



Bachelorarbeit im Studiengang B.Sc. Wirtschaftsinformatik

Realisierung einer bidirektionalen Chat-Anwendung auf Basis von HTML5-WebSockets mit Java und Angular 4 zum Einsatz in der Lehre

Implementation of a bidirectional chat application utilizing HTML5 WebSockets in Java and Angular 4 for teaching

Andre Weinkötz

Matrikelnummer: 14985716

15. Februar 2018

Fakultät für Informatik und Mathematik
Hochschule München

Betreuer: Prof. Dr. Mandl

Erklärung gemäß § 15 Abs. 10 APO i.V.m. § 35 Abs. 7 RaPO

Hiermit erkläre ich, dass ich die vorliegende Bachelorarbeit selbstständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

München, den 21. Februar 2018

*People keep asking me what I think of it now that it's done.
Hence my protest: The Web is not done!*

Tim Berners-Lee, Oktober 1999

Inhaltsverzeichnis

| | |
|---|-----------|
| 1. Einleitung | 6 |
| 2. Das WebSocket-Protokoll | 7 |
| 2.1. WebSocket-Frames | 8 |
| 2.1.1. Header-Felder | 8 |
| 2.1.2. Non-Control-Frames | 10 |
| 2.1.3. Control-Frames | 12 |
| 2.2. WebSocket-Kommunikationsphasen | 14 |
| 2.2.1. Verbindungsaufbau | 15 |
| 2.2.2. Datentransfer | 17 |
| 2.2.3. Verbindungsabbau | 17 |
| 2.3. Alternative Verfahren | 18 |
| 2.3.1. Polling | 19 |
| 2.3.2. Long-Polling | 19 |
| 2.3.3. Comet | 20 |
| 2.3.4. Server-Sent Events | 21 |
| 3. Die WebSocket-API | 22 |
| 3.1. Elemente der W3C-WebSocket-API | 22 |
| 3.1.1. WebSocket Lebenszyklus | 22 |
| 3.1.2. Event-Handling | 24 |
| 3.1.3. Kommunikation | 24 |
| 3.2. Die Java TM API for WebSocket (JSR-356) | 25 |
| 3.2.1. Grundlagen | 26 |
| 3.2.2. Kommunikation | 29 |
| 3.2.3. En- und Decoding | 31 |
| 3.2.4. Konfiguration und Session | 33 |
| 3.2.5. Path-Mapping | 37 |
| 4. Die Chat-Anwendung | 40 |
| 4.1. Analyse der bestehenden Chat-Anwendung | 41 |
| 4.1.1. Struktur des Projekts | 41 |
| 4.1.2. Auswahl zu übernehmender Klassen | 42 |
| 4.1.3. Funktionsweise | 43 |
| 4.2. Anforderungen an den WebSocket-Chat | 45 |
| 4.2.1. Funktionale Anforderungen | 46 |
| 4.2.2. Nicht-funktionale Anforderungen | 49 |
| 4.3. Entwurf und Implementierung des WebSocket-Chats | 50 |
| 4.3.1. Entwurf | 50 |
| 4.3.2. Implementierung | 52 |

| | |
|---|------------|
| 5. Evaluation | 57 |
| 5.1. Komplikationen | 58 |
| 5.2. Leistungsanalyse des WebSocket-Chats | 59 |
| 6. Fazit und Ausblick | 62 |
| A. Anhang | VII |

1. Einleitung

Der Austausch von Informationen ist für die heutige Gesellschaft von fundamentaler Bedeutung. Bis zur Entwicklung des World Wide Web (WWW) durch Tim Berners-Lee im Jahr 1989 variierte die digitale Repräsentation von Informationen je nach verwendetem System. Geleitet von seinem Grundgedanken eines möglichst einfachen, lizenzenfreien und verteilten Systems entwickelte er das Hypertext Transfer Protocol (HTTP), die Hypertext Markup Language (HTML) und die Uniform Resource Identifier (URI). Diese Simplizität trägt noch heute zur Popularität des WWW bei. Doch die Anforderungen an Webanwendungen haben sich in den vergangenen Jahren stark verändert. Mobile Geräte wie Smartphones oder Tablets ersetzen stationäre Systeme, Webanwendungen sollen zur Kollaboration eingesetzt werden und Buchungssysteme oder Finanzanwendungen verlangen die Bearbeitung tausender Anfragen mit minimaler Verzögerung. HTTP bietet hier keine zufriedenstellende Lösung. Um den Anforderungen an moderne Webanwendungen gerecht zu werden, spezifizierten das World Wide Web Consortium (W3C) und die Internet Engineering Taskforce (IETF) das WebSocket-Protokoll samt zugehörigem Application Programming Interface (API) in JavaScript. Dabei machten die Entwickler es sich zur Aufgabe, einen Mechanismus zu standardisieren, der es browserbasierten Anwendungen ermöglicht bidirektional zu kommunizieren, ohne dabei mehrere HTTP-Verbindungen zu öffnen [FM11].

HTML5-WebSockets sollen die Vorteile webbasierter Architekturen mit den Eigenschaften des verbindungsorientierten *Transmission Control Protocols* (TCP) kombinieren. Webanwendungen sollen mit nur wenigen Verwaltungsdaten Informationen gleichzeitig senden und empfangen können. Die zusätzliche Bereitstellung der WebSockets durch herkömmliche Webserver soll eine hohe Kompatibilität zu bestehenden Web-Infrastrukturen und Sicherheitssystemen gewährleisten. Das Ziel dieser Arbeit ist die Migration der Kommunikationsstrukturen einer bestehenden Anwendung auf HTML5-WebSockets, um von diesen Vorteilen profitieren zu können. Bei der Anwendung handelt es sich um ein Chatsystem, das im Rahmen der Lehrveranstaltung ‚Datenkommunikation‘ im dritten Semester des Studiengangs B.Sc. Wirtschaftsinformatik an der Hochschule München eingesetzt wird. Die zu entwickelnde Anwendung soll zur Ergänzung des Lehrmittel-Portfolios in passenden Lehrveranstaltungen eingesetzt werden können. Die Voraussetzung für eine fachgerechte Umsetzung der Anforderungen ist ein fundiertes Verständnis der Funktionsweise von WebSockets sowie deren Implementierung in verschiedenen Programmiersprachen.

Der Aufbau dieser Arbeit gliedert sich in sechs Kapitel. In Kapitel 2 wird das WebSocket-Protokoll anhand seiner Spezifikation detailliert erläutert. Kapitel 3 behandelt die zugehörige WebSocket-API und beschreibt deren Implementierung in der Programmiersprache Java. In Kapitel 4 wird die bestehende Chat-Anwendung analysiert, die Architektur des neuen WebSocket-Chats entworfen und anschließend implementiert. Diese Implementierung wird in Kapitel 5 evaluiert, indem die Messergebnisse der Leistungstest beider Anwendungen verglichen werden. Kapitel 6 schließt mit einem Fazit und gibt einen Ausblick auf zukünftige Entwicklungsmöglichkeiten.

2. Das WebSocket-Protokoll

Die Arbeit an der Spezifikation des WebSocket-Protokolls begann bereits 2009, als es von Google in Zusammenarbeit mit Apple, Microsoft und Mozilla im Rahmen ihrer Kooperation WHAT-WG¹ der Internet Engineering Task Force (IETF) vorgeschlagen wurde. Bis Mai 2010 wurden zahlreiche Verbesserungen hinzugefügt, bis es schließlich in Version 76 [Hic10] im August 2010 an die *BiDirectional or Server-Initiated HTTP* (HyBi) Task Force der IETF zur Weiterentwicklung übergeben wurde [Fet10]. Neben vielen anderen Fortschritten wurde das Protokoll um die Möglichkeit erweitert binäre Dateien auszutauschen und es wurden Sicherheitslücken geschlossen, die in Verbindung mit Proxy-Servern entstehen konnten. Im Dezember 2010 wurde das WebSocket-Protokoll von der IETF zu dem Request for Comment (RFC) 6455 erklärt [FM11]. Dieser definiert auf über 70 Seiten neben den Kommunikationsphasen und verschiedenen Frames der WebSockets zahlreiche weitere technische Details und Definitionen. In diesem Kapitel sollen die wesentlichen Grundlagen der Protokollspezifikation dargelegt werden.

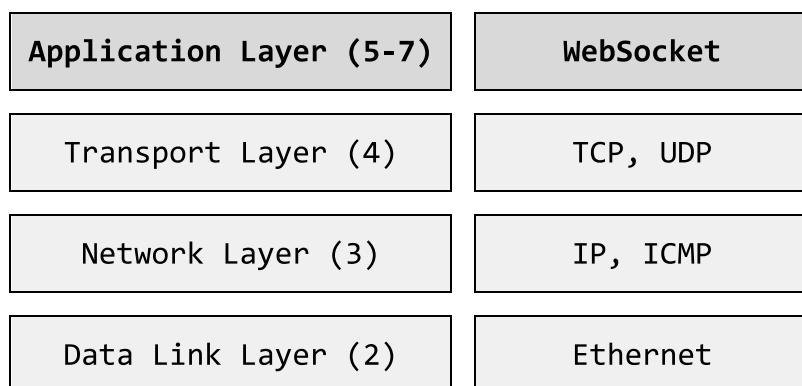


Abbildung 2.1: WebSocket-Protokoll im ISO/OSI-Modell

Die Aufgabe des WebSocket-Protokolls ist die bidirektionale Kommunikation zwischen einer (Web-)Anwendung und einem WebSocket-Server. Dabei folgt das Protokoll dem Prinzip des minimalen *Framings* [FM11, S. 9-10]. Das bedeutet, dass Zusatzinformationen zur Übermittlung von Nachrichten, wie z. B. Protokoll-Header, möglichst gering zu halten sind. Das Konzept sieht vor, dass das WebSocket-Protokoll direkt als Schicht auf das *Transmission Control Protocol* (TCP) aufsetzt und es neben HTTP in einer bereits bestehenden Infrastruktur eingesetzt werden kann. Dabei werden das Sicherheitsmodell der Same-Origin-Policy aus RFC 6454 sowie ein Adressierungs- und Bezeichnungsmechanismus hinzugefügt, durch den die Unterstützung mehrerer Dienste auf einem Port und verschiedener Hostnamen unter einer IP-Adresse sichergestellt wird. Des Weiteren soll das aufgesetzte Framing den Paketierungsmechanismus des *Internet Protocols* (IP) imitieren, ohne die Länge eines Frames zu beschränken. Außerdem wird ein zusätzlicher *Closing-Handshake* definiert, der auch bei der Verwendung von Proxy-Servern und anderen *Vermittlern*, wie z. B. Firewalls, zwischen den beiden Endpunkten durchgeführt werden

¹Web Hypertext Application Technology Working Group unter: www.whatwg.org

2. Das WebSocket-Protokoll

kann. Obwohl das WebSocket-Protokoll die für HTTP(S) vorgesehenen Ports 80 und 443 verwendet, verbindet die beiden Protokolle lediglich der *HTTP-Upgrade*-Mechanismus. Dieser kommt zum Einsatz, wenn eine WebSocket-Verbindungsanfrage von einem HTTP-Server interpretiert wird. Soll die WebSocket-Verbindung verschlüsselt sein, so kann dies durch die Verwendung eines *Transport-Layer-Security-(TLS)-Tunnels* erreicht werden.

