, dass der Wert von Max-Forwards 0 erreicht hat, dann wird er dennoch um 1 dekrementiert und danach zurückgesendet. Auf diese Weise kann ein Client die Verbindung zum Zielserver Schritt für Schritt testen. Bei einer TRACE Anfrage dürfen keine Nutzdaten übertragen werden. Antworten auf diese Methode dürfen auch nicht zwischengespeichert werden.

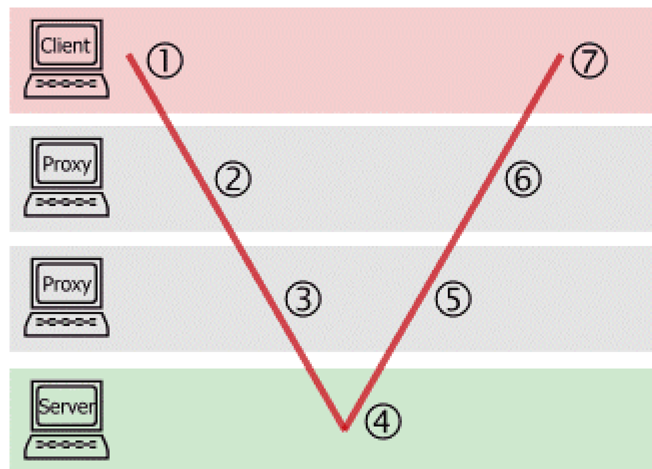


Abbildung 12 Verlauf eines TRACE-Requests vom Client über die Zwischenstationen, bis hin zum Server und wieder zurück. (HTTP: Request-Methoden - HTMLWORLD, o. J.)

Client

```
TRACE / HTTP/1.1
Host: example.com
```

Server

```
HTTP/1.1 200 OK
Date: ...
Content-Type: message/http
Content-Length: 33
Via: 1.1 proxy.example.com

TRACE / HTTP/1.1
Host: example.com
Via: 1.1 proxy.example.com
```

Abbildung 13 Methodenbeispiel für TRACE

2.1.7 Idempotent

Idempotenz ist eigentlich ein mathematischer Begriff, welcher aber auch in der Informatik vorkommt. Er bedeutet, dass die mehrfache Hintereinanderausführung einer Funktion auf sich selbst wieder das gleiche Ergebnis liefert, wie bei einer einmaligen Ausführung. Für HTTP bedeutet dies, dass eine vom Client gesendete idempotente Request-Methode immer das gleiche Ergebnis liefert, also wenn immer die gleichen Seiteneffekte[5] verursacht werden. Dabei bezeichnet ein Zustand zum Beispiel eine Auflistung von registrierten Benutzern einer Website, welche mit einer Datenbankabfrage verknüpft ist. Die Auflistung und Abfrage der Datenbank hat dabei keinen Einfluss auf den zukünftigen Verlauf bei wiederholten Anfragen[6].

2.1.8 Sicher

Eine Methode gilt als sicher, wenn sie die angeforderte Ressource nicht verändert und somit keine Seiteneffekte verursacht. Das heißt, eine sichere Methode wie GET sollte zum Beispiel Daten aus einer Datenbank anzeigen, aber keine Einträge darin verändern. Dabei gibt es jedoch eine Besonderheit zu beachten: Auch wenn eine Methode sicher ist und keine Seiteneffekte verursacht, ist es immer noch erlaubt Dateien auf dem Server oder die Ressource so zu verändern, dass die Darstellung einer Ressource dadurch nicht verändert wird. Ein Fall für solch eine Ausnahme wäre zum Beispiel das Inkrementieren eines Besucherzählers oder Aufzeichnungen von Log-Einträgen[7].

2.1.9 Übersicht

In dieser Übersicht sind noch einmal alle angesprochenen HTTP-Methoden aufgelistet. Des Weiteren befindet sich ein „X“ in der entsprechenden Zeile und Spalte, wenn eine Methode bestimmte Eigenschaften aufweist. Die hier angesprochenen Methoden sind die Gängigsten, die sich nach den Vorgaben des RFC 7231 richten. Sowohl Server als auch Client können zusätzliche Methoden unterstützen, auf welche jedoch hier nicht näher eingegangen wird.

[5] Der Begriff Seiteneffekt bezeichnet in der theoretischen Informatik die Veränderung eines Zustandes. Eigentlich ist Seiteneffekt eine falsche wörtliche Übersetzung des englischen „side effect“. Richtiger wäre die Bezeichnung Nebenwirkung.

[6] Es gibt Ausnahmen, sodass das Ergebnis sich unterscheidet, wenn mehrere Anfragen von Parametern abhängig sind. Generauere Informationen finden sich im RFC 7231

[7] Log-Einträge können Einträge in Textform sein, welche zum Beispiel bestimmte Aktionen oder Fehlermeldungen dokumentieren. Somit dienen sie zur einfacheren Fehlerauswertung.

Methode	sicher	idempotent	cache-fähig
CONNECT			
DELETE		X	
GET	X	X	X
HEAD	X	X	X
OPTIONS	X	X	
POST			
PUT		X	
TRACE	X	X	

Tabelle 1 Methodeneigenschaften

2.2 WEBSOCKET

2.2.1 Begriffserklärung

Das WebSocket-Protokoll ist ein Netzwerkprotokoll welches wie HTTP auf TCP arbeitet. Es erlaubt eine bidirektionale Kommunikation zwischen WebSocket-Client und WebSocket-Server. Das heißt, dass sowohl Client als auch Server das WebSocket-Protokoll unterstützen müssen.

Die größten Vorteile gegenüber HTTP sind, dass der HTTP-Header-Overhead bei WebSockets entfällt und dass eine Bidirektionale und nahezu Echtzeitkommunikation möglich sind. Das minimiert zum einen den entstehenden Traffic und zum anderen sinkt damit die Latenz, weil keine neuen HTTP-Anfragen vom Client zum Server gesendet werden müssen. (Vgl. Weßendorf, 2011)

2.2.2 Aufbau

Um möglichst einfache Integrierbarkeit in eine bestehende Struktur zu ermöglichen, wurde der Verbindungsaufbau von WebSockets durch valide HTTP-Requests konzipiert. Die Verbindung wird also, wie bei HTTP-Anfragen, durch einen Nachrichtenkopf (Header) hergestellt. Da diese Anfragen nur bei Verbindungsaufbau benutzt werden, werden sie bei WebSockets Handshake genannt. Ein Handshake ist der normalen HTTP-Anfrage sehr ähnlich. Es besteht aus der HTTP-Anfragemethode „GET“ sowie dem Pfad zur Ressource, der Adresse des Zielservers (Host), dem Protokoll mit Protokollversion und Header-Feldern. Ein Nachrichtenkörper (Body) wird nicht benötigt. Es wird standardmäßig der Port 80 verwendet.

Er kann aber auch manuell durch die Angabe einer anderen Zahl verändert werden. Die Anfrage des Clients und die Antwort des Servers unterscheiden sich lediglich in der Startzeile und in den Header-Feldern. Der Client fordert mit dem Header-Feld „Upgrade: websocket“ den Wechsel auf das WebSocket Protokoll. Dieses Header-Feld tritt allerdings nur in Kombination mit einem weiteren Header-Feld, dem Feld „Connection: Upgrade“, auf. (Vgl. Fette, et al., 2011)

Die folgenden Header-Felder dienen der zusätzlichen Sicherheit und genaueren Spezifikation der Verbindung.

2.2.2.1 Origin

Das Feld Origin soll die Ursprungsadresse der Client-Anfrage enthalten. Der Server kann dieses Feld auswerten, um zu entscheiden, ob er Anfragen von dieser Adresse zulässt oder verweigert. Werden sie verweigert, so erhält der Client eine Antwort mit dem Statuscode 403 Forbidden. Die Angabe des Feldes „Origin“ ist zwingend erforderlich. Dieses Header-Feld dient also dem Schutz vor unerlaubten Anfragen. Allerdings lässt diese Idee einige Fragen zur Sicherheit offen, da sich zum Beispiel der eingetragene Wert leicht verändern lässt. (Vgl. Fette, et al., 2011)

2.2.2.2 Sec-WebSocket-Key

Dieses Feld enthält einen vom Client zufällig generierten 16 Byte Schlüssel, der im Base64-Format[8] kodiert ist. Der Server nutzt diesen um zu prüfen, ob er eine valide WebSocket-Anfrage erhalten hat. Somit wird sichergestellt, dass der Server nur Anfragen von Clients erhält, welche auch WebSockets unterstützen. Das Feld Sec-WebSocket-Key ist beim Verbindungsaufbau zwingend erforderlich. (Vgl. Fette, et al., 2011)

2.2.2.3 Sec-WebSocket-Version

Das Sec-WebSocket-Version Header-Feld wird vom Client zum Server gesendet, um die verwendete Protokollversion von WebSocket anzugeben. Das ist notwendig, damit der Server die Anfrage richtig interpretiert. Auch für dieses Header-Feld ist die Angabe in der Anfrage zwingend erforderlich. Wird die angeforderte Version nicht vom Server unterstützt, dann sendet der Server eine Fehlermeldung zurück an den Client. In der Fehlermeldung sind alle

[8] Base64 ist ein Verfahren um beliebige Daten in eine Zeichenfolge nach dem ASCII-Standard (American Standard Code for Information Interchange) zu kodieren. Der Basis-Zeichensatz besteht nur aus 64 Zeichen. Deshalb wird dieses Verfahren Base64 genannt. Weitere Informationen finden sich in RFC 4648 zum Beispiel unter: http://tools.ietf.org/html/rfc4648 (aufgerufen am 12.07.2014)

Versionen aufgelistet, die vom Server unterstützt werden. Daraus folgt, dass dieses Header-Feld nur einmal pro Anfrage enthalten sein darf. Im Falle einer Fehlermeldung sind jedoch mehrere Angaben erlaubt, da es dem gleichen entspricht, wenn mehrere Versionsnummern in einem Header-Feld stehen würden. Die aktuelle Spezifikation RFC 6455 gibt einen Wert von „13“ vor. (Vgl. Fette, et al., 2011)

2.2.2.4 Sec-WebSocket-Protocol

Wenn der Client eines oder mehrere Unterprotokolle unterstützt, so kann er es dem Server über das Header-Feld „Sec-WebSocket-Protocol“ mitteilen. Unterprotokolle sind Protokolle, die auch WebSockets aufbauen. Der Server wählt dann nur eins von den übermittelten Protokollen aus und schickt dieses in der Antwort wieder zurück. (Vgl. Ullrich, 2012) Somit ist klar, dass dieses Feld mehrfach in der Anfrage vorkommen darf, in der Antwort jedoch maximal einmal. Durch das Zurücksenden an den Client wird die client-seitige Protokollwahl bestätigt. Die Angabe ist im Header optional, also nicht zwingend notwendig. (Vgl. Fette, et al., 2011)

2.2.2.5 Sec-WebSocket-Extension

Unterstützt ein Client Erweiterungen des Protokolls, so kann er das dem Server über das Header-Feld „Sec-WebSocket-Extension“ mitteilen. Genauso wie beim Header-Feld „Sec-WebSocket-Protocol“, darf der Client mehrere Angaben zu unterstützten Erweiterungen machen. Der Server hingegen, darf nur mit einer Auswahl antworten. Wie bereits erwähnt, gilt die Antwort als Bestätigung. Dieses Header-Feld kann sehr hilfreich sein, wenn zum Beispiel die Verbindung komprimiert[9] oder mehrere Nachrichten zusammengefasst (Multi-Plexing) werden sollen.[10]

2.2.2.6 Sec-WebSocket-Accept

Dieses Header-Feld wird nur vom Server in der Antwort benutzt. Es enthält eine Art Bestätigung an den Client. Der Wert setzt sich auf dem zuvor erhaltenen Header-Feld „Sec-WebSocket-key“ und dem Global Unique Identifier (GUID) zusammen. Der Global Unique Identifier ist eine Implementierung des Universal Unique Identifier. Er garantiert Einzigartigkeit über Raum und Zeit, um zum Beispiel Protokolle eindeutig zu identifizieren. Dies geschieht mit Hilfe einer 128 Bit-langen Hexadezimalzahl, die durch ein spezielles Verfahren generiert

[9] Siehe auch https://www.igvita.com/2013/11/27/configuring-and-optimizing-websocket-compression/ (aufgerufen am 12.07.2014)
[10] Siehe auch http://tools.ietf.org/html/draft-ietf-hybi-websocket-multiplexing-11 (aufgerufen am 12.07.2014)

wird.[11] Für WebSockets lautet der aktuelle GUID: „258EAFA5-E914-47DA-95CA-C5AB0DC85B11“. Nachdem nun das Header-Feld „Sec-WebSocket-key“ und der GUID konkateniert wurden, wird das Hashverfahren SHA-1[12] darauf angewandt. Zum Schluss wird der daraus entstandene Hashwert Base64 kodiert und mit in der Antwort zurück zum Client gesendet. Der Client kann diesen Wert nun auswerten, um sicherzustellen, ob der Server die Anfrage auch wirklich verstanden hat. Folgt aus der Auswertung ein negatives Ergebnis, so wird der Verbindungsaufbau unterbrochen und es werden keine Nutzdaten gesendet.

Zusätzlich können, wie auch bei HTTP, Client und Server weitere Header-Felder zur Kommunikation verwenden. (Vgl. Fette, et al., 2011)

2.2.3 Funktionsweise

Im Gegensatz zu HTTP, bei dem ein Client zuvor eine Anfrage zum Server schicken muss, damit dieser antworten kann, reicht es bei WebSockets aus, wenn der Client eine Verbindung zum Server öffnet. Somit kann auch der Server die offene Verbindung nutzen, um dem Client Nachrichten zu senden. Der Server kann allerdings nicht ohne weiteres von sich aus eine Verbindung zum Client eröffnen. Damit das dennoch möglich ist, muss wieder ein Verfahren wie Long-Polling (bereits in Abschnitt 2.1.4 kurz näher erklärt) eingesetzt werden, bei dem ein Client zuvor eine Anfrage an den Server geschickt hat und noch auf eine Antwort wartet.

Bevor es zum Informationsaustausch kommt, wird die Verbindung mit einem Handshake eröffnet. Das heißt, der Client teilt zuvor dem Server mit, dass er eine WebSocket Verbindung eröffnen möchte. Dabei werden zusätzliche Informationen wie Protokollversion oder auf WebSocket aufbauende Protokolle dem Server übermittelt. Der Server antwortet nach Empfang eines Handshakes mit einer Bestätigung sowie einer Mitteilung, ob zum Beispiel angefragte auf WebSocket aufbauende Protokolle vom Server unterstützt werden.

Im Folgenden wird der Verlauf eines Handshakes dargestellt. Die Anfrage beginnt mit einem normalen HTTP-GET-Request und zusätzlichen Header-Feldern. Der Server Antwortet daraufhin mit einer Bestätigung für den Wechsel auf das WebSocket-Protokoll durch den Statuscode 101 Switching Protocols. Da der Server nur das Chat-Protokoll unterstützt, teilt er dies dem Client mit. Am Ende haben sich beide Partner auf das Chat-Protokoll geeinigt, welches auf WebSockets aufbaut.

---

[11] Siehe auch RFC 4122, zum Beispiel unter: http://tools.ietf.org/html/rfc4122 (aufgerufen am 12.07.2014)

[12] Siehe auch http://tools.ietf.org/html/rfc3174 (aufgerufen am 12.07.2014)

Client

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

Server

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

Abbildung 14 Protokollverlauf eines WebSocket Handshakes (Fette, et al., 2011)

WebSockets unterstützen unverschlüsselte und verschlüsselte Kommunikation. Das URI-Schemata lautet „ws:“ für unverschlüsselte und „wss:“ für verschlüsselte Verbindungen. Die Verbindung wird über HTTP beziehungsweise HTTPS aufgebaut und wird nach erfolgreichem Handshake auf das WebSocket-Protokoll gewechselt. Als Verschlüsselungsart kommt, wie bei HTTPS[13], TLS zum Einsatz. (Vgl. Zimmermann, 2012) Es gibt noch ein paar weitere Sicherheitsvorkehrungen für WebSockets, welche hier nicht näher beschrieben werden.[14]

Nachdem die Verbindung durch einen Handshake aufgebaut wurde, können Daten in einer Folge von sogenannten Rahmen (engl. Frames) bidirektional ausgetauscht werden. Größere Nachrichten werden auf mehrere Frames aufgeteilt. Das hat zum einen den Vorteil, dass sich so Nachrichten bündeln lassen und zum anderen, dass Nachrichten nicht vorher zwischengespeichert werden müssen. (Vgl. Ullrich, 2012) Jede Nachricht, die ein Client zum Server sendet, muss vorher über das „MASK“-Bit maskiert werden.[15] Empfängt der Server eine unmaskierte Nachricht, so sendet er ein sogenanntes „close Frame“ und leitet damit die Schließung der Verbindung ein. Genauso darf ein Server keine Nachrichten maskieren, denn erhält ein Client eine maskierte Nachricht, so schließt er die Verbindung von seiner Seite aus.