2.1. WebSocket-Frames

Die Spezifikation eines WebSocket-Frames sieht eine Zweiteilung vor, wie sie bei Protokollnachrichten üblicherweise vorgenommen wird. Ein WebSocket-Frame besteht demnach aus einem Header Bereich der Kontrolldaten enthält, sowie dem Payload, der die Nutzdaten beinhaltet.

Anhand des Aufbaus ist leicht erkennbar, dass die Kommunikation über WebSockets mit deutlich

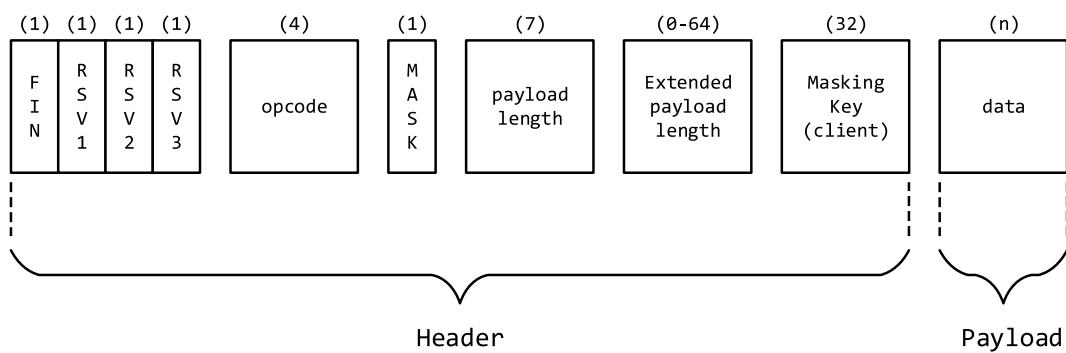


Abbildung 2.2: Vereinfachte Darstellung eines WebSocket-Frames

geringeren Paketgrößen arbeiten kann als HTTP. Eine Nachricht mit einfacher Längenangabe, die von Server zu Client geschickt wird, erzeugt durch ihren Header lediglich einen Overhead von zwei Byte. Da WebSocket-Frames auf umgekehrtem Weg zusätzlich maskiert werden müssen, entspricht die minimale Länge eines clientseitigen Frames sechs Byte. Im Vergleich dazu beträgt der Overhead bei einer einfachen GET-Anfrage über HTTP/1.1 beim Aufruf des Hosts unter cs.hm.edu 35 Byte. Die Angabe des Hostnamens in HTTP/1.1 ist obligatorisch und trägt maßgeblich zur Größe des HTTP-Frames bei [Lea+99, S. 128].

2.1.1. Header-Felder

Im nachfolgenden Abschnitt werden die Felder sowie deren jeweilige Aufgabe detailliert erläutert:

FIN (1 Bit) Das FIN-Flag gibt an, ob es sich bei dem empfangenen Frame um ein Fragment handelt. Kann der Payload nicht auf einmal übertragen werden, so wird die Übertragung über mehrere Frames gesteuert. Das FIN-Flag aller unvollständigen Fragmente weist den Wert 0 auf, wohingegen das finale Fragment mit einer 1 gekennzeichnet ist. Wenn alle Nutzdaten mit einem einzigen Frame übertragen werden können, so trägt dessen FIN-Flag ebenfalls den Wert 1. Eine Fragmentierung ist in den meisten Fällen nicht erforderlich. Liegen Daten jedoch zum Zeitpunkt des Versendens nicht vollständig vor oder können

nicht zwischengespeichert werden, so ermöglicht die Fragmentierung der Nachrichten eine teilweise Übertragung.

RSV1, RSV2, RSV3 (jeweils 1 Bit) Diese Flags sind für zukünftige Implementierungen reserviert und finden aktuell keine standardisierte Anwendung. Die Ausnahme stellt das RSV1-Bit dar, das verwendet wird, um anzugeben, ob eine Nachricht komprimiert wurde oder nicht [Int11]. Daher tragen sie meist den Wert 0. Ist eines dieser Flags gesetzt und die empfangende Stelle hat keine Erweiterung, die das jeweilige Flag interpretieren kann, so muss die WebSocket-Verbindung geschlossen werden [FM11, S. 27].

opcode (4 Bit) Das opcode-Feld enthält die Angabe darüber, wie der Payload zu interpretieren ist. Neben den sogenannten Datenframes bzw. Non-Control-Frames, die die Art der übertragenen Daten kennzeichnen, sind Control-Frames definiert, die zur Steuerung und Überprüfung der Verbindung verwendet werden. Wird ein unbekannter Opcode empfangen, so wird die WebSocket-Verbindung geschlossen.

- **0x0** definiert Fortsetzungs-Frames
- **0x1** definiert Text-Frames
- **0x2** definiert Binary-Frames
- **0x3-7** sind reserviert für weitere Non-Control-Frames
- **0x8** definiert Verbindung-beenden-Frames
- **0x9** definiert Ping-Frames
- **0xA** definiert Pong-Frames
- **0xB-F** sind reserviert für weitere Control-Frames

MASK (1 Bit) Aus Sicherheitsgründen müssen alle Frames *maskiert* werden, die von einem Client zu einem Server übertragen werden. Unter Maskierung versteht man dabei die bitweise Abwandlung der Nutzdaten mittels eines Schlüssels (Masking Key), wobei die Länge der Nutzdaten unverändert bleibt. Das MASK-Flag zeigt dabei an, ob der Payload maskiert ist. Frames, die clientseitig initiiert wurden, müssen maskiert sein, wohingegen serverseitige Frames keine Maskierung vornehmen dürfen. Der verwendete Masking Key muss im gleichnamigen Header-Feld hinterlegt werden.

payload length (7 Bit) Hier wird die Länge der Nutzdaten angegeben. Sind diese nicht größer als 125 Byte, so können sie mit nur einem Frame übertragen werden. Nutzdaten mit einem höheren Speicherplatzbedarf werden, je nach Größe, mit dem Wert 126 bzw. 127 angegeben. Diese Angaben führen zur Verwendung des Header-Felds *Extended payload length*.

Extended payload length (16 Bit oder 64 Bit) Überschreitet die Länge des Payloads 125 Byte, so wird dieses Feld verwendet. Je nach Wert im Header-Feld *payload length* umfasst die *Extended payload length* zwei Byte oder acht Byte. Dementsprechend beträgt die maximale Länge der Nutzdaten in einem WebSocket-Frame $2^{64} - 1$ Byte [GLN15, S. 41]

Masking Key (0 oder 32 Bit) Ein Frame, der von einem Client an einen Server gesendet wird, muss mit einem 32-Bit-Schlüssel maskiert werden. Dieser wird vom Client generiert und im gleichnamigen Header-Feld abgelegt. Nachrichten, die von einem Server an einen Client gesendet werden, enthalten dieses Feld nicht.

data (n Byte) Hier befinden sich die eigentlichen Nutzdaten. Ihre Größe wird in den entsprechenden Längenfeldern des Headers hinterlegt. Werden lediglich Steuerungsinformationen übertragen, wird kein Payload mitgesendet.

2.1.2. Non-Control-Frames

Durch das Header-Feld *opcode* wird die Art der Nutzdaten angegeben. Es können sowohl textbasierte Daten als auch Binärdaten übertragen werden. Bei der Kommunikation über WebSockets gibt es hier einige zusätzliche Mechanismen, die für den fehlerfreien Ablauf einer Übertragung notwendig sind. Der folgende Abschnitt handelt von der bereits aus anderen Protokollen bekannten Fragmentierung, die WebSocket-Frames zur sequentiellen Übertragung unvollständiger Daten nutzt sowie von der aufgrund von Sicherheitsproblemen eingeführten Maskierung clientseitig initierter Nachrichten.

Fragmentierung

In Abschnitt 5.4. des RFC 6455 ist die Fragmentierung von WebSocket-Frames beschrieben. Primäre Anwendung erfährt dieses Verfahren, wenn die Daten zum Zeitpunkt der Übertragung noch nicht vollständig vorliegen oder nicht zwischengespeichert werden können. Wird die vom Server oder einer vermittelnden Stelle festgelegte Puffergröße überschritten, so wird ein Nachrichtenfragment mit dem Inhalt des Puffers gesendet [FM11, S. 32].

Folgende Grundsätze müssen bei der Fragmentierung beachtet werden, Erweiterungen sind hierbei außer Acht gelassen:

- Unfragmentierte Nachrichten werden mit einem einzelnen Frame übertragen (FIN-Bit = 1, opcode ≠ 0).
- Die Übertragung fragmentierter Nachrichten erfolgt über mindestens einen Frame mit FIN-Bit = 0 und opcode ≠ 0 und wird mit einem Frame FIN-Bit = 1 und opcode ≠ 0 abgeschlossen. Die Summe der Payloads aller Fragmente entspricht dem Payload eines Einzelframes bei entsprechender Puffergröße.
- Control-Frames (Abschnitt 2.1.3) dürfen nicht fragmentiert werden. Sie können zwischen der Übertragung einer fragmentierten Nachricht gesendet werden, um beispielsweise die Latenzzeit eines Pings möglichst gering zu halten. Die Verarbeitung zwischengesendeter Control-Frames muss von allen Endpunkten unterstützt werden.
- Der Empfänger muss die Fragmente einer Nachricht in der gleichen Reihenfolge erhalten, in der sie vom Sender aufgegeben wurden.

2. Das WebSocket-Protokoll

- Vermittler dürfen an der Fragmentierung keine Änderungen vornehmen, falls eines der RSV-Bits gesetzt ist und dessen Bedeutung dem Vermittler nicht bekannt ist.