[13] Hypertext Transfer Protocol Secure ist ein HTTP Protokoll, das mit dem TLS Protokoll, einem Netzwerkprotokoll zur sicheren Übertragung von Daten, verschlüsselt wurde.
[14] Weitere Informationen zum genauen Ablauf, Aufbau und Sicherheitsmerkmale finden sich in RFC 6455
[15] Die Maskierung von Nachrichten ist unter anderem zur zusätzlichen Sicherheit notwendig. Siehe auch http://tools.ietf.org/html/rfc6455 (aufgerufen am 12.07.2014)

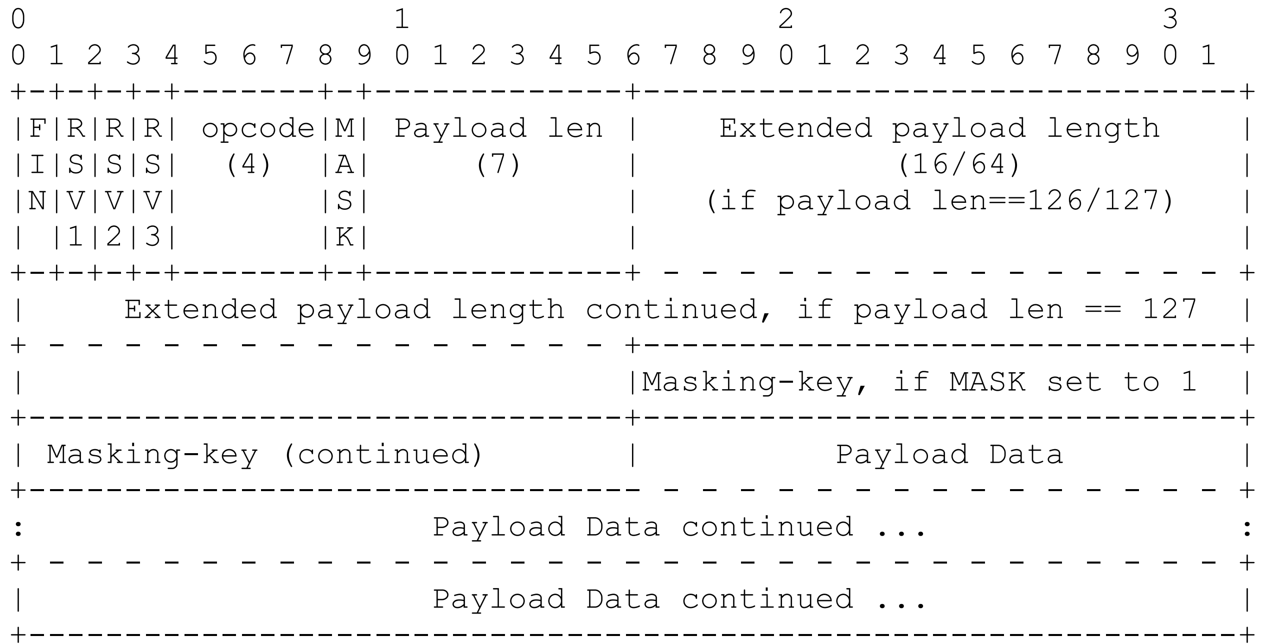


Abbildung 15 WebSocket Framing Format (Fette, et al., 2011)

Da der Overhead in WebSockets möglichst gering gehalten werden soll, sind die Header der Frames möglichst dicht gepackt. Abbildung 15 zeigt einen solchen Frame mit seinen Header-Bits und der Position der Nutzdaten (engl. payload). Jeder Frame hat eine Header-Größe von 2 – 14 Bytes. Der genaue Wert ist von der Nachrichtenlänge und der Maskierung abhängig.

Das erste Bit „FIN“ gibt an, ob der aktuelle Frame der letzte einer Nachricht ist. Die nächsten drei Bits werden für Protokollerweiterungen genutzt. Die darauffolgenden 4 Bits enthalten den OP-Code, der zur Interpretation der Nutzdaten genutzt wird. Das achte Bit ist das Maskierungsbit. Ist es gesetzt, so muss ein 32 Bit langer Maskierungsschlüssel (Masking-key) angegeben werden. Ab dem neunten Bit wird die Länge der Nachricht angegeben. Die dafür verwendeten Bits variieren jedoch, abhängig vom Umfang der Nutzdaten, in ihrer Länge. Mit dem Maskierungsschlüssel endet der Header. Alle nachfolgenden Bits werden für die Nutzdaten verwendet. (Vgl. Fette, et al., 2011)

3 BEWERTUNG DER SANE-KOMMUNIKATION

Die MapBiquitous-App für Android stürzte bei jedem Programmstart ab. Erst nach einigen Fehlerbehandlungen, war es zumindest möglich das Startverhalten zu beobachten. Aus diesem Grund beziehen sich die Bewertungen vor allem auf den Quellcode des MapBiquitous-Projektes und auf die Initialisierungsphase beim Starten der Anwendung. Der MapBiquitous-Client kommuniziert mit einer Reihe verschiedener Server, um eine Navigation vor allem im Innenbereich zu ermöglichen. In Abbildung 16 sind die Verbindungen zwischen den einzelnen Komponenten zu sehen, die hier betrachtet werden und optimiert werden sollen. Alle Schreibzugriffe und Lesezugriffe mit bestimmten Zugriffsrechten werden über den SANE-Server geleitet. Dieser verarbeitet alle einkommenden Anfragen und regelt die Zugriffe auf die Gebäudeserver. Die Kommunikation zwischen Client und SANE erfolgt hauptsächlich über die HTTP-POST-Methode. Auch zwischen SANE und Gebäudeserver wird ausschließlich über die POST-Methode kommuniziert. Werden nur Leserechte benötigt, so sendet der Client GET-Anfragen an die Gebäudeserver. Sowohl Client, als auch SANE und Gebäudeserver können auch Anfragen an den Directory-Server senden, der unter anderem vom Client genutzt wird, um einen Gebäudeserver zu ermitteln.

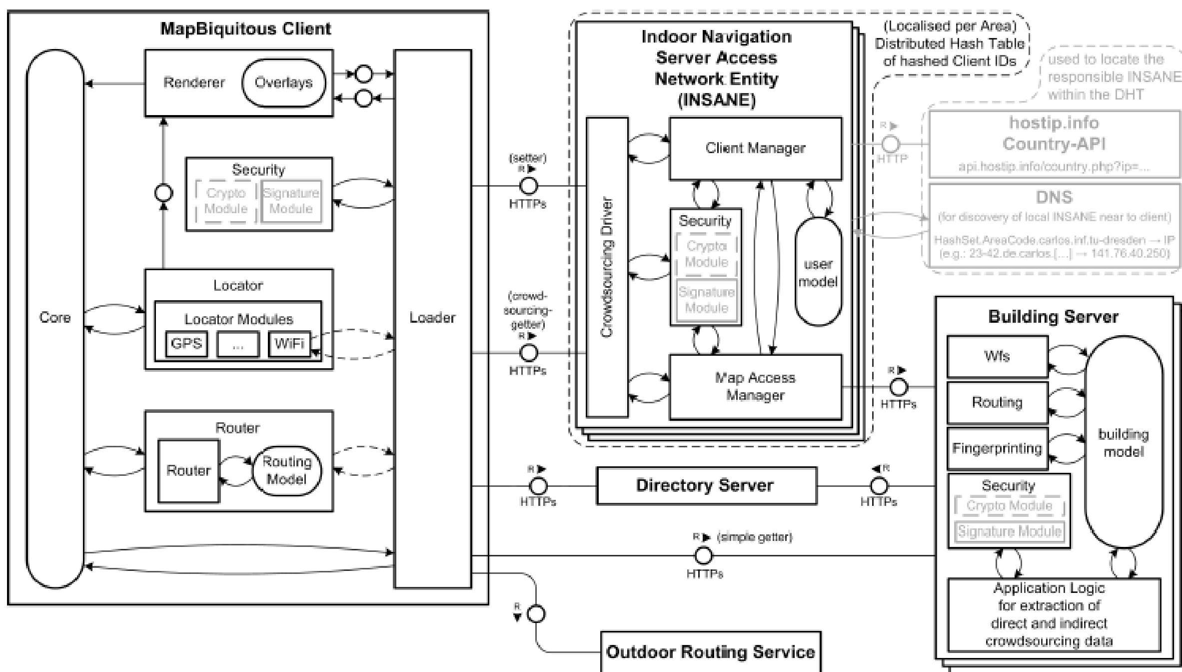


Abbildung 16 MapBiquitous Architektur (Hara, 2012)

Beim Starten der MapBiquitous-App wird eine Reihe von Gettern aufgerufen, die an den Gebäudeserver (Building Server) gesendet werden, um die verfügbaren Gebäude zu laden.

Diese Anfragen sind im Durchschnitt 128 Zeichen lang. Alle Anfragen erzeugen sehr viel Overhead, der vor allem durch die HTTP-Header entsteht. Da die Antwort vom Server die Gebäudedaten enthält, ist sie entsprechend groß. Mit Wireshark, einem Programm zur Analyse von Netzwerk-Kommunikationsverbindungen, wurde der Verlauf der HTTP-GET-Anfragen und Antworten aufgezeichnet. In der Abbildung 18 ist der Inhalt einer HTTP-Antwort (rot umrandet) zu sehen. In ihr befinden sich Gebäudeinformationen über die Fakultät Informatik. Zusätzliche Header-Daten zeigen, dass die Nutzdaten vom Server mit „GZIP" komprimiert wurden und dadurch die Gesamtlänge der Nutzdaten von 17549 Bytes auf 2685 Bytes reduziert werden konnte. Für jedes Gebäude werden drei Anfragen versendet, von denen sich lediglich der Pfad zur Ressource unterscheidet. Abbildung 17 zeigt die gekürztten Anfragen für das HSZ der TU Dresden.

Client

```
GET /geoserver/tud_hsz/...&REQUEST=GetCapabilities
GET /geoserver/tud_hsz/...&REQUEST=GetFeature&TYPENAME=tud_hsz:TUD_HSZ_G
GET /geoserver/tud_hsz/...&REQUEST=DescribeFeatureType&TYPENAME=tud_hsz:TUD_HSZ_G
```

Abbildung 17 Unterschiede der Gebäudeinformationsanfragen des HSZ ohne weitere HTTP-Header

Als nächstes wird eine Übersicht über die zusammengefassten Anfragen der einzelnen Gebäude gezeigt. Als Nutzdaten wird hier bei den GET-Anfragen nur der reine Pfad zur Ressource und bei den Antworten nur der Body angesehen. Der Overhead lässt sich einfach berechnen:

$$Overhead = 1 - \frac{Nutzdaten\ in\ Bytes}{Gesamtdaten\ in\ Bytes}$$

Anhand der HTTP-Nachrichten der geladenen Gebäude konnte folgender Overhead in Prozent ermittelt werden:

Gebäudekürzel	Overhead Anfrage	Overhead Antwort	Overhead gesamt
BAR	66,6%	13,0%	79,6%
GOE	66,6%	14,2%	80,9%
HSZ	66,6%	14,1%	80,8%
HUE	66,6%	14,9%	81,6%
INF	66,6%	14,6%	81,3%
M13	66,6%	13,2%	79,9%
TIL	66,6%	14,1%	80,8%
TST	66,6%	18,1%	84,8%

Tabelle 2 Overhead beim Laden der Gebäudeinformationen

Anhand der extrahierten HTTP-Informationen ist mitunter zu erkennen, dass der Server „mapbiquitous.inf.tu-dresden.de“ nicht richtig mit Anfragen umgehen kann, die das Header-Feld „Connection: Keep-Alive“ enthalten. In jeder Antwort von diesem Server fehlt eine Reaktion auf das Header-Feld „Connection“. Anhand der Werte von „Src Port“ bzw. „Dst Port“ (grün umrandet) in Abbildung 18 lässt sich aus Wireshark ablesen, dass immer wieder eine neue Verbindung zum gleichen Server aufgebaut wird.

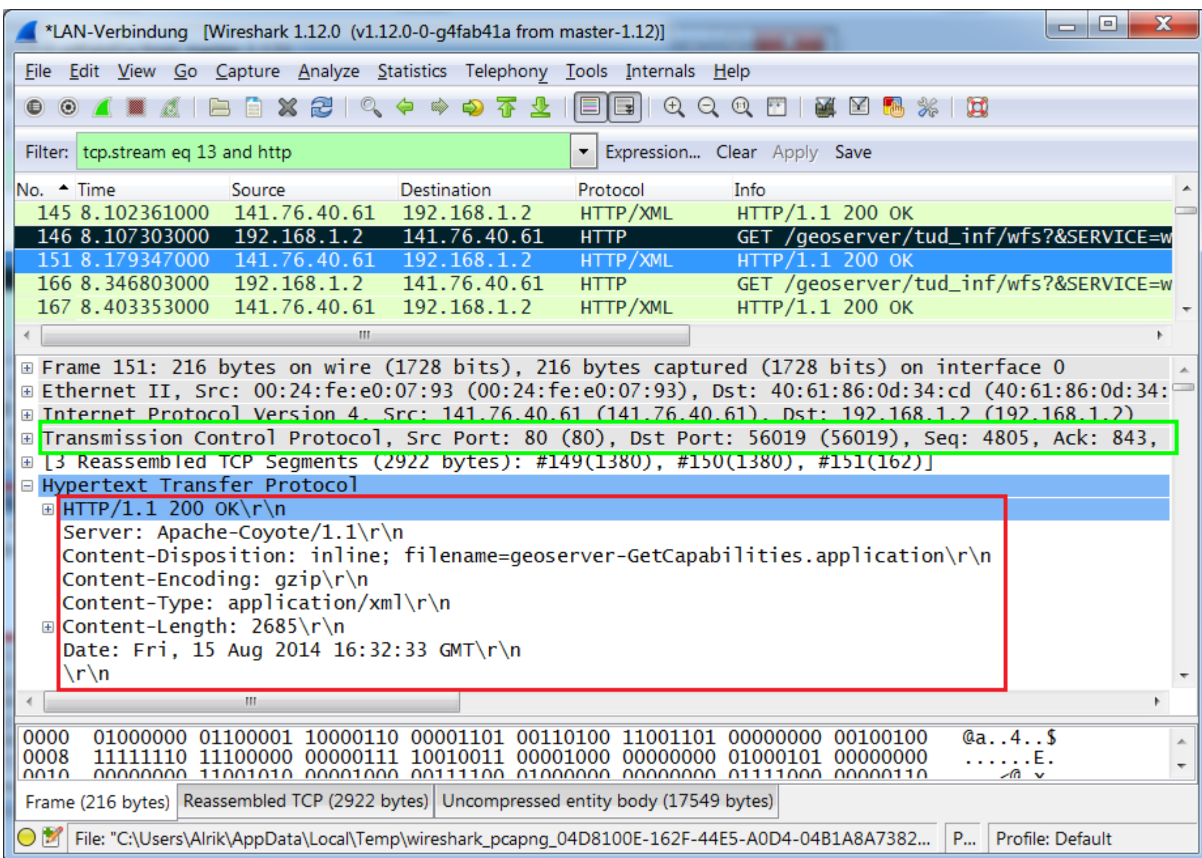


Abbildung 18 Aufzeichnung der HTTP-Transaktionen mit Wireshark

Die Nutzdaten aus der Antwort aus Abbildung 18 enthalten unter anderem Angaben über die sichtbaren Gebäudeumrisse im XML-Format, welche dann von der Client-Software in geometrische Formen umgewandelt werden. Die geometrischen Formen sind in Abbildung 19 zu sehen. Sie zeigt einen Ausschnitt aus dem MapBiquitous-Client mit dem Gebäude der Fakultät Informatik in der rechten unteren Ecke.

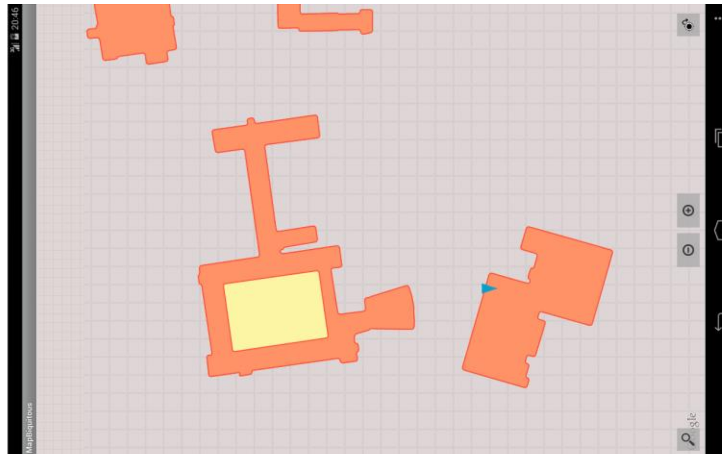


Abbildung 19 Gebäudedarstellung im MapBiquitous-Client

Beim Starten des MapBiquitous-Clients werden die für die Crowdsourcing-Teilnahme zuständigen SANE-Server ermittelt. Die zugehörige POST-Anfrage sieht folgendermaßen aus:

Client

```
POST /DS/ HTTP/1.1
Content-Length: 9
Content-Type: application/x-www-form-urlencoded
Host: carlos.inf.tu-dresden.de:80
Connection: Keep-Alive
User-Agent: Apache-HttpClient/UNAVAILABLE (java 1.4)

region=de
```

Abbildung 20 Anfrage der benötigten SANE-Server

Von den insgesamt 212 Bytes Gesamtlänge für die Anfrage sind nur 9 Bytes wirkliche Nutzdaten. Das entspricht gerade einmal 4%. Diese Anfrage wird jedoch nur einmal zu Beginn gesendet, deshalb ist dieser Umstand vernachlässigbar.

Die meisten HTTP-Nachrichten werden zu „google.com" gesendet. Dabei handelt es sich höchstwahrscheinlich um Anfragen an den Kartendienst von Google „Google Maps". Sie werden genutzt, um Satellitenbilder vom Außenbereich zu laden und sie im MapBiquitous-Client anzuzeigen. Allerdings funktioniert auch das nicht während des Testvorganges. Wie in Abbildung 19 zu sehen ist, werden nur die Gebäude ohne Satellitenbilder geladen. Da keinerlei Zugriff auf das Google-Server-Verhalten möglich ist, lassen sich diese Verbindungen kaum optimieren.

In der Testphase entstanden kaum weitere Daten, weil bei jedem Vorgang die Anwendung abstürzte oder gar nichts passierte. Weder die Navigation noch die Verwaltung der Benutzerdaten ließen sich ohne Absturz der Anwendung testen.

4 KONZEPTUELLE BETRACHTUNGEN

In diesem Kapitel werden verschiedene Kommunikationsprotokollprinzipien dargelegt und hinsichtlich ihrer Effizienz und Overheads untersucht. Die vorgestellten Verfahren sollen dann dazu dienen, die SANE-Kommunikation im MapBiquitous-Projekt zu optimieren. Da die vorhandene Kommunikation weitestgehend einer normalen HTTP-Kommunikation entspricht, werden die folgenden Optimierungsvorschläge an eher allgemeinen Beispielen betrachtet. Die zu übertragende Datenmenge ist bei der SANE-Kommunikation sehr unterschiedlich, da sie von den aufgerufenen Methoden abhängig ist. So ist zum Beispiel die Datenmenge der Antwort vom Server beim Laden von Gebäudedaten viel größer als die der Anfrage.

Zunächst werden Optimierungsideen innerhalb von HTTP betrachtet. Danach werden WebSockets und andere Alternativen diskutiert. Zum Schluss wird in Abschnitt 4.4 die weitere Vorgehensweise bei der Implementierung der Optimierungsvorschläge erläutert.

4.1 NACHRICHTENÜBERTRAGUNG DURCH HTTP

Zunächst werden Verfahren betrachtet, die durch HTTP bereits gegeben sind. Es werden verschiedene Möglichkeiten aufgezeigt, welche die zu übertragende Datenmenge reduzieren und sie möglichst effizient verarbeiten. Der allererste Schritt besteht darin, nur die nötigsten Daten wie Header-Felder zu übertragen.

Alle betrachteten Prinzipien basieren auf der noch aktuellen HTTP-Version 1.1. Die wohl einfachste Kommunikation zwischen Server und Client, erlaubt die GET-Methode des Hypertext-Transfer-Protocols.

4.1.1 Übertragung per HTTP-GET

Wie bereits in den Grundlagen im Abschnitt 2.1.6.3 erwähnt, kann die Methode GET in der Länge des URL beschränkt sein. Das Problem existiert aber nur, wenn der Client ein Browser oder Proxy ist. Dadurch, dass kein Body existiert, ist diese Methode theoretisch die kürzere im Vergleich zu POST. Durch ihre Beschränkung und dem fehlenden Body bietet sie allerdings wenig Spielraum für Verbesserungen. Auch lassen sich mit GET nur alphanumerische Zeichen

übertragen. Nicht-alphanumerische Zeichen können nur übertragen werden, wenn sie zuvor zum Beispiel mit dem Base64-Verfahren kodiert wurden. Durch die Kodierung steigt jedoch die Gesamtlänge der zu übertragenden Zeichenfolge.

4.1.2 Übertragung per HTTP-POST

Die POST-Methode erlaubt es grundsätzlich, zusätzlich zu übermittelnde Daten in den Nachrichtenkörper (Body) zu verschieben. Der Vorteil besteht darin, dass keine Probleme bezüglich der Länge entstehen können. Außerdem unterstützt POST theoretisch unbegrenzte Längen von Nutzdaten. Zusätzlich lassen sich auch binäre Nutzdaten übertragen, was bei GET nicht ohne weiteres möglich ist. Durch die binäre Nutzdatenübertragung ergeben sich weitere Möglichkeiten, um die Inhaltsmenge zu reduzieren. Soll, wie im Falle von einigen Methodenaufrufen in der SANE-Kommunikation ein Passwort übertragen werden, so sollte auch POST anstatt GET bevorzugt werden. Denn wenn der Client ein Browser ist, dann würde mit der GET-Methode das Passwort in der Adressleiste erscheinen. Wenn jedoch die Netzwerk-Kommunikationsdaten untersucht werden, dann ist ein Passwort auch mit der POST-Methode sichtbar. Abhilfe schafft da nur eine verschlüsselte Verbindung über HTTPS, bei der alle Anfragen und Antworten vollständig verschlüsselt sind.

Aufgrund der überwiegenden Vorteile von POST gegenüber GET, beziehen sich die folgenden Verbesserungsansätze hauptsächlich auf diese Methode.

4.1.3 Parameternamen ersetzen

Der einfachste Weg, Daten bei der Übertragung zu minimieren, ist es, vorhandene Parameternamen zu kürzen. In der folgenden Annahme sollen drei Parameter mit jeweiligen Werten übertragen werden. Der Body in einer HTTP-POST-Nachricht könnte dann folgendermaßen aussehen:

Client-Anfrage

```
POST / HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 39

method=abcd&position=efgh&material=ijkl
```

Abbildung 21 Beispielnachricht im POST-Body

Aus diesem Beispiel lassen sich einige Verbesserungsansätze ableiten.

4.1.3.1 Parameternamen generell kürzen

Anhand dieses Beispiels aus Abbildung 21 lassen sich die Parameternamen leicht kürzen. So könnte aus „method“, einfach „m“ werden. Der wesentliche Nachteil besteht darin, dass es leicht zu Dopplungen kommen kann. Wenn „material“ auch zu „m“ gekürzt wird, dann würde es den Parameternamen „m“ zweimal geben und eine Unterscheidung ist nicht mehr so einfach möglich. Diese einfache Maßnahme hat auch Nachteile für den Entwickler solcher Anwendungen. Die Fehlersuche wird erschwert und die Programmierung schwieriger, da sich Abkürzungen eingeprägt oder aufgeschrieben werden müssen. Der größte Vorteil ergibt sich aus der kürzeren Gesamtlänge. Aus den ursprünglich 39 Zeichen könnten dann 21 werden, wenn Namen entsprechend nach Tabelle 3 ersetzt werden.

Parametername	Abkürzung
method	m
position	p
material	m2

Tabelle 3 Parameternamenkürzung

Die entsprechende POST-Anfrage könnte dann so aussehen:

Client-Anfrage

```
POST / HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 21

m=abcd&p=efgh&m2=ijkl
```

Abbildung 22 Beispielnachricht im POST-Body mit gekürzten Parameternamen

Das ergibt ist eine Einsparung von rund 46%, die sich wie folgt berechnen lässt:

$$Einsparung = 1 - \frac{length(gekürtzt)}{length(original)} = 1 - \frac{21}{39} \approx 46\%$$

4.1.3.2 Fortlaufende Nummerierung

Bei der Kommunikation zwischen Client und SANE beginnt so gut wie jede Anfrage mit dem Parameternamen „method“. Dadurch lassen sich die nachfolgenden Parameternamen beliebig benennen, solange sie immer an der gleichen Position stehen. Der erste Parametername „method“ ruft die entsprechende Methode auf, die dann den weiteren Programmablauf steuert. Angenommen jede Methode für sich nutzt immer die gleichen

Parameter, dann lässt sich jeder Parametername zum Beispiel durch ein „x“ konkateniert mit einer fortlaufenden Nummer ersetzen. Das Beispiel aus Abbildung 21 wird in verbesserter Form in Abbildung 23 dargestellt.

Client-Anfrage

```
POST / HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 23

x0=abcd&x1=efgh&x2=ijkl
```

Abbildung 23 Beispielnachricht im POST-Body mit fortlaufender Nummerierung

Daraus ergibt sich eine Verbesserung hinsichtlich der Länge von rund 41%.

$$Einsparung = 1 - \frac{length(gekürtzt)}{length(original)} = 1 - \frac{23}{39} \approx 41\%$$

4.1.3.3 Kürzung statischer Parameterwerte

Wie in Abschnitt 4.1.3.1 bereits angesprochen, beginnt praktisch jede Anfrage zwischen Client und SANE mit dem gleichen Parameternamen „method“. Der Wert steht für die entsprechende Methode, die auf dem Server aufgerufen werden soll. Genau diese Werte sind fest und können sich immer wiederholen. Ein Methodenname lässt sich zum Beispiel durch eine dreistellige Zahl ersetzen, welche für genau einen Methodennamen steht. Somit ist gewährleistet, dass sich bis zu 1000 Methodennamen durch eine dreistellige Zahl ersetzen lassen.

Damit nicht jede Methode im Programmcode durch eine Zahlenfolge ersetzt werden muss, kann dazu eine Übersetzungstabelle verwendet werden. In der Tabelle stehen die originalen Methodennamen und die dazugehörige Zahlenfolge. Die Zahlenfolge lässt sich mit einer ID gleichsetzen. Die Verwendung dieses Verbesserungsvorschlages erfordert, dass eine Übersetzungstabelle sowohl auf dem Client als auch auf dem Server vorhanden sein muss. Des Weiteren müssen sie vollkommen identisch zueinander sein. In Tabelle 4 ist eine beispielhafte Übersetzungstabelle dargestellt.

ID	Methode
000	changePassword
001	correctWLANFingerprintingPosition
002	getDHTArea
003	getWFS
004	getWLANFingerprintingPosition

Tabelle 4 Übersetzungstabelle für Methodennamen

Möchte der Client eine Anfrage an den Server schicken, so muss er die entsprechende ID für die angeforderte Methode aus der Tabelle heraussuchen. Der Methodenname wird nun in der Anfrage durch die ID ersetzt und danach zum Server gesendet. Der Server nutzt die Tabelle in umgekehrter Reihenfolge wie der Client, um die ID wieder durch den originalen Namen zu ersetzen. Danach kann der Server die gesamte Nachricht wie gewohnt verarbeiten.

Die Einsparung der Länge ist sehr variabel, da sie von der ursprünglichen Länge des Originalnamens abhängig ist. Im Falle der Beispieltabelle wird eine Verbesserung von rund 50% bis 91% erreicht. Diese Werte sehen auf den ersten Blick hoch aus, doch es handelt sich hierbei nur um einen relativen Vergleich zwischen der Länge des ersetzten Methodennamens und der Länge des Originalnamens der Methode.

Berechnung der Verbesserung bei kurzem Methodennamen:

$$Einsparung_{kurz} = 1 - \frac{length(ID)}{length(Methode)} = 1 - \frac{length(003)}{length(\text{getWFS})} = 1 - \frac{3}{6} \approx 50\%$$

Berechnung der Verbesserung bei längerem Methodennamen:

$$Einsparung_{lang} = 1 - \frac{length(001)}{length(\text{correctWLANFingerprintingPosition})} = 1 - \frac{3}{33} \approx 91\%$$

4.1.4 Cookies als Zwischenspeicher

Cookies sind Textinformationen die beim Client gespeichert werden. Sie können durch den Server oder clientseitig durch Scripts generiert werden. Dadurch lassen sich zum Beispiel Sitzungsdaten wie eine Session-ID zwischenspeichern. So kann ein Server, der mehrere Benutzer mit Passwort verwaltet, einen Benutzer wiedererkennen. Anstatt dass der Benutzer für jede Anfrage den Benutzernamen und Passwort mitsendet, sendet er nur die Session-ID. Wenn der Benutzername und der Hash-Wert vom Passwort kürzer sind als die Session-ID,

dann lässt sich wieder etwas Traffic sparen. Das Einsparpotential ist abhängig vom Verhältnis zwischen Session-ID und Benutzername-/Passwort-Hash-Kombination. (Vgl. Barth, 2011)

Da auch der Inhalt von Cookies bei jeder Anfrage an den Server mit gesendet wird, macht es kaum Sinn HTTP-Header-Daten in Cookies auszulagern. Im Gegenteil, durch die Cookie-Header entsteht sogar zusätzlicher Overhead.