Daraus folgt, dass die Verarbeitung sowohl fragmentierter als auch unfragmentierter Nachrichten von allen Endpunkten unterstützt werden muss. Da Control-Frames nicht fragmentiert werden dürfen, können Fragmente lediglich von den Typen Text, Binary oder einem der reservierten opcodes (0x3-7) sein. Die Angabe des Typs erfolgt durch den opcode des ersten Fragments. Alle weiteren Fragmente tragen den opcode 0x0, der dem Empfänger mitteilt, dass die empfangenen Nutzdaten an den Payload des vorangegangenen Frames angehängt werden sollen [She17].

Maskierung

WebSocket-Frames müssen maskiert werden, wenn sie von einem Client zu einem Server gesendet werden. Dieser Mechanismus war im ursprünglichen Draft für das WebSocket Protokoll nicht vorgesehen. In Folge einer Untersuchung durch eine Gruppe von Forschern rund um ein Team der Carnegie-Mellon-Universität wurden Sicherheitslücken in dem Konzept aufgedeckt. Bei der Verwendung transparenter Proxy-Server konnten sie durch eine fehlerhafte Implementierung des bei dem Aufbau einer WebSocket-Verbindung genutzten *Upgrade*-Mechanismus den Cache des Proxies infizieren [Hua+11]. Daraufhin wurde der Draft überarbeitet und um die Maskierung über eine bitweise XOR-Verknüpfung ergänzt. Da der zur Maskierung des Payloads verwendete Schlüssel im Frame enthalten ist, steht hier nicht die Vertraulichkeit der gesendeten Daten im Vordergrund. Vielmehr sollen Proxy-Server daran gehindert werden, den Inhalt des Payloads zu lesen. Somit wird vermieden, dass Angreifer den Payload manipulieren und diesen für einen Angriff gegen einen Proxy einsetzen können [GLN15]. Im Folgenden soll der Ablauf der Maskierung einer Textnachricht an einem Beispiel verdeutlicht werden:

Zunächst wählt der Client einen bisher nicht verwendeten 32-Bit-Schlüssel aus, der zur Maskierung verwendet werden soll. Die Nachricht, die maskiert werden soll, lautet "cs.hm.edu", konvertiert in hexadezimale Darstellung "63 73 2e 68 6d 2e 65 64 75". Als Schlüssel wird "3c 2e 3f 4a" verwendet. Dieser wird nun zyklisch auf den zu maskierenden Payload mit der XOR-Operation angewendet.

| | | | | | | | | | |
|------------------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| Unmaskierte Byte | 63 | 73 | 2e | 68 | 6d | 2e | 65 | 64 | 75 |
| Operator | \oplus |
| Schlüssel | 3c | 2e | 3f | 4a | 3c | 2e | 3f | 4a | 3c |
| Maskierte Byte | 5f | 5d | 11 | 22 | 51 | 00 | 5a | 2e | 49 |

Tabelle 2.1: Maskierung durch bitweise XOR-Verknüpfung

Die maskierte Nachricht wird mit dem Schlüssel im WebSocket-Frame abgelegt. Der Server kann diese nach dem Empfang durch erneute Anwendung der selbstinversen XOR-Operation demaskieren [GLN15].

2. Das WebSocket-Protokoll

| | | | | | | | | | |
|------------------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| Maskierte Byte | 5f | 5d | 11 | 22 | 51 | 00 | 5a | 2e | 49 |
| Operator | \oplus |
| Schlüssel | 3c | 2e | 3f | 4a | 3c | 2e | 3f | 4a | 3c |
| Unmaskierte Byte | 63 | 73 | 2e | 68 | 6d | 2e | 65 | 64 | 75 |
| Nachricht | c | s | . | h | m | . | e | d | u |

Tabelle 2.2: Demaskierung durch bitweise XOR-Verknüpfung

Die Übertragung von Server zu Client darf hingegen nicht maskiert werden. Empfängt ein Server eine unmaskierte bzw. ein Client eine maskierte Nachricht, so muss in beiden Fällen die Verbindung geschlossen werden. Dabei kann der im RFC 6455 spezifizierte Statuscode *1002 (protocol error)* beim Verbindungsabbau angegeben werden [FM11, S. 26].

2.1.3. Control-Frames

Control-Frames werden dazu genutzt, den Status einer WebSocket-Verbindung zu steuern und zu überwachen. Der RFC 6455 unterscheidet dabei drei Arten von Control-Frames, die anhand des *opcodes* unterschieden werden. Der Close-Frame wird zur Einleitung und Bestätigung einer schließenden Verbindung eingesetzt. Ping- bzw. Pong-Frames werden genutzt, um festzustellen, ob der jeweilige Endpunkt noch erreichbar ist oder um eine Verbindung aktiv zu halten (keep-alive) [FM11, S. 35-36]. Im Folgenden werden die verschiedenen Arten der Control-Frames sowie deren Aufgaben erläutert.

Ping- und Pong-Frames

Wie eingangs erwähnt, dienen Ping- und Pong-Frames zur Verwaltung offener WebSocket-Verbindungen. Ping-Frames tragen den *opcode* 0x9, Pong-Frames hingegen den *opcode* 0xA. Das Übertragen von Nutzdaten bis zu 125 Byte durch einen Ping- oder Pong-Frame ist durch den RFC 6455 zwar vorgesehen, kommt aber typischerweise nicht zum Einsatz [GLN15, S. 48]. Enthält ein Ping-Frame Nutzdaten, so müssen diese von einem Pong-Frame übernommen und zurückgesendet werden. Empfängt ein Endpunkt einen Ping-Frame, so muss schnellstmöglich mit einem Pong-Frame darauf reagiert werden. Werden mehrere Ping-Frames empfangen, bevor ein Pong-Frame zurückgesendet wurde, so kann lediglich der zuletzt eingetroffene Ping-Frame beantwortet werden [FM11, S. 36]. Ping- und Pong-Frames dienen aktuell als rein interner Mechanismus und können nicht über eine Schnittstelle direkt versendet werden [GLN15, S. 49].

Bei den in Abbildung 2.3 dargestellten Frames handelt es sich um den Wireshark-Mitschnitt eines Heartbeats zwischen einem Client und dem Host unter <ws://echo.websocket.org>. Der Server sendet dabei einen Ping-Frame, um festzustellen, ob der Client noch erreichbar ist. Der Client antwortet mit einem korrespondierenden Pong-Frame. Da es sich um eine Nachricht von einem Client zu einem Server handelt, ist dieser Frame maskiert.

2. Das WebSocket-Protokoll

| | Ping-Frame | Pong-Frame |
|---------------|---|---|
| Byte | 89 00 | 8a 80 (85 e1 ef 27) |
| Binär | 1000 1001 | 1000 1010 (...) |
| Detail | FIN = 1, RSV n. gesetzt opcode = 0x9 | FIN = 1, RSV n. gesetzt opcode = 0xA |

Abbildung 2.3: Ping- und Pong-Frames zur Verbindungskontrolle

Close-Frames

Der Abbau einer WebSocket-Verbindung erfolgt durch den Versand eines Close-Frames. Der zugehörige *opcode* lautet 0x8. Empfängt ein Endpunkt einen Close-Frame, ohne bereits selbst einen solchen versendet zu haben, so muss er ebenfalls mit einem Close-Frame antworten. Wurde der initiale Close-Frame mit einem Status-Code versehen, so wird dieser in der Antwort übernommen [FM11, S. 35]. Wird ein Close-Frame empfangen, so können ausstehende Fragmente noch vor dessen Bestätigung gesendet werden. Allerdings kann dabei nicht garantiert werden, dass der Endpunkt, der den Verbindungsabbau eingeleitet hat, diese Nachrichten noch verarbeitet. Haben beide Endpunkte je einen Close-Frame gesendet und empfangen, so müssen sowohl die WebSocket- als auch die TCP-Verbindung getrennt werden. Der Server muss dies unverzüglich durchführen, wohingegen der Client auf den Server warten sollte [FM11, S. 36]. Sollte der Verbindungsabbau von beiden Endpunkten gleichzeitig eröffnet werden, so kann angenommen werden, dass die Verbindung geschlossen ist.

| | |
|---------------------------|---|
| Byte | 88 82 (2a 1f fc f2) 29 f7 |
| Binär | 10001000 10000010 (...) 00101001 11110111 |
| Detail | FIN = 1, RSV n. gesetzt |
| opcode | 0x8 |
| Maskierung | 1 |
| Payload length | 2 |
| Payload (maskiert) |)÷ |

Abbildung 2.4: Close-Frame initiiert von Client

| | |
|-----------------------------|-------------------------------------|
| Byte | 88 02 03 e8 |
| Binär | 10001000 00000010 00000011 11101000 |
| Detail | FIN = 1, RSV n. gesetzt |
| opcode | 0x8 |
| Maskierung | 0 |
| Payload length | 2 |
| Payload (unmaskiert) | 1000 |

Abbildung 2.5: Close-Frame-Antwort des Servers

2. Das WebSocket-Protokoll

Abbildung 2.4 zeigt einen maskierten Close-Frame, der von einem Client abgesetzt wurde. Der maskierte Payload enthält dabei den Status-Code *1000*. Dieser wurde in der Antwort des Servers ebenfalls zurückgemeldet. Hierbei handelt es sich um die Angabe *Normal Closure*, die besagt, dass die Verbindung ordnungsgemäß beendet wurde. Der RFC 6455 definiert vier Status-Gruppen mit folgender Bedeutung [FM11, S. 46]:

0-999 Dieser Bereich wird nicht genutzt.

1000-2999 Diese Status-Codes sind für die Definition innerhalb des RFC 6455 sowie für zukünftige Überarbeitungen und Erweiterungen reserviert.

3000-3999 Status-Codes in diesem Bereich sind für die Verwendung in Frameworks, Bibliotheken und Anwendungen gedacht und bedürfen einer Registrierung bei der Internet Assigned Numbers Authority (IANA).

4000-4999 In diesem Bereich können eigene Status-Codes definiert werden. Diese benötigen keine Registrierung und müssen den kommunizierenden Anwendungen bekannt sein.

Die Spezifikation des WebSocket-Protokolls definiert bereits einige Status-Codes innerhalb des zweiten Bereichs. Allerdings werden diese nur als Vorschlag angegeben, was inzwischen zu einer unkontrollierten Vielfalt und damit verbunden zu einigen Problemen führte [GLN15, S. 53]. Bisher wurden nur wenige Status-Codes neben jenen aus der Spezifikation registriert. Die bisher offiziellen Registrierungen finden sich auf den zugehörigen Seiten der IANA.² Tabelle 2.3 (S.15) zeigt einen kurzen Auszug der relevantesten Status-Codes. Es fällt auf, dass alle Neuregistrierungen durch Alexey Melnikov erfolgten, der bereits an der Spezifikation des Protokolls beteiligt war. In einer der betreffenden Anfragen zur Registrierung eines Status-Codes fragt Melnikov auch, ob diese Status-Codes genutzt werden und noch ein Interesse an deren Registrierung besteht [Mel12].

2.2. WebSocket-Kommunikationsphasen

Das WebSocket-Protokoll basiert auf dem Transmission Control Protocol und durchläuft wie dieses verschiedene Phasen der Kommunikation. Zu Beginn wird ähnlich zu TCP ein Opening-Handshake zum Verbindungsaufbau durchgeführt. Auch am Ende der Datenübertragung wird ein solcher Handshake eingesetzt, um die Verbindung ordnungsgemäß zu beenden. Die Ports, die vom WebSocket-Protokoll verwendet werden, sind Port 80 (ungesicherte Verbindungen) und Port 443 (gesicherte Verbindungen). Aktuell wird deren Verwendung jedoch vom HTTP(S)-Protokoll dominiert und sie müssen daher geteilt werden. Deshalb wird der Verbindungsaufbau als HTTP-Upgrade-Anfrage initiiert. Zukünftige Implementierungen könnten dies jedoch durch einen einfacheren Mechanismus über einen dedizierten Port ablösen, ohne das WebSocket-Protokoll umgestalten zu müssen [FM11, S. 3]. Im folgenden Abschnitt werden die notwendigen Schritte zum Verbindungsaufbau und -abbau erläutert sowie ein beispielhafter Ablauf der Übertragung über eine WebSocket-Verbindung gezeigt.

²unter <https://www.iana.org/assignments/websocket/>

2. Das WebSocket-Protokoll

| Status-Code | Bedeutung | Beschreibung | Spezifiziert durch |
|-------------|------------------|--|--------------------|
| 1000 | Normal Closure | Zeigt einen ordnungsgemäßen Verbindungsabbau an. | RFC 6455 |
| 1002 | Protocol Error | Der Endpunkt beendet die Verbindung aufgrund eines Protokollfehlers. Beispielsweise wird der Empfang einer nicht maskierten Nachricht von einem Server durch selbigen mit diesem Status-Code beendet. | RFC 6455 |
| 1006 | Abnormal Closure | Dieser Status darf nicht in einem Close-Frame verwendet werden. Anwendungen können durch den Status signalisieren, dass die Verbindung unerwartet und auf unbekannte Weise beendet wurde. | RFC 6455 |
| 1010 | Mandatory Ext. | Ein WebSocket-Client kann diesen Status-Code verwenden, wenn er die Verbindung aufgrund fehlender Erweiterungen beendet. Ein Server verwendet diesen Status-Code nicht, da er aufgrund mangelnder clientseitiger Erweiterungen den Opening-Handshake abbrechen kann. | RFC 6455 |
| 1012 | Service Restart | Gibt an, dass der Dienst neustartet. Verbundene Clients sollen im Falle einer Wiederherstellung der Verbindung eine zufällige Zeit zwischen 5-30 Sekunden warten [Mel12]. | Alexey Melnikov |
| 1013 | Try Again Later | Signalisiert, dass der Dienst überlastet ist. Sollte das Ziel über mehrere IP-Adressen verfügen, kann eine andere verwendet werden. Ist nur eine IP-Adresse verfügbar, so soll der Benutzer entscheiden, ob er sich erneut verbinden möchte [Mel12]. | Alexey Melnikov |

Tabelle 2.3: Auszug der registrierten Status-Codes

2.2.1. Verbindungsauflaufbau

Da Verbindungen über die Ports 80 bzw. 443 in den meisten Firewalls zugelassen werden, bedarf es keiner neuen Freigaben bei der Verwendung von WebSockets. Allerdings sind diese Ports oftmals durch einen HTTP-Server belegt. Aus diesem Grund bedarf es beim Aufbau einer WebSocket-Verbindung der Kooperation mit HTTP. Der Client sendet dazu einen HTTP-Upgrade-Request, der um spezielle Felder erweitert wurde.

Der Client stellt in Abbildung 2.6 einen solchen HTTP-Upgrade-Request an den Endpunkt des WebSocket-Servers unter `localhost:8080`. Durch `Connection: Upgrade` wird dem Server mitgeteilt, dass auf ein anderes Protokoll gewechselt werden soll, in diesem Fall `websocket`. Die Versionsnummer entspricht den unterschiedlichen IETF-Drafts. Über `Sec-WebSocket-Extensions` werden die Erweiterungen angefragt, die der Client unterstützt. Die übermittelte Erweiterung

2. Das WebSocket-Protokoll

```
GET /dako-backend/advancedchat HTTP/1.1
Host: localhost:8080
Connection: Upgrade
Upgrade: websocket
Origin: localhost:8080
Sec-WebSocket-Version: 13
Sec-WebSocket-Key: 1zUsRVYbBDvRKqqZq9cRsg==
Sec-WebSocket-Extensions: permessage-deflate; client_max_window_bits
(...)
```

Abbildung 2.6: Opening-Handshake des Clients

`permessage-deflate` wird in Kombination mit dem Parameter `client_max_window_bits` angegeben, der die *LZ77-Sliding-Window*-Größe limitiert, mit der der Client die Nachricht komprimiert. Bei dieser Methode handelt es sich um einen Algorithmus zur verlustfreien Kompression von Daten. So kann der zu reservierende Speicherbedarf bei der serverseitigen Dekompression reduziert werden [Yos15, S. 18]. Der Client generiert den Base64-codierten `Sec-WebSocket-Key`, der anschließend vom Server zur Empfangsbestätigung des Handshakes verwendet wird.

```
HTTP/1.1 101
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Accept: mi8r8dWsVBJS1B3908SLbnd5tHM=
Sec-WebSocket-Extensions: permessage-deflate; client_max_window_bits=15
(...)
```

Abbildung 2.7: Opening-Handshake des Servers

Damit der Server bestätigen kann, dass der Handshake eingetroffen ist, wird der übermittelte `Sec-WebSocket-Key` mit dem Globally Unique Identifier (GUID) 258EAFA5-E914-47DA-95CA-C5AB0DC85B11, der das WebSocket-Protokoll identifiziert, verbunden und ein SHA-1-Hashwert gebildet. Dieser Wert wird in einer Base64-Kodierung im Feld `Sec-WebSocket-Accept` übermittelt. Der Client kann beim Empfang der Nachricht anhand des gleichen Vorgehens einen SHA-1-Hash erzeugen und mit dem empfangenen vergleichen. Dieses Verfahren wird eingesetzt, um zu verhindern, dass der Server Verbindungen annimmt, die keine WebSocket-Verbindungen sind. Dadurch sollen Angreifer gehindert werden, über Formulareingaben oder XMLHttpRequests den WebSocket-Server zu manipulieren [FM11, S. 7].

1. **Kombinierter Wert** 1zUsRVYbBDvRKqqZq9cRsg==258EAFA5-E914-47DA-95CA-C5AB0DC85B11
2. **SHA-1-Hashwert** 9A2F2BF1D5AC541252D41DFDD3C48B6E7779B473
3. **Base64-Kodierung** mi8r8dWsVBJS1B3908SLbnd5tHM=

Des Weiteren teilt der Server dem Client mit, dass die Erweiterung `permessage-deflate` unterstützt wird. Die maximale Größe eines LZ77-Sliding-Windows beträgt demnach 2^{15} Byte =

2. Das WebSocket-Protokoll

32 KB. Als HTTP-Status wird der Code 101 `Switching Protocols` angegeben. Dieser wird eingesetzt, um zu signalisieren, dass ab sofort das WebSocket-Protokoll genutzt wird. Weitere Anwendung findet dieser Status-Code beim Wechsel von HTTP-Versionen [Lea+99]. Meldet der Server einen anderen Status, so wird angenommen, dass der WebSocket-Handshake nicht vollständig durchgeführt werden konnte und die Semantik von HTTP gilt [FM11, S. 7]. Ebenso wird verfahren, wenn der vom Client ermittelte Hashwert nicht mit dem vom Server übertragenen `Sec-WebSocket-Accept` übereinstimmt.

Optionale Felder können ebenfalls übertragen werden. Hierzu zählt beispielsweise das Header-Feld `Sec-WebSocket-Protocol`, das angibt, welches Sub-Protokoll für die Übertragung genutzt werden soll. Hat der Server den Opening-Handshake erfolgreich zurückgesendet, so gilt die WebSocket-Verbindung als geöffnet und Daten können gesendet werden.

2.2.2. Datentransfer

Befindet sich die WebSocket-Verbindung nach dem erfolgreichen Opening-Handshake im Status `OPEN`, so können Daten bidirektional gesendet werden. Wie in Abschnitt 2.1.1 erläutert, definiert das Header-Feld `opcode` eines WebSocket-Frames die übertragene Art von Nutzdaten. Aktuell sind lediglich Text- und Binärdaten offiziell spezifiziert.

| | |
|------------------|---|
| Byte | 81 0e [57 65 62 53 6f 63 6b 65 74 20 32 30 31 38] |
| Header | FIN = 1, RSV = 0, opcode = 0x1, Payload length = 14 |
| [Payload] | WebSocket 2018 |

Abbildung 2.8: Datentransfer von Server zu Client

In Abbildung 2.8 ist eine Nachricht mit dem Inhalt `WebSocket 2018` von einem Server an einen Client gesendet worden. Als `opcode` wurde `0x1` angegeben, der besagt, dass es sich bei der Art des Payloads um eine Textnachricht handelt. Die Größe der Nutzdaten beträgt 14 Byte und die Nachricht konnte ohne Fragmentierung vollständig übertragen werden.

2.2.3. Verbindungsabbau

Der Abbau einer WebSocket-Verbindung erfolgt anhand eines Closing-Handshakes. Er soll den FIN/ACK-Closing-Handshake von TCP ergänzen, da dieser insbesondere bei der Verwendung von Proxy-Servern oder Vermittlungsstellen nicht garantiert zwischen genau zwei Endpunkten abläuft [FM11, S. 9]. Laut RFC 6455 sollen so Szenarien vermieden werden, in denen Daten unnötigerweise verloren gehen.