4.1.5 Nachrichten komprimieren

Je nachdem, wie viel Zeichen ein Methodenaufruf besitzt, sollte eine Komprimierung in Betracht gezogen werden. Durch Komprimierung lässt sich der gesamte Aufruf verkleinern. Allerdings ist die Kompressionsrate von der Häufigkeit gleicher Zeichen und der Gesamtlänge abhängig. Da die Server-/Clientarchitektur auf Java beziehungsweise PHP setzt, wäre es am besten, ein geeignetes Verfahren zu wählen, welches sowohl von Java als auch von PHP bereits unterstützt wird. Zudem soll es möglichst effizient in Hinsicht auf Schnelligkeit und Kompressionsrate sein. Zur Auswahl stehen ZIP und GZIP. Beide Verfahren basierten auf dem Deflate-Algorithmus[16], einem Verfahren zur verlustlosen Kompression. (Vgl. Deutsch, 1996) Mit GZIP lassen sich Dateien oder auch Datensätze komprimieren, wohingegen ZIP mehrere Dateien zu einem Archiv komprimiert. Da es sich bei der Anfrage an den Server nur um einen Methodenaufruf mit Parametern handelt, eignet sich GZIP am besten. (Vgl. Yuan, 2013) GZIP komprimiert nicht so stark im Vergleich zu anderen Kompressionsverfahren wie BZIP2 oder LZMA, aber es ist deutlich schneller. (Vgl. Odzangba, 2009)

Zu den Vorteilen von GZIP gibt es allerdings auch Nachteile. Zum einen bedeutet die Komprimierung eine zusätzliche Last für die CPU. Zum anderen kann eine zu kurze Zeichenkette nach der Komprimierung länger sein als vorher. Ein weiterer Nachteil entsteht zum Beispiel durch die Kompression eines Text-Strings[17]. GZIP gibt grundsätzlich eine binäre Zeichenkette aus. Diese lässt sich aber nicht ohne weiteres mit der HTTP-GET-Methode übertragen, da nur Zeichen im ASCII-Format unterstützt werden. Um die komprimierten Daten dennoch zu übertragen, müssen sie vorher in das ASCII-Format umgewandelt werden. Das wird durch das zusätzliche Anwenden des Base64-Verfahrens erreicht. Durch die Umwandlung in das ASCII-Format wird allerdings die Gesamtgröße des Text-Strings noch größer. Eine Alternative zur GET-Methode ist die POST-Methode, denn sie unterstützt binäre

[16] Siehe auch http://tools.ietf.org/html/rfc1951 (aufgerufen am 14.07.2014)

[17] Ein String ist eine Zeichenkette variabler Länge. Sie kann aus einer Kombination von Zahlen, Buchstaben oder Sonderzeichen bestehen. Der Zeichensatz ist vorgegeben. Siehe auch http://www.info-wsf.de/index.php/Datentyp_String (aufgerufen am 14.07.2014)

Daten im Body. Das führt dazu, dass GZIP mit HTTP-POST kombiniert werden sollten, um bestmögliche Einsparungen zu erreichen.

Der Server kann eine Nachricht mit komprimiertem Inhalt nicht ohne weiteres lesen und weiterverarbeiten. Aus diesem Grund muss dem Server mitgeteilt werden, dass es sich um eine komprimierte Nachricht handelt. HTTP bietet dafür mehrere Möglichkeiten:

4.1.5.1 Zusätzlicher Parameter

Die Möglichkeit des zusätzlichen Parameters setzt eine feste Zeichenfolge voraus, nämlich die binäre Zeichenfolge der komprimierten Nachricht. Sie könnte zum Beispiel „gz=“ lauten. Nach dem Gleichheitszeichen folgt direkt die komprimierte Nachricht. Das hat aber den Nachteil, dass die gesendete Nachricht um drei Zeichen länger wird. Auf der Serverseite müssen dann allerdings bei jeder POST-Anfrage die ersten drei Zeichen geprüft werden, ob sie dem Muster „gz=“ entsprechen. Ist das der Fall, dann müssen die ersten drei Zeichen von der restlichen Nachricht entfernt werden, damit sie wieder dekomprimiert werden kann. Andersfalls handelt es sich scheinbar um eine normale Anfrage, die nicht mit GZIP komprimiert wurde. Sie kann also wie gewohnt weiter verarbeitet werden.

4.1.5.2 Header-Feld „Content-Encoding“

Die nächste Möglichkeit besteht darin, das Header-Feld „Conent-Encoding“ zu nutzen. Eigentlich ist es nach RFC 2616 nur für Responses vom Server gedacht. Der Client missbraucht das Feld für seine Anfrage und gibt darin zum Beispiel den Wert „gzip“ an. Der Server muss wieder bei jeder empfangenen POST-Anfrage überprüfen, ob das Header-Feld mit dem entsprechenden Wert gesetzt wurde. Ist das der Fall, dann handelt es sich um eine mit GZIP komprimierte Nachricht und sie kann direkt dekomprimiert werden. Andernfalls wird, wie schon im Abschnitt 4.1.5.1 beschrieben, fortgefahren.

4.1.5.3 Header-Feld „Content-type“

Die sauberste Lösung wird durch das Feld „Content-type“ gegeben. Mit diesem Feld wird der MIME-Typ der Nachricht angegeben. Er gibt an, um welche Art von Daten es sich bei der Nachricht handelt. Der Wert „application/gzip“[18], welcher in RFC 6713 beschrieben wird, eignet sich dafür am besten. Wie schon bei den anderen beiden möglichen Lösungen, muss der Server wieder jede Anfrage auswerten. Ist das Feld mit dem entsprechenden Wert

[18] Siehe auch RFC 6713 http://tools.ietf.org/html/rfc6713 (aufgerufen am 17.07.2014)

gesetzt, so handelt es sich wieder um eine Komprimierte Nachricht. Andernfalls wird die Nachricht normal weiterverarbeitet.

Der letzte der drei Vorschläge scheint der sauberste und somit beste zu sein. Durch die Angabe des „Content-type“ Header-Feldes wird die Gesamtlänge der Nachricht nicht unnötig verlängert. Außerdem fallen ein zusätzlicher String-Vergleich und ein Abschneiden von Zeichen weg. Es werden so auch keine Vorgaben des RFC-Standards verletzt, wie es bei dem zweiten Vorschlag der Fall wäre. Nach der Dekomprimierung liegt der ursprüngliche Nachrichteninhalt vor. Die Verarbeitung der Nachricht kann dann wie gewohnt ablaufen.

Da nun die Fälle für Anfragen vom Client an den Server behandelt wurden, bleiben noch die Antworten des Servers. Dabei gibt es zwei Varianten. Die erste ist die gleiche, wie sie schon bei der Client-Anfrage genutzt wird. Somit ergibt sich sogar der Vorteil, dass der Server selbst entscheiden kann, ob er eine Nachricht komprimieren wird oder nicht. Die meisten Antworten werden sehr kurz sein, weshalb sich eine Komprimierung eher nachteilig auswirkt.

Die zweite Variante ist die in HTTP bereits integrierte Kompression. Dabei muss der Client bei seiner Anfrage das Header-Feld „Accept-Encoding: gzip, deflate“ mitsenden. Die beiden Werte „gzip“ und „deflate“ stehen für zwei Kompressionsverfahren, die vom Client unterstützt werden. Der Server wertet diese aus und entscheidet sich für ein Verfahren. Angenommen, der Server entscheidet sich für „gzip“. Dafür fügt er das Header-Feld „Content-Encoding: gzip„ zur Antwort hinzu. Bevor der Nachrichteninhalt der angeforderten Ressource zur Antwort hinzugefügt wird, komprimiert der Server die zugehörigen Daten mit dem gewählten Komprimierungsverfahren. Der Client erkennt anhand des Header-Feldes, ob die Nachricht komprimiert wurde und kann so entsprechend weiter verfahren. Bevor jedoch diese Variante benutzt werden kann, muss der Server entsprechend vorkonfiguriert sein. (Vgl. Azad, 2007) Des Weiteren werden alle vorgegebenen Nutzdaten komprimiert. Bei Nachrichten mit kurzem Inhalt kann das zum Nachteil werden, wie es bereits in der ersten Variante beschrieben wurde. Der allgemeine Ablauf bei komprimierten Nachrichten soll in Abbildung 24 noch einmal verdeutlicht werden. Beide Kommunikationspartner können vor dem Versenden eine Nachricht komprimieren, wenn es gewünscht ist. Wird eine komprimierte Nachricht empfangen, so muss diese dekomprimiert werden und kann erst danach verarbeitet werden.

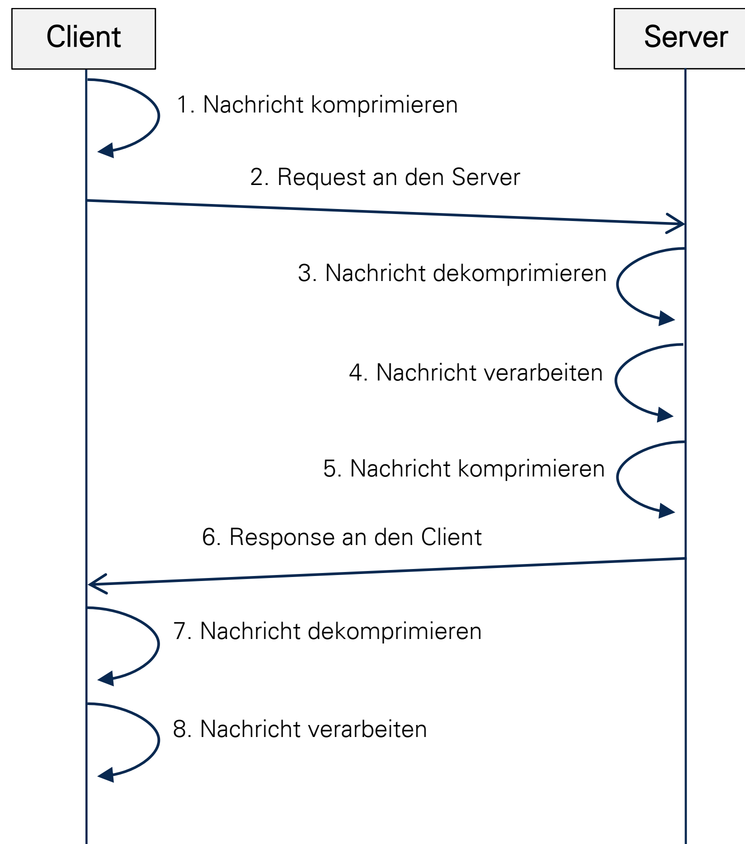


Abbildung 24 Ablaufdiagramm einer komprimierten Nachricht

4.1.6 Nachrichten bündeln

In diesem Abschnitt werden verschiedene Methoden vorgestellt, die versuchen Nachrichten zusammenzufassen bzw. sie effizienter zu verarbeiten. Auch hier bietet HTTP bereits einige Ansätze, die allerdings auch ihre Vor- und Nachteile haben.

4.1.6.1 TCP-Verbindung wiederverwenden

Wenn eine TCP-Verbindung aufgebaut bzw. abgebaut wird, dann wird ein sogenannter „Drei-Wege-Handschlag“ durchgeführt. Dabei sendet zum Beispiel der Client eine Nachricht (SYN-Paket) zum Server, sodass eine Verbindung aufgebaut werden soll. Der Server sendet daraufhin eine Bestätigung (ACK-Paket) für den Verbindungsaufbau zurück an den Client. Als Letztes bestätigt auch der Client die Nachricht des Servers. Nun wurde die Verbindung aufgebaut und der Client kann sofort im Anschluss die ersten Datenpakete senden. Doch zunächst werden die Pakete nur in kleinen Größen versendet, da die Netzauslastung zwischen Sender und Empfänger noch nicht bekannt ist. Mit dem Versenden von weiteren Datenpaketen, werden die Paketgrößen Stück für Stück angehoben, bis eine bestimmte

Schwelle erreicht ist, in der Daten zuverlässig übertragen werden können. Dieses Verfahren wird Slow-Start genannt.[19] Abbildung 25 zeigt den Ablauf eines Drei-Wege-Handschlages beim Aufbau einer TCP-Verbindung zwischen Client (Browser) und Server. Die Nachrichten „SYN“ und „ACK“ stehen für „synchronize“ und „acknowledgement“.

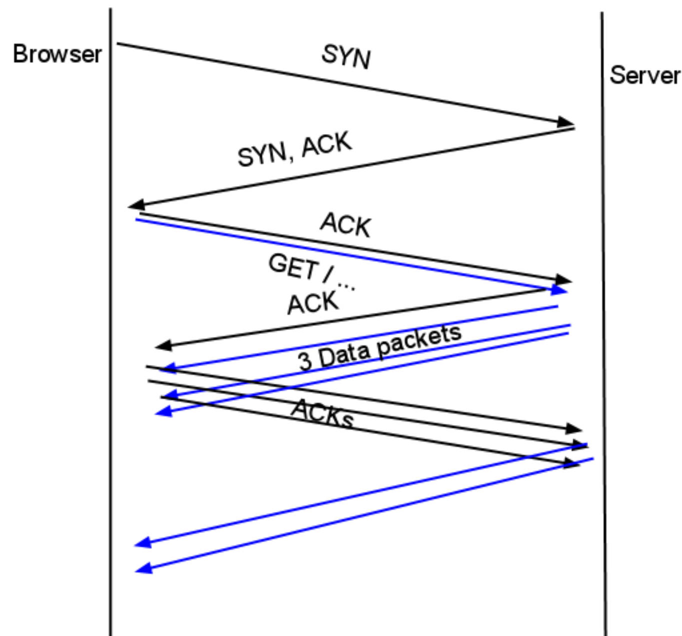


Abbildung 25 Drei-Wege-Handschlag beim Aufbau einer TCP-Verbindung (Sajal, 2011)

HTTP/1.0 baut laut Spezifikation vor jedem HTTP-Request eine neue TCP-Verbindung auf und schließt sie nach Erhalt der Antwort vom Server wieder. Durch jeden weiteren HTTP-Request und dem damit verbundenen Verbindungsauf- und Verbindungsabbau entsteht Overhead. Des Weiteren bremst der TCP-Slow-Start aus. Vor allem dann, wenn der HTTP-Header oder die HTTP-Nachricht an sich größer ist als die Paketgröße. (Vgl. Spero, o. J.)

Mit HTTP/1.1 wurden persistente Verbindungen eingeführt. Damit ist es möglich, eine TCP-Verbindung offen zu halten (keep-alive) und mehrere HTTP-Nachrichten über die gleiche Verbindung zu senden und zu empfangen. Die Verbindung wird aber nur für einen bestimmten Zeitraum offen gehalten. Wenn keine weiteren Nachrichten gesendet wurden, wird die Verbindung nach Auflauf des sogenannten „timeout“ geschlossen.[20]

---

[19] Weitere Informationen zum genauen Ablauf des Slow-Starts finden sich zum Beispiel auf http://packetlife.net/blog/2011/jul/5/tcp-slow-start/ (aufgerufen am 23.07.14)

[20] Eine Verbindung kann aber auch manuell geschlossen werden, wenn sich das Header-Feld „Connection: close“ im Header einer HTTP-Nachricht befindet.

Um bei der Kommunikation zwischen Client, SANE und BS ein optimales Ergebnis zwischen Serverlast und effizienter Übertragung zu erzielen, muss der Wert für „timeout“ entsprechend gewählt werden. Ein zu hoher Wert kann bei vielen Clients eine hohe Serverlast verursachen, ein zu niedriger Wert hingegen zu viel Overhead. Durch einen optimalen Wert werden also weniger Verbindungen neu aufgebaut, was CPU-Zeit und Speicher spart und Daten werden effizienter übertragen. (Vgl. Fielding, et al., 1999)

4.1.6.2 Multi-Messages

Gerade bei der Kommunikation zwischen Client und SANE beinhalten die HTTP-Nachrichten zusätzliche Daten im Body. Nachrichten, welche die gleiche HTTP-Methode mit Header-Daten verwenden und zusätzlich Daten im Body übertragen möchten, lassen sich zu einer „Multi-Message“ kombinieren. Die verschiedenen „Bodys“ werden einfach zusammengefasst und durch eine spezielle Markierung, wie einer Leerzeile, voneinander getrennt. In Abbildung 26 ist eine POST-Methode dargestellt, die zwei verschiedene Nachrichteninhalte miteinander kombiniert und sie durch eine Leerzeile als Unterscheidungsmerkmal kennzeichnet. Empfängt der Server diese sogenannte Multi-Message, muss er als Erstes überprüfen ob sich im Body das vorher festgelegte Markierungszeichen befindet. Ist das der Fall, so befinden sich höchstwahrscheinlich mehrere Nachrichteninhalte im Body. Der gesamte Body muss somit vorher vollständig gelesen werden und bei Auftreten des Markierungszeichens geteilt werden. In der Nachricht in Abbildung 26 befinden sich zwei Nachrichteninhalte und ein Markierungszeichen (in diesem Fall eine Leerzeile). Beide Inhalte können nach der Teilung verarbeitet werden. Da die Methode POST weder sicher noch idempotent ist, empfiehlt es sich, die Inhalte in vorgegebener Reihenfolge zu verarbeiten. Es können leicht unerwartete Ergebnisse bei der Verarbeitung entstehen, wenn sich die Inhalte durch ihre Auswirkungen gegenseitig beeinflussen. Der Server sieht die Multi-Message immer noch als eine einzige Anfrage. Aus diesem Grund sendet er auch erst nach Verarbeitung des gesamten Bodys eine Antwort zurück zum Client. An diesem Punkt könnte man das Verhalten des Servers so verändern, dass er auf jeden verarbeiteten Nachrichteninhalt eine Antwort sendet. Somit kann der Client in gewissem Maße reagieren.

Client-Anfrage

```
POST / HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 69

method=abcd&position=efgh&material=ijkl

method=mnop&user=qrst&id=uvwx
```

Abbildung 26 HTTP-POST Methode mit gebündelten Bodys

Genauso wie die Anfrage an den Server lässt sich auch die Antwort für den Client bündeln. Das Verfahren ist dabei das gleiche, wie schon oben beschrieben wurde, nur dass der Server die Nachrichten zusammenfasst und der Client sie wieder trennen muss. Eine Alternative zu dem oben vorgestellten Verfahren existiert bereits in Kombination mit JSON.[21] (Vgl. Google Inc., 2014) Dabei können sogar verschiedene Pfade zur Ressource, Header-Felder und Nutzdaten gebündelt übertragen werden. Zusätzlich sind auch einzelne Antworten an den Client möglich.

Durch die Multi-Message lässt sich eine ganze Menge an Daten einsparen. Als Beispiel dient wieder die HTTP-Anfrage aus Abbildung 26. Die Gesamtlänge dieser Anfrage inklusive Header-Daten beträgt 177 Bytes. Würden beide Bodys getrennt mit den gleichen Header-Feldern übertragen werden, dann wäre die erste Anfrage 146 Bytes und die zweite 136 Bytes, insgesamt also 282 Bytes. Daraus ergibt sich eine Einsparung von rund 37%.

$$Einsparung = 1 - \frac{Nachricht\ kombiniert}{Nachricht\ A + Nachricht\ B} = 1 - \frac{177\ Bytes}{146\ Bytes + 136\ Bytes} \approx 37\%$$

Der genaue Wert ist abhängig von der Gesamtlänge der Header-Felder sowie dem Nachrichteninhalt. Aufgrund des relativ hohen Einsparpotentials sollten Vorgänge, die mehrere Nachrichten hintereinander absenden, Multi-Messages nutzen.

4.1.6.3 Zeitfensterbündelungen

Multi-Messages, welche im Abschnitt 4.1.6.2 beschrieben wurden, können auch in Kombination mit einem Zeitfenster eingesetzt werden. Wenn Nachrichten in einer dynamischen Reihenfolge versendet werden, ist es oft schwierig, diese in einer

[21] JSON (JavaScript Object Notation) ist ein menschenlesbares Datenformat, welches zum Datenaustausch zwischen Anwendungen dient.

Multimessage zusammenzufassen. Das Zeitfenster soll dazu dienen, dass Nachrichten nur in gewissen Abständen versendet werden. Alle Nachrichten, die innerhalb eines Zeitfensters versendet werden sollen, werden zwischengespeichert und möglichst in Multi-Messages zusammengefasst. Nachdem das Ende des Fensters erreicht wurde, werden die Multi-Messages und eventuelle nicht kombinierbare Nachrichten versendet. Der Vorteil besteht darin, dass es nun viel einfacher möglich ist Nachrichten zu bündeln. Der Nachteil ist jedoch, dass die Reihenfolge möglichst erhalten bleiben muss und das ist wiederum ein erschwerender Faktor bei der Realisierung des Zeitfenster-Verfahrens. Die Verzögerung bis eine Nachricht wirklich gesendet wird, wird um die Dauer des Zeitfensters verlängert. Umso länger die Dauer eines Zeitfensters, desto wahrscheinlicher ist es, Nachrichten bündeln zu können. Auch das Zwischenspeichern von Nachrichten verbraucht zusätzlichen Speicher, der mit bedacht werden muss. Es muss also ein Kompromiss zwischen steigender Verzögerung und Bündelungswahrscheinlichkeit getroffen werden. Der zusätzliche Speicheraufwand ist stark von den zu übertragenden Daten abhängig. So sind es bei einfachen Parametern nur wenige Kilobytes, wohingegen es bei Multimedia-Inhalten mehrere Megabytes werden können.

4.1.6.4 HTTP-Pipelining

Seit HTTP/1.1 ist es möglich, HTTP-Nachrichten im Pipeline-Verfahren zu übertragen. Dabei werden mehrere Anfragen an den Server gesendet, ohne auf die Antwort zu warten. Das Verfahren setzt eine persistente Verbindung voraus, die bereits in Abschnitt 4.1.6.1 beschrieben wurde. Sowohl Server, als auch Client müssen pipelining unterstützen. Aus diesem Grund wird bei einer neuen Verbindung zunächst eine normale Anfrage versendet, um zu überprüfen, ob pipelining möglich ist. HTTP-Pipelining ist vor allem bei Verbindungen mit hohen Latenzen[22] sinnvoll. Ein Client sollte nur idempotente Methoden wie GET oder HEAD im pipeline-Verfahren versenden, da sonst unbestimmte Fehlverhalten auftreten können. Falls doch nicht-idempotente Methoden versendet werden sollen, dann sollte auf die Antwort der vorhergehenden Anfrage gewartet werden. Der Server muss auch in der gleichen Reihenfolge antworten, wie er die Anfragen erhalten hat. Das hat allerdings den Nachteil, dass eine Anfrage, die eine längere Bearbeitungszeit benötigt, alle folgenden Antworten verzögert.

[22] Latenz oder auch Latenzzeit ist die Verzögerung bis ein erwartetes Ereignis eintritt.

4.2 NACHRICHTENÜBERTRAGUNG DURCH WEBSOCKETS

Bisher basierten alle vorgestellten Verfahren auf HTTP. Es wurde versucht die Datenmenge zu reduzieren, indem die zu übertragenden Nutzdaten gekürzt wurden. Ein Großteil der Datenmenge entsteht durch die Headerdaten von HTTP. Durch WebSockets lassen sich diese weiter reduzieren. WebSockets eignen sich hervorragend, um Daten zwischen Client und Server zu übertragen. Des Weiteren können beide Kommunikationspartner bidirektional Daten versenden, ohne vorher auf eine Anfrage warten zu müssen. Der Vorteil durch den Wegfall der HTTP-Headerdaten ist zugleich ein Nachteil, denn es existiert immer die gleiche Verbindung und somit der gleiche Pfad zum Partner. In HTTP hingegen wird der Pfad zur Ressource bei jeder Anfrage angegeben. Dieses Problem lässt sich leicht umgehen, indem der Ressourcenpfad zusätzlich zu den Nutzdaten angegeben wird. Ein weiterer Nachteil ist, dass bei WebSockets keine klare Unterscheidung zwischen sicheren bzw. idempotenten Methoden mehr möglich ist. WebSockets sind nicht mehr zustandslos wie HTTP. Da jedoch hauptsächlich ein reiner Datenaustausch zwischen Client und SANE mit WebSockets betrachtet werden soll, wird hier nicht auf dieses Problem eingegangen. WebSockets basieren auf Sockets und sind somit auch auf Android-Clients möglich. Um die Request-Response-Beziehung von HTTP beizubehalten, können Unterprotokolle eingesetzt werden, die eine Antwort genau einer Anfrage zuordnen können. Dafür gibt es bereits einige Implementierungen.[23] Eigentlich sind WebSockets für die Kommunikation zwischen Server und Clients in Form von Webbrowsern gedacht. Allerdings ist auch eine Server-zu-Server-Kommunikation möglich.

Der Gewinn an Einsparung durch WebSockets soll anhand des Beispiels aus Abbildung 21 gezeigt werden. Da hier nur die Kommunikation an sich betrachtet wird, wird der Verbindungsaufbau des WebSockets vernachlässigt. Die Gesamtlänge der Beispielnachricht beträgt 146 Bytes, die Gesamtlänge der WebSocket-Nachricht jedoch nur 41 Bytes, wovon der WebSocket-Header nur 2 Bytes benötigt. Daraus ergibt sich folgende Berechnung:

$$Einsparung = 1 - \frac{WebSocket - Nachricht}{HTTP - Nachricht} = 1 - \frac{41\,Bytes}{146\,Bytes} \approx 72\%$$

Das Einsparpotential für jede gesendete Nachricht ist riesig. Dieser Wert wird allerdings nur erreicht, wenn nur die reinen Nutzdaten übertragen werden. Alle zusätzlichen Header-Informationen entfallen hier und können somit auch nicht genutzt werden. Bei so einem

[23] Eine Beispielimplementierung befindet sich unter: http://alabor.me/articles/request-response-oriented-websockets/ (aufgerufen am 30.07.2014)

kleinen WebSocket-Header verbessert sich das Ergebnis durch eine Bündelung von Nutzdaten nur noch minimal. Denn wenn eine bestimmte länge an Nutzdaten erreicht ist, dann wächst die Header-Länge um ein bis zwei Bytes. Auch kann sich die Zwischenspeicherung von einer zu bündelnden Nachricht, wie in Abschnitt 4.1.6.3 beschreiben, negativ auf die Latenz auswirken. Sind die Nutzdaten lang genug, so lässt sich wieder eine GZIP-Komprimierung in Betracht ziehen. Dadurch entsteht aber das Problem, dass der Kommunikationspartner nicht weiß, ob eine Nachricht komprimiert wurde oder nicht. Am einfachsten lässt sich das Problem beheben, indem zum Beispiel die Zeichenfolge „gzip=“ unkomprimiert vor die GZIP-komprimierte-Zeichenfolge gestellt wird. Der Empfänger muss aber nun bei jeder Nachricht die ersten drei Bytes auf dieses Muster prüfen. Der Aufwand für diese drei Zeichen sollte jedoch vernachlässigbar gering sein.

4.3 ALTERNATIVEN ZU HTTP

Eine Alternative zu HTTP um Nachrichten zu versenden, bieten Sockets. Sowohl in JAVA, als auch in PHP stehen diese zur Verfügung. Mit Sockets lassen sich Daten hardwarenahe von Endpunkt zu Endpunkt übertragen. Da die Clients kabellos mit dem SANE kommunizieren, ist es wichtig, verlorene Pakete neu zu senden. Aus diesem Grund sollte TCP als Netzwerkprotokoll gewählt werden. HTTP und WebSockets basieren auf Sockets. Beide besitzen jedoch einen Protokolloverhead wodurch die zu sendende Datenmenge steigt. Um möglichst viel Overhead im Datenverkehr (engl. Traffic) zu sparen, können die Daten direkt zum Server gesendet werden. Die Schwierigkeit besteht darin, ein eigenes anwendungsspezifisches Protokoll zu erschaffen. Dafür gibt es verschiedene Wege. HTTP könnte als Grundlage dienen und so verändert werden, dass die Header-Daten möglichst klein sind. Die Namen der Header-Felder könnten gekürzt werden. Auch könnte versucht werden alle möglichen Leerzeichen zu entfernen, um den Header möglichst klein zu halten. Ein weiterer Weg ist komplett bei Null anzufangen und ein eigenes RPC-Protokoll (Remote Procedure Call) zu entwickeln. Die Kommunikation zwischen Client und SANE entspricht dem RPC-Schema, bei dem durch Parameter festgelegt wird, welche Methoden mit welchen Werten auf dem Server aufgerufen werden. Im Anschluss wird auf eine Antwort gewartet.

Ein eigenes Protokoll zu entwickeln hat viele Vor- und Nachteile. Zum einen lässt sich so der Overhead auf ein Minimum reduzieren, zum anderen ist der Aufwand dafür sehr hoch. Auch die Latenz sinkt auf ein Minimum. Durch eigene Protokolle geht zum Beispiel auch die Unterstützung durch Browser verloren, die sehr nützlich sein kann. Trotz der vielen Möglichkeiten mit einem eigenen Protokoll, ist es nicht ohne weiteres möglich die TCP-

Verbindung an sich zu verändern. Viele Parameter für eine TCP-Verbindung werden durch das Betriebssystem festgelegt. Durch den Slow-Start von TCP können zunächst keine größeren Mengen an Nutzdaten versendet werden. Es existieren einige Versuche, bei denen die maximale Anfangsgröße an Nutzdaten (MSS) bei TCP im Betriebssystem verändert wurde, sodass die Übertragungsdauer für eine neu eröffnete TCP-Verbindung reduziert werden konnte. (Vgl. Davies, 2011)

4.4 WEITERE VORGEHENSWEISE

In diesem Kapitel wurden verschiedene Methoden gezeigt, die zu einer Verbesserung bezüglich des Overheads und der Effizienz führen können. Im darauffolgenden Kapitel 5 wird ein Prototyp für die ausgewählte Methode entworfen.

Es hat sich gezeigt, dass das Ersetzen von Parameternamen ein gutes Einsparpotential bietet. Allerdings ist es stark von dem Inhalt der Nutzdaten abhängig. Werden keine parametrisierten Methoden auf dem Server aufgerufen, sondern nur reine Informationsdaten gesendet, dann hat dieses Verfahren keine Wirkung mehr. Deshalb wird dieses Verfahren nicht mit im Prototyp eingebracht. Bei Verwendung von Cookies hat sich schnell gezeigt, dass dessen Verwendung für eine Datenübertragung wenig sinnvoll ist. Darum werden auch Cookies nicht bei der Implementierung verwendet. Die Bündelung von Nachrichten hat zwar ein hohes Einsparpotential, doch ist der Verwaltungsaufwand für die Zwischenspeicherung von Nachrichten zu groß.

Die sinnvollste Vorgehensweise beruht auf der Komprimierung von Nachrichten, sowie dem Einsatz von WebSockets. Bereits in Abschnitt 4.2 wurde beschrieben, dass WebSockets ein sehr großes Potential besitzen, um Overhead zu sparen und Daten effizient zu übertragen. Des Weiteren bietet das Protokoll die Möglichkeit zur Verschlüsselung und asynchronen Datenübertragung. Außerdem kann der Nachrichteninhalt zusätzlich komprimiert werden, um so noch mehr an Daten bei der Übertragung einzusparen. In Kapitel 3 wurde schon ersichtlich, dass eine serverseitige Komprimierung bereits eingesetzt wird. Eine Verbesserung liegt darin, sie auch clientseitig einzusetzen. Die Komprimierung der Nachrichten senkt die Datenmasse, aber erhöht jedoch die Prozessorauslastung. Zunächst wird versucht, den Einsatz von WebSockets zu ermöglichen und somit möglichst den Overhead zu senken und die Effizienz zu steigern.

4.4.1 Erwartungen an den Prototyp

Die Erwartungen für die Einsparung an Overhead liegen bei rund 70%. Gleichzeitig sollten die Prozessor- und Speicherauslastung gleichbleibend oder nur geringfügig steigen. Auch die Latenz zwischen Server und Client sollte deutlich zurückgehen, wenn die gleiche Verbindung genutzt wird. Das vorhandene System von MapBiquitous soll möglichst bestehen bleiben. Falls nötig, kann es dennoch verändert werden. Die Möglichkeit einer zukünftigen verschlüsselten Verbindung wie HTTPS soll auch erhalten bleiben.

5 IMPLEMENTIERUNG

Bevor mit der Implementierung begonnen werden kann, sind einige Voraussetzungen notwendig. Da WebSockets weitestgehend normale HTTP-Verbindungen ersetzen sollen, würde ein kompletter Umbau im MapBiquitous-Projekt den verfügbaren Zeitrahmen sprengen. Weiterhin ist die vorhandene Software, wie schon in Kapitel 3 beschrieben, nicht vollständig funktionsfähig. Aus diesem Grund und weil sich so eine Kommunikation exakter simulieren lässt, wird der Prototyp vollständig vom vorhandenen MapBiquitous-Projekt abgekapselt. Somit sind Client und Server eigenständig und unabhängig von MapBiquitous. Der Vorteil besteht nun darin, dass die Implementierung einfacher und klarer vonstattengeht. Sowohl Client, als auch Server werden nicht den vollen Funktionsumfang der bereits vorhandenen Versionen aufweisen, da es hier lediglich um die Demonstration der Nutzbarkeit von WebSockets geht. Die hier gezeigten Quellcodes sind nur Ausschnitte. Vollständige Quellcodes liegen in elektronischer Form vor.