Wird beispielsweise ein Proxy-Server verwendet, so verbindet sich ein Client mit diesem anstelle des eigentlichen Ziels. Der Proxy-Server stellt daraufhin selbst eine Verbindung zum Zielhost her und vermittelt die Pakete zwischen Client und Server. Schließt einer der Endpunkte die TCP-Verbindung, obwohl der andere Kommunikationspartner noch Daten überträgt, so kann es sein, dass die Verbindung zwischen dem Proxy und dem schließenden Endpunkt bereits abgebaut ist.

2. Das WebSocket-Protokoll

Die noch ausstehende Datenübertragung schlägt fehl. Über den zusätzlichen Closing-Handshake des WebSocket-Protokolls wird die zugrundeliegende TCP-Verbindung erst geschlossen, wenn die WebSocket-Sitzung bereits abgebaut ist.

Die Einleitung eines Verbindungsabbaus kann sowohl vom Client als auch vom Server erfolgen. Dazu wird ein Frame mit dem `opcode`-Wert 0x8 an den Kommunikationspartner versendet. Der versendende Endpunkt wechselt mit dem Absenden des Closing-Frames vom `OPEN`- in den `CLOSING`-Status. Ebenso gilt dies beim Empfang eines Closing-Frames, insofern selbst noch keiner versendet wurde. Danach eintreffende oder zu sendende Nachrichten müssen verworfen werden. Haben beide Endpunkte sowohl einen Closing-Frame gesendet als auch empfangen, so gilt die Verbindung als beendet (Status: `CLOSED`) und die darunterliegende TCP-Verbindung kann geschlossen werden. Dies sollte stets vom Server begonnen werden, damit er im Status `TIME_WAIT` verbleibt. Der Client kann so die Verbindung durch das Senden eines neuen `SYN` unverzüglich wieder eröffnen [FM11, S. 41]. Die Zustände, die ein WebSocket annehmen kann, sowie deren Übergänge sind detailliert in der Abbildung auf Seite 23 dargestellt.

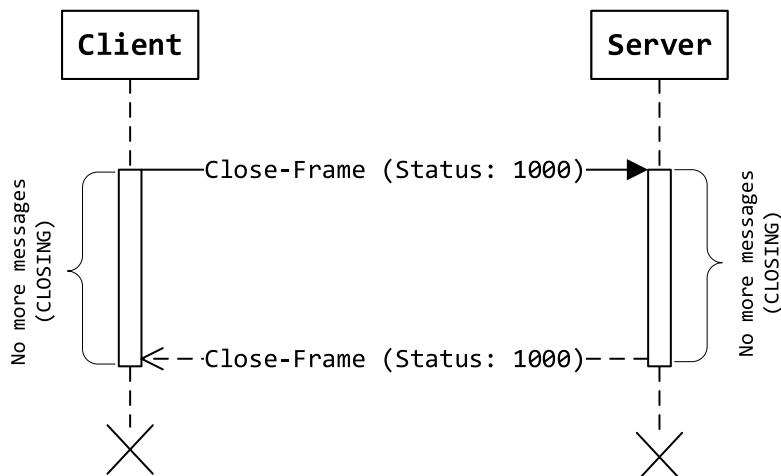


Abbildung 2.9: Clientseitiger Closing-Handshake

Abbildung 2.9 zeigt den schematischen Ablauf des Closing-Handshakes. Der Client initiiert diesen, indem er einen Closing-Frame mit dem Statuscode 1000 versendet. Der Server reagiert mit einer entsprechenden Antwort, wobei der Status aus dem ersten Frame übernommen wurde. Nachdem der Closing-Handshake abgeschlossen wurde, befinden sich beide Endpunkte im Status `CLOSED` und die zugrundeliegende TCP-Verbindung kann beendet werden.

2.3. Alternative Verfahren

Die Kommunikation mittels HTTP basiert auf einem Request/Response-Verfahren. Dabei sendet der Client immer eine Anfrage an einen Server. Dieser antwortet mit einer Statuszeile und entsprechendem Statuscode. Anschließend gilt die Übertragung als beendet und kann durch eine erneute

2. Das WebSocket-Protokoll

Anfrage des Clients neu gestartet werden. Eine Möglichkeit, Daten vom Server an den Client zu senden, ist in der Spezifikation von HTTP nicht vorgesehen. Verändert sich der Zustand einer serverseitigen Ressource, so kann der Client darüber bei einer reinen HTTP-Verbindung nur durch erneutes Anfragen informiert werden. Allerdings benötigen viele Anwendungsfälle diese Funktion. Aus diesem Grund wurde eine Vielzahl von Verfahren entwickelt, die diese Mechanismen abbilden sollen. Im Folgenden werden einige dieser Verfahren kurz vorgestellt und hinsichtlich ihrer Vor- und Nachteile gegenüber WebSockets analysiert.

2.3.1. Polling

Durch die vom W3C standardisierte API XMLHttpRequest (XHR) wurde im Jahr 2012 eine Möglichkeit geschaffen, mittels JavaScript HTTP-Requests zu konstruieren [WHA06]. Ein XHR-Objekt registriert verschiedene Methoden, die bei einer Änderung des Objekts asynchron aufgerufen werden. Dadurch können Anfragen gesendet werden, ohne den weiteren Programmablauf zu blockieren. Dies bildet die technische Grundlage vieler moderner Web-Frontends, die unter dem Oberbegriff Ajax (*Asynchronous JavaScript and XML*) zusammengefasst werden [GLN15, S. 26].

Beim Polling-Verfahren wird eine Anfrage an den Server in festgelegten Zeitabständen wiederholt. Durch die Verwendung eines XHR-Objekts blockiert die Anwendung zwischen dem Absenden und Empfang der Serverantwort nicht. Liegt keine Änderung vor, so sendet der Server eine leere Antwort zurück.

Dieses Verfahren wird von allen Browsern unterstützt und bedarf keiner besonderen Kenntnisse. Allerdings ist der erzeugte Overhead je nach Einsatzszenario nicht zu unterschätzen. Auch die Belastung des Servers bei der Verarbeitung der eintreffenden Anfragen kann problematisch sein. Treten demnach viele Änderungen in kurzen Zeitabständen auf, so ist von der Verwendung eines einfachen Polling-Verfahrens abzuraten.

2.3.2. Long-Polling

Die regelmäßigen Abfragen des einfachen Pollings erzeugen einen starken Overhead. Um diesen zu reduzieren, wird beim Long-Polling nur eine Antwort vom Server zurückgesendet, wenn eine Änderung vorliegt. Der Client reagiert umgehend mit einem neuen Request auf die Antwort des Servers. Die Verbindung bleibt dabei geöffnet, solange keine Änderung auftritt. Läuft die vorgefnierte Wartezeit ab, so antwortet der Server auch hier mit einer leeren Nachricht [GLN15, S. 29].

Das Long-Polling reduziert den Overhead gegenüber dem einfachen Polling-Mechanismus deutlich. Aufgrund des sofortigen Wiederaufbaus durch den Client wird eine nahezu andauernde Verbindung simuliert. Dennoch ist auch in diesem Fall der Overhead deutlich größer als bei der Verwendung einer WebSocket-Verbindung. Mit steigender Frequenz von Nachrichten nähert sich dieser dem Overhead des einfachen Pollings.

2.3.3. Comet

Aufgrund diverser Limitierungen für Mehrbenutzeranwendungen entwickelten sich Programmiermodelle, die unter der Bezeichnung *Comet* [Rus06] zusammengefasst wurden. Da es sich hierbei nicht um einen spezifizierten Standard handelt, wurde im Rahmen des CometD-Projekts der Dojo Foundation das Bayeux-Protokoll ins Leben gerufen. Ziel des Projekts ist die Reduzierung der Komplexität bei der Implementierung von Comet-Anwendungen [Rus+07]. Um dies zu erreichen wurden Referenzimplementierungen des Bayeux-Protokolls in verschiedenen Programmiersprachen bereitgestellt.

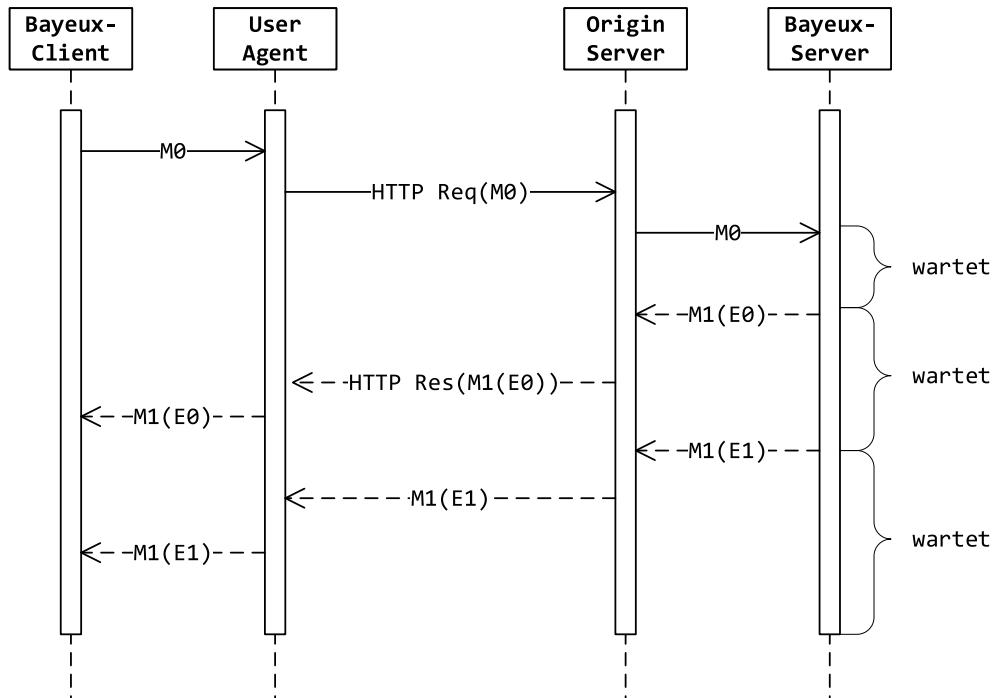


Abbildung 2.10: HTTP-Streaming im Bayeux-Protokoll

Neben der Verwendung des Long-Polling-Mechanismus zur ereignisgesteuerten Nachrichtenübertragung von Server zu Client sieht das Bayeux-Protokoll auch Streaming-Techniken vor. Dabei wird die HTTP-Verbindung nach der Antwort des Servers auf den Request des Clients nicht geschlossen, sondern anhand des Connection-Headers `keep-alive` weiterhin offen gehalten. Somit können weitere Nachrichten vom Server an den Client gesendet werden, ohne dass eine erneute Anfrage durch den Client gestellt werden muss. Abbildung 2.10 illustriert die Übertragung einer Nachricht M0 von einem Client, der in einem Webbrowser (**User Agent**) ausgeführt wird, zu einem Webserver (**Origin Server**), auf dem die serverseitige Implementierung des Bayeux-Protokolls ausgeführt wird. Die Übertragung erfolgt als HTTP-Request und die korrespondierende Antwort des Servers als HTTP-Response. Die Nachricht des Servers enthält dabei etwaige Bayeux-Events (**E**). Nach der Übermittlung von M1(E0) wird die Verbindung nicht geschlossen, sodass der Server weiterhin Events an den Client übertragen kann.