5.1 VORAUSSETZUNGEN

Server und Client behalten ihre aktuelle Plattform bei. Das heißt, dass der Server auch im Prototyp in Programmiersprache PHP geschrieben ist. Die Server-Software benötigt einen Apache-Server ab Version 2 und eine aktuelle PHP-Version. Der Client läuft hingegen auf Android und benötigt eine aktuelle JAVA-Version (JDK) zum Entwickeln.

5.2 VERWENDETE SOFTWARE

Bei der Entwicklung des Prototyps wurden verschiedene Tools und Software verwendet. Für die Programmierung des PHP-Servers und einen alternativen JAVA-Client zum Testen der Verbindung ohne Android, kam die Entwicklungsumgebung „Eclipse Luna“ zum Einsatz. Der

Android-Client wurde mit Googles offizieller Entwicklungsumgebung, den „Android Developer Tools“, entwickelt. Zum Starten und Testen des Servers wurde „XAMPP“ mit der PHP-Version 5.5.11 verwendet. Als Betriebssystem kamen Windows 7 Ultimate 64Bit und Android 4.2.2 zum Einsatz. Zur Überprüfung der Verbindungen wurde wieder „WireShark“ verwendet.

5.3 VORGEHENSWEISE

Zunächst wird der Server erstellt, danach der Client. Um mit der Implementierung des WebSocket-Protokolls nicht komplett von vorn zu beginnen, wird eine geeignete fertige Implementierung gewählt. Der Server benötigt eine PHP-Implementierung des WebSocket-Protokolls und der Client eine für Java bzw. speziell für Android. Ist diese gefunden, so wird eine Verbindung aufgebaut und eine Testnachricht zum Server gesendet. Hierfür soll als Nächstes ein RPC-Verhalten emuliert werden, bei dem der Client auf dem Server eine Methode mit bestimmten Parametern aufrufen kann und deren Rückgabewert vom Server zurückerhält. Dieses Verhalten entspricht weitestgehend dem des MapBiquitous-Clients, wenn Anfragen zum SANE oder Gebäudeserver gesendet werden. Um das Alte Verfahren mit dem Neuen besser vergleichen zu können, wird außerdem eine HTTP-Version von Client und Server benötigt.

5.4 ERSTELLEN DES SERVERS

5.4.1 WebSocket-Server

Zuerst wird der WebSocket-Server implementiert. Eine sehr einfach zu bedienende WebSockets-API bietet „Rachet“[24]. Als Erstes wird die Rachet-API eingebunden. Danach wird das WebSocket-Serverobjekt angelegt und die Klasse „EchoTest“ übergeben. Diese Klasse enthält dann die Methoden, die aufgerufen werden, wenn z.B. eine neue Verbindung geöffnet wurde oder eine neue Nachricht erhalten wurde. Der Server lauscht auf die lokale IP-Adresse und auf dem TCP-Port 9000. Abbildung 27 zeigt den Quellcode des WebSocket-Servers. Alle zukünftigen Veränderungen oder Erweiterungen werden in der Klasse „EchoTest“ vorgenommen.

[24] Rachet – WebSockets for PHP bietet eine API für WebSockets. Für mehr Informationen siehe unter http://socketo.me/ (aufgerufen am 17.08.2014)

WebSocket-Server

```
use Ratchet\WebSocket\WsServer;
use Ratchet\Http\HttpServer;
use Ratchet\Server\IoServer;

require '/vendor/autoload.php';
require_once ('./EchoTest.php');

$ws = new WsServer(new EchoTest());

// Disable a specific version of the WebSocket protocol
$ws->disableVersion(0);

// Turn UTF-8 checks on or off
$ws->setEncodingChecks(false);

$server = IoServer::factory(new HttpServer($ws), 9000, '0.0.0.0');
$server->run();
```

Abbildung 27 Implementierung des WebSocket-Servers

Wurde eine Nachricht erhalten, so wird als Erstes überprüft, ob der Inhalt mit „method=" beginnt. Diese Überprüfung kann beliebig verändert werden. Der Sinn liegt darin, dass der Server versuchen soll eine Methode auszuführen, wenn er eine Nachricht erhält, die so beginnt. Die Methode „parse()" zerteilt die Nachricht in „$msg" anhand eines Trennungssymbols in mehrere Teile. Danach wird versucht die entsprechende Methode, deren Name sich als Teil-String in „$msg" befindet, auszuführen. Abbildung 28 enthält einen Ausschnitt aus der Klasse „EchoTest". Sie zeigt die Methode „onMessage()", die aufgerufen wird, wenn eine neue Nachricht ankommt und die Beispielmethode „add()", die vom Client aufgerufen werden könnte. Sie addiert zwei Parameter miteinander und gibt das Ergebnis zurück. Danach kann das Ergebnis zurück zum Client gesendet werden.

EchoTest

```
class EchoTest implements MessageComponentInterface {
...
public function onMessage (ConnectionInterface $from, $msg)
{
    // message starts with "mehtod=", try to parse it
    if ($this->startsWith($msg, "method="))
    {
        $this->parse($from, $msg);
    } else {
        $from->send($msg); // send message back
    }
}
...
private function add ($param)
{
    if (count($param) < 2)
    {
        throw new Exception("Too less parameters.");
    }

    return $param[0] + $param[1];
}
...
```

Abbildung 28 Aufbau der onMessage-Methode

5.4.2 HTTP-Server

Die Verarbeitung der empfangenen Daten bei einer HTTP-Anfrage ist relativ einfach, da der Aufbau fast identisch ist, mit dem der Klasse „EchoTest“. Zuerst wird geprüft, ob Daten per POST oder GET gesendet wurden. Ist das der Fall, werden alle Daten konkateniert und durch ein Sonderzeichen voneinander getrennt. Danach wird wieder die Methode „parse()“ angewandt, welche die entsprechenden Methoden aufruft und das Ergebnis zurücksendet. Ein Ausschnitt des Quellcodes für den HTTP-Server wird in Abbildung 29 gezeigt.

HTTP-Server

```
$postData = file_get_contents("php://input"); // get raw post data

if (! empty($postData) || ! empty($_GET))
{
    $msg = $postData;

    // get all get data
    if (! empty($_GET))
    {
        // append separator symbol before get data concatenation
        $msg .= ",";

        // iterate through all values
        foreach ($_GET as $key => $value)
        {
            // concatenate key and value
            $msg .= $key . "=" . $value . ",";
        }
    }

    // message starts with "mehtod=", try to parse it
    if (startsWith($msg, "method="))
    {
        parse($msg);
    } else {
        echo $msg; // send message back
    }
} …
```

Abbildung 29 Einfacher HTTP-Server, der alle erhaltenen Nachrichten zurücksendet

5.5 ERSTELLEN DES CLIENTS

5.5.1 WebSocket-Client

Auch der WebSocket-Client benutzt eine bereits implementierte WebSocket-API mit dem Namen „Autobahn|Android“[25]. Sie ist eine der derzeit sehr wenigen WebSocket-APIs, die nicht nur für den Einsatz im Browser programmiert wurde. Das Öffnen einer Verbindung ist durch die API wieder einfach. Abbildung 30 zeigt die Grundstruktur, die eine WebSocket-Verbindung öffnet und danach in der Methode „onOpen“ eine Testnachricht sendet. Wurde eine Textnachricht durch die Methode „onTextMessage“ empfangen, wird sie über die Android-Konsole ausgegeben.

[25] Weitere Informationen zur „Autobahn|Android“ WebSocket-API finden sich unter http://autobahn.ws/android/ (aufgerufen am 19.08.2014)

WebSocket-Client

```
final WebSocketConnection connection = new WebSocketConnection();
final String uri = "ws://localhost:9000";

connection.connect(uri, new WebSocketConnectionHandler()
{
	@Override
	public void onOpen()
	{
		Log.d(TAG, "Connected to " + uri);
		connection.sendTextMessage("test message");
		…
	}

	@Override
	public void onTextMessage(String payload)
	{
		Log.d(TAG, "received: " + payload);
		…
	}

	@Override
	public void onClose(int code, String reason)
	{
		Log.d(TAG, "Connection lost...");
	}
});
…
```

Abbildung 30 WebSocket-Client, Herstellen einer Verbindung

5.5.2 HTTP-Client

Für den Android-HTTP-Client wurde die Klasse „AsyncINSANETask“ aus dem MapBiquitous-Projekt kopiert, um möglichst gleiche Verhältnisse zwischen dem Prototypen und MapBiquitous zu schaffen. Die Klasse öffnet eigenständig eine HTTP-Verbindung, um Daten zu senden und wieder zu empfangen. Somit ist es einfach, eine HTTP-Nachricht zu versenden und deren Antwort auszuwerten. In Abbildung 31 ist der Teil des Quellcodes zu sehen, der eine Nachricht sendet und die Antwort abruft. Alle weiteren Funktionen stellt Android bereits zur Verfügung.

HTTP-Client

```
…
// create POST task
AsyncINSANETask post = new AsyncINSANETask(uri, message);
// execute task and send data
LinkedHashMap<String, String> result = post.execute().get();

…
// receive response data
String responseData = result.get("message");

// check whether the response message is compressed
if (responseData.startsWith("gzip="))
{
	// decode from Base64 to binary and remove “gzip=” from the String
	Byte[] decodedData = Base64.decode(responseData.substring(5), 0);
	// decompress data
	responseData = uncompress(new ByteArrayInputStream(decodedData));
}
…
```

Abbildung 31 HTTP-Client zum Senden und Empfangen auf Android

5.6 PROBLEME

Die größten Probleme bereitete die Auswahl der fertigen WebSocket-Implementierung. Viele der ausgesuchten, basierten auf einer alten WebSocket-Protokollversion oder es kam einfach keine Verbindung zustande. Am schwierigsten war die Auswahl einer geeigneten Client-Implementierung für Android, da hier Java nicht in vollem Funktionsumfang zur Verfügung steht.

Da der Kommunikationsverlauf stark vom Verhalten des Benutzers des Clients abhängig ist, lässt sich ein genauer Ablauf nur schwer vorhersagen. Wenn sich der Benutzer bewegt, dann entsteht mehr Datenverkehr, als wenn er nur still stehen würde. Das kommt unter anderem daher, dass eine Veränderung der Position den Wechsel auf andere WLAN-Access-Points zur Folge hat. Auch ein Wechsel der Etage im Gebäude muss erkannt und ausgewertet werden.

Die MapBiquitous Client-Software ließ sich anfangs überhaupt nicht starten. Sowohl mit dem offiziellen Android-Emulator aus den Android Developer Tools (ADT), als auch mit drei verschiedenen Smartphones mit den Android Versionen 2.3.7, 4.2.2 bzw. 4.4.3 stürzte die Anwendung mit einer Reihe verschiedener Fehlermeldungen ab. Bei den Fehlermeldungen handelte es sich um mehrere „NullPointerExceptions“, sowie einer „ConcurrentModificationException“. Beide Fehlerarten wurden insoweit behandelt, dass ein Starten der Anwendung möglich war. Dennoch läuft sie nicht stabil und nur sehr langsam.

Dadurch ist es kaum möglich, einen Kommunikationsverlauf im angemessenen Maße zu simulieren und auszuwerten. Selbst die Simulation von verschiedenen GPS-Koordinaten durch den Android-Emulator zeigte keinerlei Reaktion. Damit ist es nicht möglich, einen Bewegungsablauf zu simulieren. Doch selbst wenn die Anwendung einwandfrei funktioniert hätte, so hätte sie unvorhergesehene Seiteneffekte auf die Messbarkeit des Kommunikationsverlaufs. Zum Beispiel kann sich der Kommunikationsverlauf auf einem langsameren mobilen Endgerät anders verhalten, als auf einem schnelleren. Das ist damit zu begründen, weil sich die Verarbeitungsgeschwindigkeit auf den Geräten unterscheidet. So können Anfragen schneller gesendet und Antworten schneller verarbeitet werden als auf anderen Geräten. Auch aus diesem Grund wurde der Prototyp vom MapBiquitous-Projekt ausgelagert. So kann sichergestellt werden, dass der Kommunikationsverlauf steuerbar und vorhersagbar ist.

6 EVALUATION UND AUSWERTUNG

Nachdem die Implementierung abgeschlossen ist, können verschiedene Testdurchläufe durchgeführt werden. Anhand der Tests soll gezeigt werden, ob die Erwartungen erfüllt wurden und welche Vorteile bzw. Nachteile zu erkennen sind. Obwohl die MapBiquitous-Software nur sehr eingeschränkt nutzbar ist, soll dennoch versucht werden, die Ergebnisse auf das MapBiquitous-Projekt zu beziehen.

6.1 VERWENDETE TESTKONFIGURATION

Für die Tests wurden folgende Geräte verwendet:

Server	Client
Intel i5-2300 @ 2,8GHz 8GB RAM Windows 7 Professional	Samsung Galaxy S4 Mini - Android 4.2.2
	Samsung Galaxy S3 - Android 4.3

Tabelle 5 Verwendete Testkonfiguration

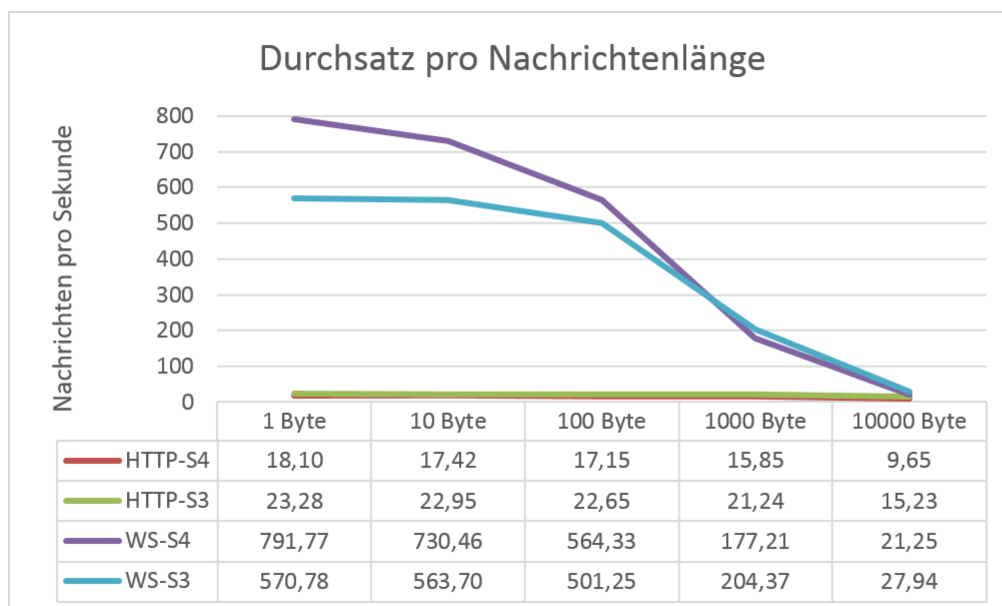
Beide Clients waren mit 54Mbit/s über WLAN mit dem Internet verbunden, der Server hingegen über ein Cat5e-Kabel. Die Internetverbindung zwischen Server und Client verfügt über einer Download-Rate von 50Mbit/s und einer Upload-Rate von 10Mbit/s. Die Ping zwischen beiden Partnern beträgt 32ms.

6.2 ERGEBNISSE DER MESSUNGEN

Die Zeitmessungen wurden in Java auf der Client-Seite durchgeführt und basieren auf den von Java bereitgestellten Nanosekunden. Das Vorgehen ist dabei immer das Gleiche. Der Client sendet eine bestimmte Anzahl von Nachrichten bzw. Anfragen an den Server. Dieser wertet die Nachrichten aus und sendet die Ergebnisse nach der Verarbeitung zurück an den Client. Die Zeitmessung beginnt mit dem Senden der ersten Nachricht und endet mit der Ankunft der letzten Antwort vom Server. Für die Endergebnisse wurden alle Messungen mehrfach ausgeführt und der Mittelwert aus allen Teilergebnissen berechnet.