2. Das WebSocket-Protokoll

Allerdings muss die Verwendung von HTTP-Streaming von den User-Agenten unterstützt werden, da unvollständige HTTP-Responses für dieses Verfahren genutzt werden [Rus+07]. Um bidirektional kommunizieren zu können, verwendet das Bayeux-Protokoll zwei HTTP-Verbindungen, wohingegen das WebSocket-Protokoll mit einer einzigen Verbindung arbeitet. Der größte Nachteil des Bayeux-Protokolls ist jedoch die fehlende Standardisierung durch eine etablierte Organisation.

2.3.4. Server-Sent Events

Die bislang vorgestellten Verfahren bieten keine standardisierte API, die es Servern ermöglicht, Daten über HTTP an eine Website zu senden. In HTML5 wurden dafür vom W3C die sogenannten *Server-Sent Events (SSE)* eingeführt, die ein `EventSource`-Objekt bereitstellen. Dieses sorgt auf der Clientseite dafür, dass Benachrichtigungen des Servers empfangen werden können. Dazu stellt der Browser drei Event-Handler bereit, die DOM-Events an die Anwendung weiterleiten [GLN15, S. 31]. Bei der Öffnung einer SSE-Verbindung wird der Event-Handler `onopen` aufgerufen, im Falle eines Fehlers der `onerror`-Handler. Kommen Nachrichten des Servers an, so wird der `onmessage`-Handler aktiviert [Hic09a]. Zu jedem Event-Handler werden entsprechende Methoden registriert, die die Bearbeitung des jeweiligen Events übernehmen.

Server-Sent Events erzeugen keinen Overhead durch wiederholte Anfragen an den Server. Die Kommunikation von Server zu Client wird ermöglicht, ohne mehrere HTTP-Verbindungen zu öffnen. Allerdings handelt es sich hier um einen unidirektionalen Kanal von Server zu Client. Dies ist für Anwendungsfälle ausreichend, in denen lediglich der Server Informationen zu übermitteln hat. Beispiele hierfür wären u. a. das Hardware-Monitoring eines Servers oder ein Newsticker. Sollen jedoch Daten in beide Richtungen übertragen werden, so ist die Verwendung von WebSockets die bessere Alternative. Insbesondere Mehrbenutzeranwendungen wie Spiele oder Chats, die auf kurze Latenzen angewiesen sind, profitieren von deren bidirektionalem Kommunikationskanal.

Bei aller Vielfalt der Alternativen besteht jedoch ein Bedarf nach einer standardisierten Methode, mit der Daten zwischen Client und Server bidirektional übertragen werden können. Dieser Bedarf führte zur Entwicklung der WebSockets, die als Einzige auch für die Übertragung binärer Daten geeignet sind.

3. Die WebSocket-API

Neben der Protokollspezifikation durch das Standardisierungsgremium IETF wurde auch eine passende API für die Verwendung von WebSockets definiert. Da es sich hierbei um eine Webtechnologie handelt, die in Browsern Anwendung findet, obliegt deren Spezifikation dem W3C. Der erste Working Draft vom 23. April 2009 wurde von Ian Hickson eingereicht, einem der führenden Entwickler der Protokollspezifikation [Hic09b]. Darin werden alle benötigten Eigenschaften und Funktionen der WebSocket-Schnittstelle definiert. Die Beschreibung erfolgt anhand der *Web Interface Description Language* (Web IDL), einer formalen Darstellung, die an die sprachlichen Konstrukte von ECMAScript bzw. JavaScript angelehnt ist. Dabei handelt es sich um eine Programmiersprache, die 1995 von Netscape Communications entwickelt und durch die European Computer Manufacturers Association (ECMA) unter der Bezeichnung ECMAScript standardisiert wurde [Pey17]. Alle Beispiele beziehen sich auf die aktuelle Version 8, die im Juni 2017 offiziell freigegeben wurde. Im Folgenden wird die geläufigere Bezeichnung JavaScript verwendet.

3.1. Elemente der W3C-WebSocket-API

Das Erstellen eines WebSocket-Endpunkts erfolgt laut Spezifikation anhand des Konstruktors. Dieser empfängt die URL, mit der der Endpunkt sich verbinden soll und legt diese im **read-only**-Attribut `url` ab. Optional empfängt der Konstruktor einen String oder ein String-Array, das die unterstützten Sub-Protokolle enthält.

```
var ws = new WebSocket(in DOMString url, in optional DOMString[] protocols);
```

Listing 3.1: WebSocket anlegen und verbinden

Wird die URL nicht im korrekten Schema angegeben, so wird eine `DOMException` mit der Beschreibung `SyntaxError` ausgelöst. Falls Sub-Protokolle angegeben werden, so sendet der Client diese bei der Anfrage im HTTP-Headerfeld `Sec-WebSocket-Protocol`. Wenn Server und Client ein gemeinsames Protokoll vereinbaren konnten, wird es im schreibgeschützten Attribut `protocol` abgelegt. Etwaige Erweiterungen werden nach dem erfolgreichen Verbindungsaufbau im Attribut `extensions` abgelegt.

3.1.1. WebSocket Lebenszyklus

Jede WebSocket-Verbindung unterliegt einem Lebenszyklus. Die Spezifikation unterscheidet hier vier mögliche Zustände, die als konstante Zahlenwerte hinterlegt sind. Je nachdem, in welchem Zustand sich der WebSocket gerade befindet, wird der Wert einer dieser Konstanten im Attribut `readyState` abgelegt [WHA10].

3. Die WebSocket-API

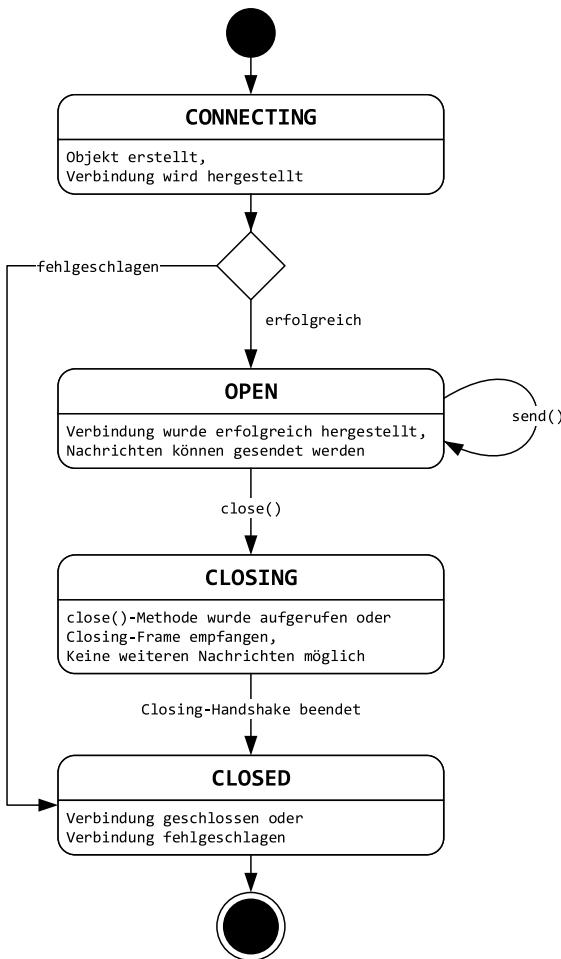


Abbildung 3.1: UML-Zustandsdiagramm des WebSocket-Lifecycle

Die Implementierung in JavaScript spiegelt den Lebenszyklus eines WebSocket-Objekts gemäß der Beschreibung des W3C wider. So wird beim Anlegen des Objekts die Verbindung aufgebaut. Konnte diese erfolgreich hergestellt werden, ist anschließend das Senden von Nachrichten möglich. Durch den Aufruf der `close()`-Methode wird der Verbindungsauflauf eingeleitet. Das Schließen kann zudem optional mit Angabe eines Status-Codes (Tabelle 2.3, S.15) und einer textuellen Begründung erfolgen.

```

var ws = new WebSocket("ws://echo.websocket.org"); // 0 = CONNECTING
console.log(ws.readyState); // 1 = OPEN, wenn erfolgreich
ws.send("Message"); // 1
ws.close(1000, "Work done."); // 2 = CLOSING
console.log(ws.readyState); // 3 = CLOSED, wenn Verbindung abgebaut
    
```

Listing 3.2: WebSocket-Lebenszyklus

3. Die WebSocket-API

3.1.2. Event-Handling

Die Zustände eines WebSockets werden durch Events beeinflusst, die während der Kommunikation mit einem Endpunkt eintreffen. Diese werden von Event-Handlern empfangen und können anschließend verarbeitet werden. Die WebSocket-API definiert die vier Event-Handler `onopen`, `onerror`, `onmessage` und `onclose`. Ein erfolgreicher Verbindungsaufbau löst das Event `open` aus, das vom Event-Handler `onopen` verarbeitet wird. Fehler werden vom `onerror`-Event-Handler signalisiert. Wurde die Verbindung geschlossen oder konnte nicht geöffnet werden, so wird der `onclose`-Event-Handler ausgelöst.

```
var ws = new WebSocket("ws://echo.websocket.org");
// Registrieren des OnOpen-Handlers
ws.onopen = function(){
    console.log("Verbindung hergestellt!");
}
// Registrieren des OnError-Handlers
ws.onerror = function(e){
    console.log("Fehler aufgetreten: " + e.data);
}
// Registrieren des OnMessage-Handlers
ws.onmessage = function(e){
    showMessageInUI(e.data);
}
// Registrieren des OnClose-Handlers
ws.onclose = function(e){
    console.log("CloseEvent erhalten - Verbindung wird abgebaut")
    console.log("Code: " + e.code);
    console.log("Grund: " + e.reason);
}
```