6.2.1 Durchsatz

Zuallererst wurde der Durchsatz gemessen. Der Durchsatz gibt an, wie viele Nachrichten pro Sekunde in Abhängigkeit zur Nachrichtenlänge übertragen werden können. Abbildung 32 zeigt den deutlichen Unterschied zwischen HTTP und WebSockets. Solange weniger als 10 kB pro Nachricht übertragen werden, weisen WebSockets eine vielfach höhere Übertragungsrate gegenüber HTTP auf. Hierbei muss beachtet werden, dass die HTTP-Verbindung für 100 HTTP-Anfragen wiederverwendet wird. Erst danach wird eine neue Verbindung geöffnet. Die WebSocket-Verbindung wird hingegen während des gesamten Testdurchlaufs wiederverwendet.

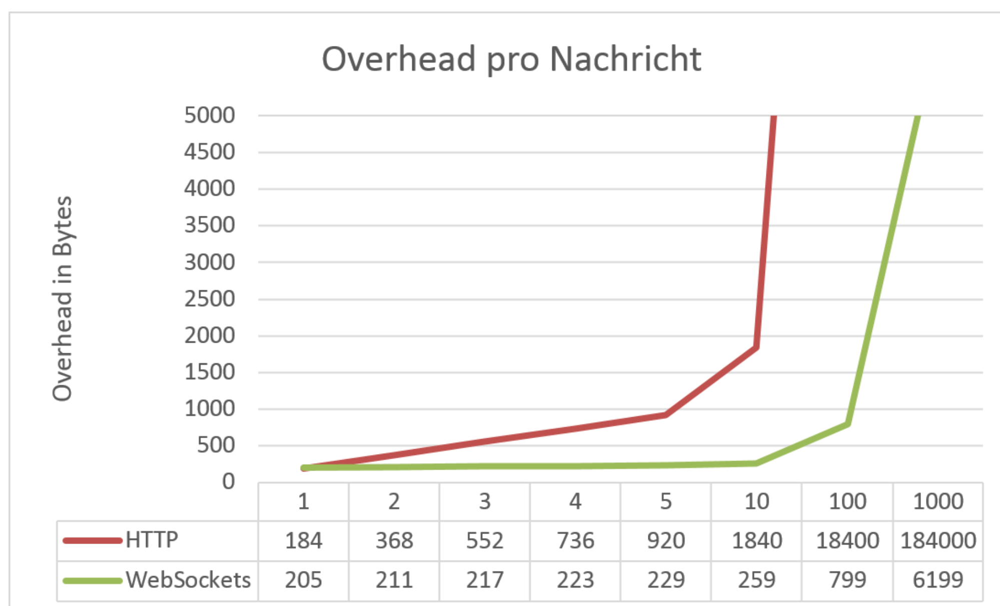


	1 Byte	10 Byte	100 Byte	1000 Byte	10000 Byte
HTTP-S4	18,10	17,42	17,15	15,85	9,65
HTTP-S3	23,28	22,95	22,65	21,24	15,23
WS-S4	791,77	730,46	564,33	177,21	21,25
WS-S3	570,78	563,70	501,25	204,37	27,94

Abbildung 32 Durchsatz pro Nachrichtenlänge

6.2.2 Overhead

Der interessanteste Aspekt an WebSockets ist die Einsparung an Daten-Overhead. Schon ab der Zweiten Nachricht erzeugt eine WebSocket-Nachricht weniger Overhead als eine HTTP-Nachricht. Es ist sehr wichtig zu beachten, dass die Overhead-Betrachtungen aus der Sicht von WebSockets gesehen werden. Denn hier wurden auch die Zieladresse des Servers sowie der Pfad zu der Ressource als Overhead definiert. In Abbildung 33 wird das noch einmal grafisch unterlegt. Die erste WebSocket-Nachricht ist durch den Handshake in Form einer HTTP-Anfrage größer, als eine reine HTTP-Nachricht. Hier ist wieder anzumerken, dass bei beiden Übertragungsverfahren der Header in Minimalform vorliegt. Das heißt, dass nur die nötigsten Header-Felder vorhanden sind. Sobald mehr Header-Daten hinzukommen, kann sich das Endergebnis deutlich verändern. Die bei der Berechnung verwendeten Header-Größen setzen sich aus dem Durchschnitt der Header aus Anfrage und Antwort zusammen. Somit gelten die berechneten Werte für die Sende- und Empfangsrichtung.



	1	2	3	4	5	10	100	1000
HTTP	184	368	552	736	920	1840	18400	184000
WebSockets	205	211	217	223	229	259	799	6199

Abbildung 33 Overhead pro Nachricht

Wenn nur der Overheadanteil einer einzelnen Nachricht im Verhältnis zu den Nutzdaten betrachtet wird, dann wird schnell ersichtlich, dass sich eine Anfrage erst dann lohnt, wenn die Nutzdaten mindestens 200 Byte betragen und somit der Overhead-Anteil unter 100% fällt. Der Unterschied zwischen den beiden Verfahren ist so gering, weil die Größe der Header-

Daten fast gleich ist. HTTP besitzt mehr Header-Daten, wohingegen WebSockets nur wenige besitzen. Doch WebSockets benötigen für eine Nachricht einen zusätzlichen Handshake in Form einer HTTP-Nachricht. Die beiden Linien fallen potentiell ab und nähern sich aneinander an, aber überschneiden sich nie.

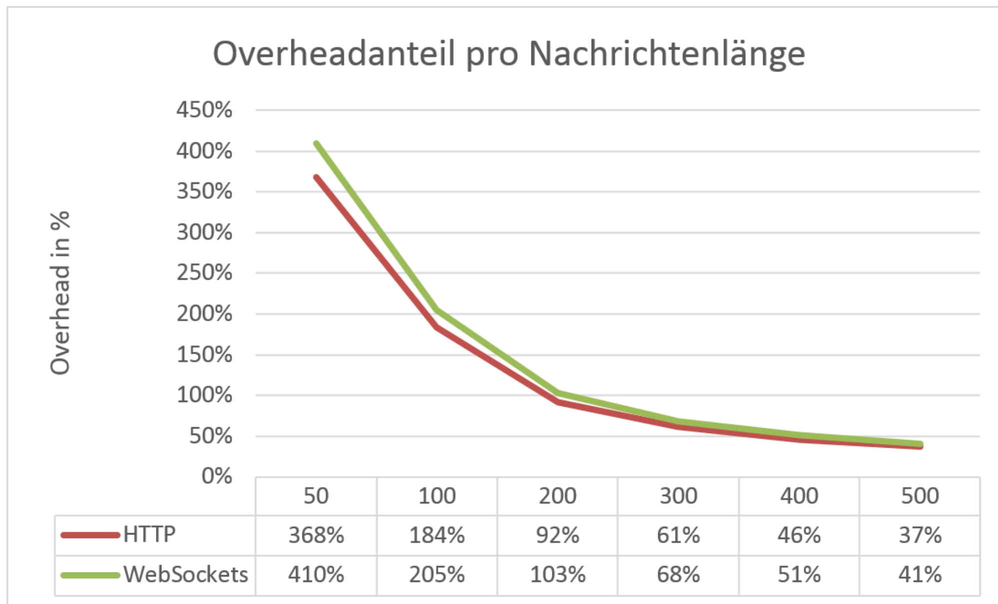
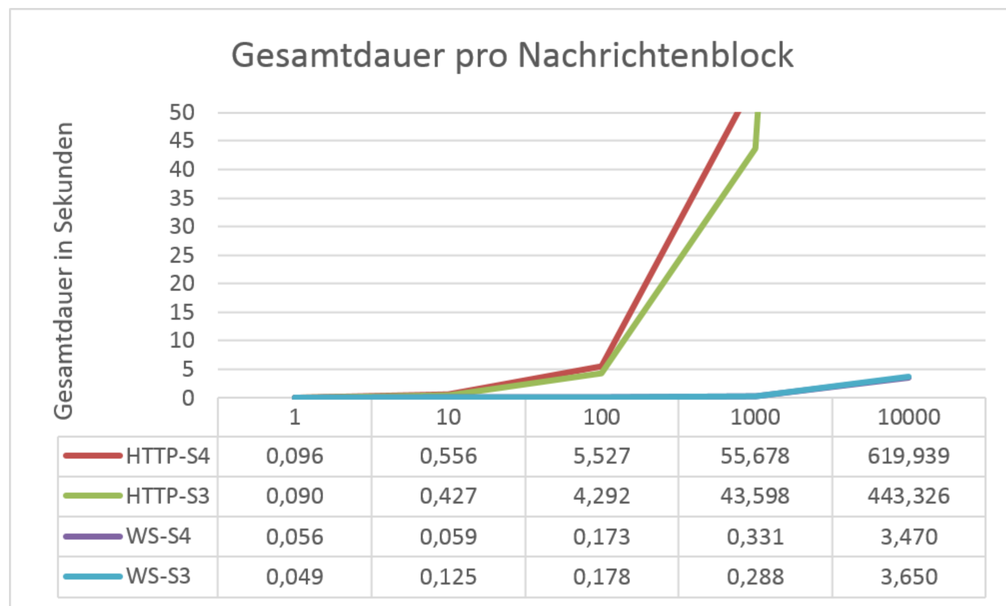


Abbildung 34 Overhead-Anteil pro Nachrichtenlänge einer einzelnen Nachricht

6.2.3 Übertragungsdauer

Bei der Messung der Überragungsdauer wurden jeweils 100 Byte Nachrichten versendet. Schon bei der Übertragung von nur einer Nachricht sind WebSockets doppelt so schnell im Vergleich zu HTTP, wie in Abbildung 35 zu sehen ist. Ab 100 Nachrichten erreicht die Übertragungsdauer von HTTP-Nachrichten unakzeptable Werte. Aus diesem Grund sollten mehrere Nachrichten gebündelt werden um diesem Verlauf entgegenzuwirken. Ein geeignetes Verfahren zur Bündelung von Nachrichten wurde bereits im Abschnitt 4.1.6.3 beschrieben. Die WebSocket-Verbindungen bleiben bis 1000 Nachrichten nahezu unverändert.

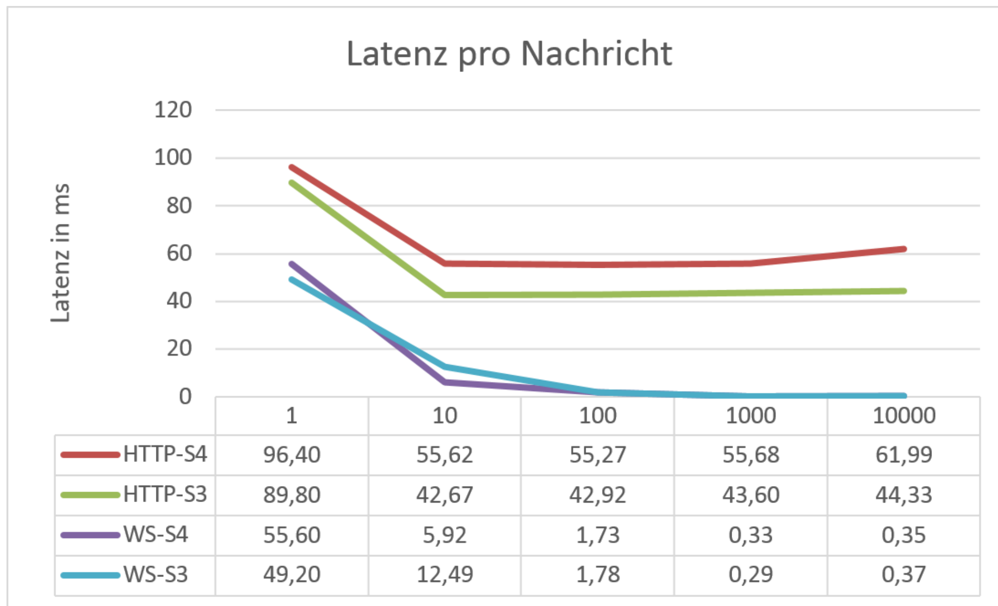


	1	10	100	1000	10000
HTTP-S4	0,096	0,556	5,527	55,678	619,939
HTTP-S3	0,090	0,427	4,292	43,598	443,326
WS-S4	0,056	0,059	0,173	0,331	3,470
WS-S3	0,049	0,125	0,178	0,288	3,650

Abbildung 35 Gesamtdauer der Nachrichtenübertragung pro Nachrichtenblock

6.2.4 Latenz

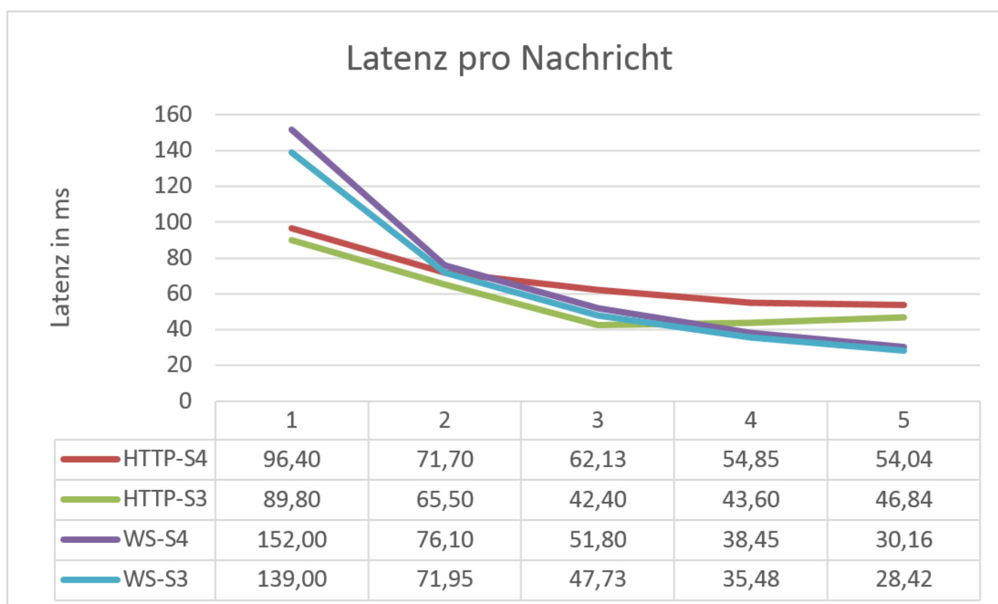
Die Latenz gibt an, wie lange es dauert, eine Nachricht zum Server zu senden und dessen Antwort zu erhalten. Die Latenz ist stark von der Verbindungsqualität des Netzwerkes und von der Verarbeitungsgeschwindigkeit des Servers abhängig. An diesem Punkt lässt sich die vorhandene Struktur in MapBiquitous nicht mit dem Test vergleichen, da bei der Beschaffung von Informationen über HTTP Long-Polling zum Einsatz kommt. Eine Messung der Latenz ist nur dann sinnvoll, wenn der Server sofort auf eine Anfrage antwortet. Die Abbildung 36 zeigt das Latenzverhalten bei einer bereits bestehenden Verbindung. Ab ca. 10 Nachrichten verlaufen die Latenzen relativ gleichbleibend niedrig. Das liegt daran, dass eine neu aufgebaute TCP-Verbindung zunächst langsam ist und mit der Zeit ihr Maximum aushandelt. Umso mehr Nachrichten gesendet werden, umso geringer ist der Einfluss des sogenannten TCP-Slow-Start. WebSockets profitieren durch ihr Streaming-Verhalten über ein und dieselbe Verbindung.



	1	10	100	1000	10000
HTTP-S4	96,40	55,62	55,27	55,68	61,99
HTTP-S3	89,80	42,67	42,92	43,60	44,33
WS-S4	55,60	5,92	1,73	0,33	0,35
WS-S3	49,20	12,49	1,78	0,29	0,37

Abbildung 36 Latenzverhalten zwischen WebSockets und HTTP

Erst bei der Betrachtung einzelner zu übertragenden Nachrichten wird deutlich, worin WebSockets ihre Schwäche haben. Der Aufbau einer WebSocket-Verbindung findet über eine HTTP-Nachricht statt. Erst danach wird auf das WebSocket-Protokoll gewechselt. Die Abbildung 37 verdeutlicht diesen Nachteil, wenn zuvor noch keine Verbindung besteht. Aus diesem Grund sind WebSockets erst ab drei bis vier zu übertragenden Nachrichten schneller, als eine reine HTTP-Verbindung.