Listing 3.3: WebSocket-Event-Handler

3.1.3. Kommunikation

Empfangene Nachrichten können mittels des `onmessage`-Event-Handlers an die Anwendung weitergegeben werden. Entsprechend der Spezifikation des RFC 6455 ermöglicht die WebSocket-API das Senden von textuellen und binären Daten. Zur Unterscheidung verschiedener Arten binärer Daten stellt sie die Enumeration `BinaryType` bereit. Diese enthält die Werte `blob` und `ArrayBuffer`. Die Übermittlung der Daten erfolgt durch die Methode `send()`. Bei deren Aufruf wird das Attribut `bufferedAmount` um die Anzahl Byte erhöht, die nicht in das Netzwerk übermittelt werden konnten [WHA10].

```
var ws = new WebSocket("ws://echo.websocket.org");
ws.send("Nachricht"); // als DOMString versenden
ws.send(BlobUtils.fromString("Nachricht")) // als binary blob versenden
```

Listing 3.4: WebSocket-Nachrichten senden

3.2. Die Java™ API for WebSocket (JSR-356)

Die WebSocket-APIs beschränken sich nicht auf JavaScript. Verschiedene Implementierungen des Standards ermöglichen dem Entwickler, WebSockets in seiner bevorzugten Programmiersprache zu nutzen. Laut dem PopularitY of Programming Language (PYPL) Index ist Java der Oracle Corporation die weltweit beliebteste Programmiersprache [Cab17]. Sie wurde 1990 von James Gosling mit seinem Team bei dem Unternehmen Sun Computers entwickelt. Die erste öffentlich zugängliche Version wurde 1995 publiziert [Smy09]. Aktuell ist Java 9 verfügbar.

Java Community Process

Soll eine neue Technologie in den Standardumfang von Java aufgenommen werden, so muss diese den Java Community Process (JCP) durchlaufen [Ora17b]:

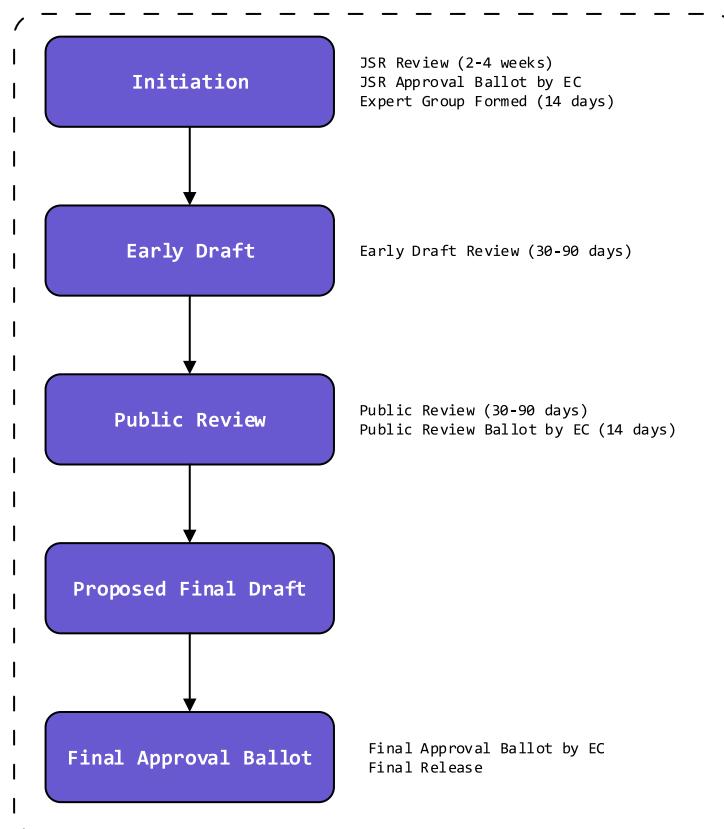


Abbildung 3.2: JSR Approval Timeline

In Phase 1 können Mitglieder des JCP einen Vorschlag zur Aufnahme einer Erweiterung machen. Diese Vorschläge werden als *Java Specification Request* (JSR) bezeichnet. Wird der JSR vom *Executive Committee* (EC) angenommen, so wird bis Phase 2 eine Expertengruppe gebildet, die einen Early Draft erstellt. Darauf aufbauend entwickelt diese Gruppe einen Public Draft, der in Phase 3 öffentlich eingesehen und kommentiert werden kann. Der Entwurf wird dann entsprechend überarbeitet und eine finale Version eingereicht. Zudem werden eine Referenzimplementierung

3. Die WebSocket-API

(RI) und das Technology Compatibility Kit (TCK) erstellt, das Tests und Tools bereitstellt, um Implementierungen auf Konformität mit dem JSR zu prüfen. Abschließend stimmt das EC über die Aufnahme in die Standardbibliothek ab. Befindet sich der JSR dann im produktiven Betrieb, können in der Maintenance-Phase Updates und Änderungen des JSR beantragt und eingepflegt werden [Ora17b].

Der JSR 356 trägt die Bezeichnung Java™ API for WebSocket und regelt die Aufnahme der WebSocket-Technologie in die Standardbibliothek der Java Enterprise Edition (JEE). Er befindet sich aktuell in der vierten Phase und ist seit dem 22. Mai 2013 offizieller Bestandteil von JEE. Die folgenden Beispiele verwenden die achte Version von Java.

3.2.1. Grundlagen

Alle Klassen und Interfaces zur Implementierung von WebSocket-Clients und -Servern wurden für Java in dem Paket `javax.websocket` zusammengefasst. Für einen WebSocket-Client, der weder im Browser noch in einem anderen Framework zum Einsatz kommt, kann in Java die Open-Source-Referenzimplementierung *Tyrus* von Oracle verwendet werden. Nachfolgende Beispiele nutzen das aktuellste Release 1.12. Die Bereitstellungen erfolgen über Maven:

```
<dependency>
    <groupId>javax.websocket</groupId>
    <artifactId>javax.websocket-api</artifactId>
    <version>1.1</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>org.glassfish.tyrus.bundles</groupId>
    <artifactId>tyrus-standalone-client</artifactId>
    <version>1.12</version>
</dependency>
```

Für die Implementierung eines WebSocket-Serverendpunkts wird zudem ein Webserver benötigt, der die WebSocket-API implementiert. Für Java gibt es hier eine Vielzahl von Alternativen zur Auswahl:

- Apache Tomcat ab Version 7.0.56 [The14]
- GlassFish Server ab Version 4.1 [Ora14a]
- Grizzly ab Version 2 [Ora17a]
- Eclipse Jetty ab Version 7 [The16]
- Netty ab Version 4.0 [Lee11]

3. Die WebSocket-API

Ansätze der Programmierung

Die JavaTM API for WebSocket kann auf zwei unterschiedliche Arten verwendet werden. Der programmatische Ansatz sieht die Vererbung der `Endpoint`-Klasse der API vor. Dabei müssen u. a. die Methoden der Oberklasse überschrieben und Funktionen auf Events registriert werden. [Cow14, S. 9]:

```
public class MyWebSocketClient extends Endpoint {  
    @Override  
    public void onOpen(Session session) {  
        session.addMessageHandler(new MessageHandler.Whole<String>() {  
            @Override  
            public void onMessage(String message) {  
                // Verarbeitung der Nachricht  
            }  
        });  
    }  
}
```

Listing 3.5: Java: programmatischer Ansatz

Alternativ können einfache Java-Objekte durch den Einsatz von Annotationen zu WebSocket-Endpunkten transformiert werden. Handelt es sich um eine Annotation auf Klassenebene, so können diverse Konfigurationsparameter angegeben werden. Alle Informationen sind somit an einer Stelle gesammelt und auf eine Konfigurationsklasse kann in vielen Fällen verzichtet werden.

```
@ServerEndpoint( value = "/endpoint", subprotocols = {"chat"})  
public class MyWebSocketServer { ... }
```

Listing 3.6: Java: Annotationen

Die nachfolgenden Code-Beispiele sind immer annotations-basiert, aufgrund besserer Lesbarkeit, verkürzter Schreibweise und der einfachen Verständlichkeit.

Erzeugung eines WebSockets

Clientseitige Endpunkte werden mit der Annotation `@ClientEndpoint` versehen. Die Konstruktion eines WebSockets erfolgt hier durch die Erzeugung eines Container-Objekts, das anschließend zum Verbindungsauflaufbau verwendet wird. Die Erstellung des Objekts ist nur dann erfolgreich, wenn die WebSocket-Verbindung aufgebaut werden konnte.

```
@ClientEndpoint  
public class MyClient {  
    public MyClient(URI hostPath){  
        // Container-Objekt erstellen  
        WebSocketContainer wsc = ContainerProvider.getWebSocketContainer();  
        // Verbindung fuer Endpunkt MyClient herstellen  
        wsc.connectToServer(this, hostPath);  
    } }
```

Listing 3.7: Java Client: Verbindungsauflaufbau

3. Die WebSocket-API

Ein WebSocket-Server wird mit der Annotation `@ServerEndpoint` gekennzeichnet. Sowohl bei Server- als auch bei Clientendpunkten können verschiedene Konfigurationsparameter verwendet werden (siehe Listing 3.6). Der Parameter `value` gibt dabei den relativen Pfad an, unter dem der WebSocket-Endpunkt erreichbar ist. Wird die Anwendung auf einem lokalen Webserver ausgeführt und ihre Bezeichnung lautet `example`, so ist die Anwendung erreichbar unter `ws://localhost[:port]/example/endpoint`. Unterstützte Subprotokolle können optional mittels des Parameters `subprotocols` angegeben werden. Falls die Anwendung eigene Java-Objekte oder komplexere Datentypen übertragen soll, so müssen Encoder- bzw. Decoder-Klassen angegeben werden. Diese werden im Abschnitt 3.2.3 näher erläutert.

Event-Handling

Bei der Verwendung von Annotationen erfolgt in Java keine manuelle Deklaration der Handler. Eine Methode, die ein bestimmtes Event bearbeitet, wird durch Angabe der jeweiligen Annotation zu einem Event-Handler transformiert.

```
@OnOpen
public void init(Session session, EndpointConfig config){
    // Session und Konfiguration in Variable speichern
}

@OnError
public void error(Throwable t) {
    System.out.println("Fehler aufgetreten: " + t.getMessage());
}

@OnClose
public void close(CloseReason c) {
    System.out.println("Die Verbindung wurde geschlossen.");
    System.out.println("Code: " + c.getCloseCode());
    System.out.println("Grund: " + c.getReasonPhrase());
}

@OnMessage
public void incoming(String msg){
    System.out.println("Neue Nachricht empfangen: " + msg);
}
```

Listing 3.8: Java: Event-Handler

Die JavaTM API for WebSocket sieht dafür die Annotationen `@OnOpen`, `@OnError`, `@OnMessage` und `@OnClose` vor. Jede auf diese Weise deklarierte Methode ermöglicht die Angabe einiger Parameter. Die aktuell verwendete `Session` kann dabei immer empfangen werden.