	1	2	3	4	5
HTTP-S4	96,40	71,70	62,13	54,85	54,04
HTTP-S3	89,80	65,50	42,40	43,60	46,84
WS-S4	152,00	76,10	51,80	38,45	30,16
WS-S3	139,00	71,95	47,73	35,48	28,42

Abbildung 37 Latenzverhalten einzelner Nachrichten zwischen WebSockets und HTTP

6.3 AUSWERTUNG

Die Ergebnisse haben deutlich gezeigt, dass WebSockets sehr starke Vorteile aufweisen. Doch diese Werte können nur erreicht werden, wenn immer die gleiche Verbindung verwendet wird. Die aktuelle Struktur von MapBiquitous stellt allerdings genau da ein Problem dar. Momentan werden zwar häufig dieselben Server kontaktiert, doch der Ressourcenpfad ist immer verschieden. Solange der Pfad zur Ressource nicht über die Nutzdaten mitgesendet werden kann, um so die gleiche Verbindung wieder zu nutzen, sind WebSockets eher ein Nachteil. Für jeden neuen Ressourcenpfad muss eine neue Verbindung aufgebaut werden. Im Abschnitt 6.2.2 wurde gezeigt, dass WebSockets bei einzelnen Nachrichten mehr Overhead erzeugen, als bei HTTP. Aus diesen Gründen ergibt der Umbau auf WebSockets keinen Mehrwert. Leider konnte aufgrund der instabilen MapBiquitous-App keine genaueren Untersuchungen unternommen werden, inwiefern die gleichen Server kontaktiert werden. Nur die Gebäudeinformationen werden immer vom gleichen Server angefordert und die Nutzdaten sind dabei relativ groß. Hier würde sich der Einsatz von WebSockets dennoch lohnen.

Theoretisch gesehen wurden die Erwartungen an den Prototypen nur teilweise erfüllt. Die Einsparung von 70% der Daten-Overhead wird erst ab vier Nachrichten über die gleiche Verbindung erreicht. Da während dieser Arbeit nie solch ein Fall vorkam, in dem die gleiche Verbindung bis zu viermal wiederverwendet wurde, mit Ausnahme der Anforderung von Gebäudeinformationen, wurden die 70% nicht erfüllt. Die Prozessorlast ist hingegen gesunken, da der zusätzliche Verwaltungsaufwand beim Aufbau einer neuen Verbindung wegfällt.

HTTP ist also hinsichtlich des Overheads in der aktuellen Form von MapBiquitous besser geeignet als WebSockets, wenn nur einzelne Nachrichten versendet werden. Sobald aber die Verarbeitung der Ressourcenpfade angepasst wird, um dauerhafte Verbindungen zum Server zu erlauben, sind die Vorteile durch WebSockets enorm. Auch eine zukünftige Verschlüsselung ist über WebSockets per SSL möglich.

Der Durchsatz von WebSocket-Nachrichten beträgt bei kleineren Nachrichten weit über das 30-fache gegenüber HTTP. Auch die Latenz ist bei mehreren Nachrichten über das 10-fache geringer. So benötigen 100 Nachrichten weniger als 2 ms, um gesendet und empfangen zu werden Bei einer bereits bestehenden Verbindung hat sich gezeigt, dass mehrere 1000 Nachrichten in weniger als einer Sekunde übertragen werden können. Selbst 10.000 Nachrichten benötigen nur rund 3,5 Sekunden, wohingegen HTTP über 7 Minuten benötigt.

Das alles sind Werte, die WebSockets gerade für Echtzeitanwendungen mit Web-Technologien interessant werden lässt.

7 ZUSAMMENFASSUNG UND AUSBLICK

7.1 ZUSAMMENFASSUNG

Die Aufgabe für diese Arbeit bestand darin, die Kommunikation zwischen Client und Servern im MapBiquitous-Projekt zu optimieren. Dabei wurde zunächst die vorhandene Struktur untersucht und bewertet. Es stellte sich heraus, dass HTTP bei der Kommunikation eingesetzt wird. Des Weiteren wurde untersucht, inwieweit sich die Kommunikation optimieren lässt. Im Anschluss daran wurden verschiedene Optimierungsansätze untersucht und weitere Kommunikationsprotokollprinzipien betrachtet. Dabei hat sich gezeigt, dass WebSockets sehr viele Vorteile gegenüber den eingesetzten HTTP-Verbindungen aufweisen können. Bei der Implementierung musste zunächst eine geeignete fertige WebSocket-Implementierung für den Android-Client und den PHP-Server gefunden werden, da eine Eigenentwicklung des WebSocket-Protokolls den verfügbaren Zeitrahmen sprengen würde. Es stellte sich heraus, dass sich die WebSocket-API „Autobahn|Android“ am besten für den Android-Client und „Ratchet“ für den PHP-Server eignet. Während der Implementierungsphase stellte sich heraus, dass die MapBiquitous-App nur sehr instabil lief und der notwenige Umbau innerhalb der APP einen zu großen Aufwand darstellt. Außerdem wären mit der App keine exakten Messungen möglich gewesen, da der Kommunikationsverlauf nicht genau vorhersagbar ist. Aus diesem Grund wurde der Prototyp als eigenständige Anwendung entwickelt und getestet. Der Prototyp wurde evaluiert und ausgewertet, indem zahlreiche Zeitmessungen im Prototyp vorgenommen wurden. Die Messungen ergaben, dass WebSockets gegenüber HTTP deutlich überlegen schienen. Doch da die Bedingung für eine derartige Überlegenheit darin liegt, dass eine Verbindung immer wiederverwendet werden muss, konnten WebSockets am Ende doch nicht überzeugen. Denn momentan unterscheiden sich die Ziele der HTTP-Verbdingen in MapBiquitous zu stark, um eine Verbindung wiederverwenden zu können. Solange einzelne Nachrichten mit verschiedenen Zielen versendet werden, ist HTTP besser geeignet als WebSockets.

7.2 AUSBLICK

Im Hinblick auf die Vorteile von WebSockets sollte als Nächstes die Grundstruktur der Kommunikation von MapBiquitous umgebaut werden. Das bedeutet, dass die enormen Vorteile von WebSockets dennoch genutzt werden könnten, wenn zuvor die Handhabung der Zieladressen innerhalb des MapBiquitous-Projektes zu Gunsten von WebSockets verändert werden. Wenn nur wenige Verbindungen wiederverwendet werden können, sollte das Zusammenfassen mehrerer HTTP-Nachrichten zu einer untersucht werden. Auch ein zusätzliches Komprimieren aller Nutzdaten zur weiteren Reduzierung von Overhead sollte in Betracht gezogen werden. Es könnte auch versucht werden, die Nutzdaten an sich zu reduzieren. Zum Beispiel werden bei den meisten Anfragen die Client-Hash-ID und das Passwort des Benutzers mitgesendet, weil sich die Client-Hash-ID sonst für einen Angreifer berechnen lässt. Würde die Client-Hash-ID während des Hashvorganges mit dem Passwort kombiniert werden, dann müsste das Passwort nicht länger bei jeder Anfrage mitgesendet werden. Natürlich erfordert die neue Hash-ID besondere Behandlungen, wenn der Benutzer sein Passwort ändert. Das kommt jedoch nicht jeden Tag vor und sollte somit kein größeres Problem darstellen. Ein weiterer Punkt, der untersucht werden sollte, ist das Verhalten bei Verbindungsabbruch, wie es zum Beispiel bei mobilen Clients öfter der Fall ist. Lohnt sich bei erneutem Verbindungsaufbau der Einsatz von WebSockets immer noch?

Als letzter Punkt bleibt noch das Übertragungsprotokoll an sich zu optimieren oder zu ersetzen. Der TCP-Slow-Start könnte minimiert werden, indem die Fenstergröße beim Verbindungsaufbau generell etwas größer gewählt wird[26]. Ein Wechsel auf das schnellere UDP-Übertragungsprotokoll verursacht bei mobilen Clients eher Probleme, da bei kabellosen Verbindungen Datenpakete schneller verloren gehen und erneut gesendet werden müssen. Doch wie wäre es, wenn von der paketbasierten Übertragung abgesehen wird? Es gibt Forschungen über eine andere Übertragungsart, bei der nur einzelne Koeffizienten von Gleichungen gesendet und diese auf der Empfängerseite wieder zusammengesetzt werden. Die Übertragungsgeschwindigkeit soll dadurch um das 5 bis 10-fache gesteigert werden[27]. All diese Ideen könnten in weiteren Forschungsarbeiten näher untersucht werden.

[26] Einige Versuche zur Beschleunigung des TCP-Slow-Start wurden bereits unternommen. Weitere Informationen dazu finden sich unter http://andydavies.me/blog/2011/11/21/increasing-the-tcp-initial-congestion-window-on-windows-2008-server-r2/ (aufgerufen am 19.08.2014)

[27] Weitere Informationen über eine andere Übertragungsart, die ohne Pakete auskommt, finden sich unter http://www.en.aau.dk/News+and+Events/News//math-can-make-the-internet-5-10-times-faster.cid102747 (aufgerufen am 19.08.2014).

LITERATURVERZEICHNIS

Azad, Kalid. 2007. How To Optimize Your Site With GZIP Compression. *BetterExplained.* [Online] 8. April 2007. [Zitat vom: 17. Juli 2014.] http://betterexplained.com/articles/how-to-optimize-your-site-with-gzip-compression/.

Barth, Adam. 2011. RFC 6265 - HTTP State Management Mechanism. *Internet Engineering Task Force (IETF).* [Online] April 2011. [Zitat vom: 29. Juli 2014.] http://tools.ietf.org/html/rfc6265.

Berners-Lee, Tim, Fieldling, Roy und Frystyk Nielsen, Henrik. 1996. RFC 1945 - Hypertext Transfer Protocol -- HTTP/1.0. *The Internet Engineering Task Force.* [Online] Mai 1996. [Zitat vom: 2. Juni 2014.] http://www.ietf.org/rfc/rfc1945.txt.

Davies, Andy. 2011. Increasing the TCP Initial Congestion Window on Windows 2008 Server R2. *Andy Davies.* [Online] 21. November 2011. [Zitat vom: 2014. August 01.] http://andydavies.me/blog/2011/11/21/increasing-the-tcp-initial-congestion-window-on-windows-2008-server-r2/.

Deutsch, L. Peter. 1996. RFC 1952 - GZIP file format specification version 4.3. *The Internet Engineering Task Force.* [Online] Mai 1996. [Zitat vom: 14. Juli 2014.] http://tools.ietf.org/html/rfc1952.

Fette, Ian und Melnikov, Alexey. 2011. RFC 6455 - The WebSocket Protocol. *Internet Engineering Task Force.* [Online] Dezember 2011. [Zitat vom: 04. Juli 2014.] http://tools.ietf.org/html/rfc6455.

Fielding, Roy T. und Reschke, Julian F. 2014. RFC 7231 - Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. *The Internet Engineering Task Force.* [Online] Juni 2014. [Zitat vom: 26. Juni 2014.] https://tools.ietf.org/html/rfc7231.

Fielding, Roy T., Lafon, Yves und Reschke, Julian F. 2014. RFC 7233 - Hypertext Transfer Protocol (HTTP/1.1): Range Requests. *Internet Engineering Task Force.* [Online] Juni 2014. [Zitat vom: 9. Juli 2014.] http://tools.ietf.org/html/rfc7233.

Fielding, Roy, et al. 1999. RFC 2616 - Hypertext Transfer Protocol -- HTTP/1.1. *The Internet Engineering Task Force.* [Online] Juni 1999. [Zitat vom: 5. Juni 2014.] http://tools.ietf.org/html/rfc2616.

Google Inc. 2014. Sending Batch Requests - Google Cloud Storage. *Google Developers.* [Online] 4. Juni 2014. [Zitat vom: 25. Juli 2014.] https://developers.google.com/storage/docs/json_api/v1/how-tos/batch.

Hara, Tenshi Christian. 2012. *Untersuchung von Methoden des impliziten Crowd-Sourcing für Location-based Services in MapBiquitous.* Dresden : Technische Universität Dresden, 2012.

HTTP Message Methods - Part 2 of Chapter 3 from HTTP: The Definitive Guide (3/4) | WebReference. 2003. *WebReference.* [Online] QuinStreet, 21. Januar 2003. [Zitat vom: 26. Juni 2014.] http://www.webreference.com/programming/http/chap3/2/3.html.

HTTP: Request-Methoden - HTMLWORLD. o. J.. HTMLWORLD. [Online] Eisbär Media GmbH, o. J. [Zitat vom: 26. Juni 2014.] http://www.html-world.de/209/request-methoden/.

li_service. 2011. Lorem Ipsum web.solutions. *Hypertext Transfer Protocol (HTTP).* [Online] 10. November 2011. [Zitat vom: 2. Juni 2014.] http://www.loremipsum.at/wissen/lexikon/hypertext-transfer-protocol-http/.

Odzangba. 2009. GZIP vs. BZIP2 vs. LZMA. *Odzangba Kafui Dake.* [Online] 25. März 2009. [Zitat vom: 14. Juli 2014.] http://odzangba.wordpress.com/2009/03/25/gzip-vs-bzip2-vs-lzma/.

Ramsey, Ben. 2008. Ben Ramsey. *HTTP Status: 206 Partial Content and Range Requests.* [Online] 5. Mai 2008. [Zitat vom: 9. Juli 2014.] http://benramsey.com/blog/2008/05/206-partial-content-and-range-requests/.

Reibold, Holger. 2001. TecChannel . *Hypertext Transfer Protocol - HTTP-Grundlagen.* [Online] 30. August 2001. [Zitat vom: 2. Juni 2014.] http://www.tecchannel.de/netzwerk/management/401210/hypertext_transfer_protocol/.

Sajal. 2011. Tuning initcwnd for optimum performance. *CDN Planet.* [Online] 25. Oktober 2011. [Zitat vom: 27. Juli 2014.] http://www.cdnplanet.com/blog/tune-tcp-initcwnd-for-optimum-performance/.

Schröder, Jacob und Müller, Martin. 1999. *Webserver betreiben - HTTP und Apache: Grundlagen, Konzepte, Lösungen.* iX Edition. Heidelberg : dpunkt.verlag GmbH, 1999. ISBN 3-932588-00-2.

Spero, Simon E. o. J.. Analysis of HTTP Performance problems. [Online] o. J. [Zitat vom: 24. Juli 2014.] http://www.ibiblio.org/mdma-release/http-prob.html.

Tilkov, Stefan. 2011. *REST und HTTP.* 2., aktualisierte und erweiterte Auflage. Heidelberg : dpunkt.verlag GmbH, 2011. ISBN 978-3-89864-732-8.

Ullrich, Benjamin. 2012. WebSockets: Spezifikation / Implementierung. [Online] April 2012. [Zitat vom: 12. Juli 2014.] http://www.net.in.tum.de/fileadmin/TUM/NET/NET-2012-04-1/NET-2012-04-1_08.pdf.

Weßendorf, Matthias. 2011. WebSocket: Annäherung an Echtzeit im Web. *heise Developer.* [Online] 15. Juni 2011. [Zitat vom: 2014. Juli 12.] http://www.heise.de/developer/artikel/WebSocket-Annaeherung-an-Echtzeit-im-Web-1260189.html.

Wilde, Erik. 1999. *World Wide Web.* Berlin Heidelberg : Springer-Verlag, 1999. ISBN 3-540-64700-7.

Wöhr, Heiko. 2004. *Web-Technologien: Konzepte - Programmiermodelle - Architekturen.* Heidelberg : dpunkt.verlag GmbH, 2004. ISBN 3-89864-247-X.

Yuan, Jeffery. 2013. Java: Use Zip Stream and Base64 Encoder to Compress Large String Data. *Programmer: Lifelong Learning.* [Online] 12. November 2013. [Zitat vom: 14. Junli 2014.] http://lifelongprogrammer.blogspot.de/2013/11/java-use-zip-stream-and-base64-to-compress-big-string.html.

Zimmermann, Robin. 2012. Kaazing. [Online] Kaazing Corporation, 28. Februar 2012. [Zitat vom: 04. Juli 2014.] http://blog.kaazing.com/2012/02/28/html5-websocket-security-is-strong/.