3.2.2. Kommunikation

Der RFC 6455 definiert Ping- und Pong-Frames, durch die beispielsweise geprüft werden kann, ob ein Endpunkt noch erreichbar ist oder um eine inaktive Verbindung aufrechtzuerhalten. Allerdings spezifiziert die W3C-WebSocket-API keine Möglichkeit, diese Art von Control-Frames zu versenden. Die Java-API hingegen stellt dafür die Methoden `sendPing()` und `sendPong()` zur Verfügung, die ein `ByteBuffer`-Objekt als Parameter empfangen, über das bis zu 125 Byte Nutzdaten angehängt werden können. Diese Limitierung wird aufgrund der im RFC 6455 definierten Größe von Control-Frames durch eine `IllegalArgumentException` sichergestellt.

```
public void sendPing(ByteBuffer applicationData) throws IOException,
    IllegalArgumentException

public void sendPong(ByteBuffer applicationData) throws IOException,
    IllegalArgumentException

@OnMessage
public void catchPong(PongMessage pong, Session session){ ... }
```

Listing 3.9: Java: Ping-/Pong-Nachrichten

Die Antwort auf eine eingehende Ping-Nachricht muss nicht durch den Entwickler erfolgen. Die Implementierungen der Java-WebSocket-API stellen sicher, dass eine korrespondierende Pong-Nachricht so bald wie möglich zurückgesendet wird. In Listing 3.9 ist die Methode `catchPong` definiert, die als Parameter ein `PongMessage`-Objekt empfängt. Durch sie ist es möglich, die Antwort auf eine zuvor gesendete Ping-Nachricht weiterzuverarbeiten. Beispielsweise können so Messungen der *Round Trip Time* (RTT) durchgeführt werden. Allerdings ist deren Aussagekraft nur bedingt gegeben, da diese bei größeren Nachrichten stark abweichen können [Cow14, S. 116]. Das Senden einer Pong-Nachricht ohne vorangegangenen Ping kann genutzt werden, um einen unidirektionalen *Heartbeat* abzubilden. Ein Endpunkt antwortet spezifikationsgemäß nicht auf eine eintreffende Pong-Nachricht, es sei denn, eine entsprechende Methode wurde dafür bereitgestellt.

Das Senden und Empfangen sowohl textueller als auch binärer Daten ist ebenfalls in der Java-WebSocket-API definiert. Die grundlegendsten Datentypen, die dabei eingesetzt werden können, sind `String`, `ByteBuffer` und `byte[]` [Cow14, S. 30]. Darüber hinaus ist es möglich, komplexere Datenstrukturen oder eigene Objekte zu versenden. Dies setzt jedoch entsprechende Decoder- und Encoder-Klassen voraus. Liegen Daten zum Zeitpunkt des Versendens noch nicht vollständig vor oder können aufgrund ihrer Größe nicht in einem einzigen Frame übertragen werden, so kann auf Methoden zurückgegriffen werden, die das partielle Übertragen von Nachrichten ermöglichen. Aufgrund der protokolleigenen Fragmentierung können diese Teilnachrichten von den tatsächlich übertragenen Fragmenten abweichen. Wird der letzte Teil einer Nachricht übertragen, so muss das `isLast`-Flag der jeweiligen `send`-Methode gesetzt werden. Das Versenden erfolgt immer über die `RemoteEndpoint`-Schnittstelle, die man durch das aktuelle `Session`-Objekt erhält.

3. Die WebSocket-API

```
public void sendMessages(Session session){  
  
    RemoteEndpoint.Basic rEndpoint = session.getBasicRemote();  
  
    // Textnachricht als String  
    rEndpoint.sendText("Message");  
  
    // Binaere Daten senden  
    rEndpoint.sendBinary(ByteBuffer.wrap("Message".getBytes()));  
  
    // Partielles Senden einer Textnachricht  
    String part1 = "Mess";  
    String part2 = "age";  
    boolean isDone = false;  
    rEndpoint.sendText(part1, isDone);  
    rEndpoint.sendText(part2, !isDone);  
}
```

Listing 3.10: Java: synchrone Nachrichtenübertragung

Die bisher vorgestellten Methoden zum Nachrichtenversand blockieren, bis der gesamte Payload übertragen wurde. Soll die Applikation hingegen weitere Aufgaben während des Sendens erledigen können, so muss auf die Funktionen der Schnittstelle `RemoteEndpoint.Async` zurückgegriffen werden. Beispiele für den Einsatz asynchroner Verarbeitung finden sich häufig. So soll eine grafische Oberfläche beim Versand einer umfangreichen Nachricht weiterhin reagieren können. Um Nachrichten asynchron zu übertragen, stellt die API zwei Varianten zur Verfügung.

```
// Liefert Future-Objekt zurück  
public Future<Void> sendText(String text);  
  
// Benötigt SendHandler  
public void sendText(String text, SendHandler handler);
```

Listing 3.11: Java: Future und Handler

Der erste Weg besteht aus dem Senden einer WebSocket-Nachricht *by future* [Cow14, S. 119]. Der Zweck eines `Future`-Objekts ist die Nachverfolgung des Versendungsstatus durch den Aufruf der zugehörigen Methoden. So kann mittels `isDone()` geprüft werden, ob der Sendevorgang abgeschlossen wurde. Die Methode `cancel()` bricht eine ausstehende Übertragung ab. Im Unterschied zur synchronen Nachrichtenübertragung werden etwaige Ausnahmen oder Fehler nicht automatisch zurückgemeldet. Diese können nur durch `Future.get()` ermittelt werden. Diese Methode blockiert die weitere Ausführung, bis die Nachricht übermittelt wurde oder ein Fehler aufgetreten ist.

Die zweite Möglichkeit, Nachrichten asynchron zu übermitteln, erfordert die Bereitstellung eines `callback`-Objekts. Dabei handelt es sich um eine Implementierung der `SendHandler`-Schnittstelle,

3. Die WebSocket-API

die beim Eintreffen eines Ereignisses ausgelöst wird. Man spricht vom Senden einer Nachricht *with handler*.

```
public class MyHandler implements SendHandler {  
    @Override  
    public void onResult(SendResult arg0) {  
        // Verarbeiten des Ergebnisses  
    }  
}
```

Listing 3.12: Java: Callback-Interface SendHandler

Beide Varianten der asynchronen Nachrichtenübermittlung können sowohl textuelle als auch binäre Daten übertragen. Für das Senden komplexer Strukturen oder eigener Objekte wird auch hier ein Encoder benötigt. Ping- bzw. Pong-Nachrichten können nicht asynchron übertragen werden. Die Notwendigkeit asynchroner Übertragung ist in diesem Fall auch nicht gegeben, da diese Control-Frames maximal 125 Byte Nutzdaten enthalten können und dementsprechend selbst bei geringen Bandbreiten unverzüglich übermittelt werden [Cow14, S. 120]. Die Entscheidung für eine der beiden Varianten der asynchronen Nachrichtenübertragung hängt von verschiedenen Faktoren ab. Soll nur eine kleine Anzahl anderer Aufgaben während des Sendevorgangs erledigt werden, so kann die Nachricht *by future* versendet, anschließend andere Aufgaben gestartet und nach deren Beendigung durch den Aufruf der `get()`-Methode auf die Sendebestätigung gewartet werden. Dieser Ansatz blockiert dadurch einen laufenden Thread. Folgen jedoch weitere Aufgaben nach dem Versand der Nachricht, deren Bearbeitung Vorrang hat, so sollten `Handler` eingesetzt werden. Die Benachrichtigung einer eintreffenden Sendungsbestätigung muss hierbei durch einen eigenen Thread der zugrundeliegenden WebSocket-Implementierung erfolgen [Cow14, S. 123]. So mit blockiert dieser Ansatz den laufenden Thread des eigenen Programms nicht. Zu beachten ist weiterhin, dass lediglich durch die Verwendung des `Future`-Ansatzes das Senden einer Nachricht abgebrochen werden kann.

3.2.3. En- und Decoding

Die W3C-WebSocket-API definiert lediglich textuelle und binäre Datentypen zur Übermittlung von Nutzdaten und entspricht damit den Anforderungen, die im RFC 6455 für das WebSocket-Protokoll festgelegt wurden. Die Java-WebSocket-API stellt hierfür standardisierte Methoden bereit, die das Senden eines `String`-Objekts oder binärer Datentypen wie `ByteBuffer` bzw. `byte[]` ermöglichen. In vielen Fällen ist es jedoch notwendig, komplexere Datentypen oder eigene Objekte übertragen zu können. Dafür wurde die `send`-Methode der `RemoteEndpoint`-Schnittstelle überladen, sodass sie einen Parameter vom Typ `Object` empfängt. Mit Hilfe dieser Methode lassen sich demnach Objekte eines beliebigen Datentyps versenden. Da das WebSocket-Protokoll lediglich textuelle oder binäre Daten kennt, müssen alle komplexeren Objekte in eine dieser beiden Formen gebracht werden. Zu diesem Zweck wurden die Schnittstellen `Encoder` und `Decoder` im Paket `javax.websocket` bereitgestellt. Über untergeordnete Schnittstellen kann zwischen der Umwandlung in textuelle oder binäre Daten gewählt werden.

3. Die WebSocket-API

```
public interface Encoder {  
    // Aufruf bei Verbindungsaufbau  
    void init(EndpointConfig config);  
    // Aufruf bei Verbindungsabbau  
    void destroy();  
    // Sub-Interfaces mit jeweiliger encode-Methode  
    interface Text<T> extends Encoder { String encode(T object) throws  
        EncodeException; }  
    interface Binary<T> extends Encoder { ByteBuffer encode(T object) throws  
        EncodeException; }  
}
```

Listing 3.13: Java: Encoder-Interface

Die textuelle Enkodierung eines Objekts mit der Klassenbezeichnung `MyObject` erfolgt demnach über eine Klasse, die das `Encoder.Text<MyObject>`-Interface implementiert. Sowohl die Enkodierung als auch die Dekodierung erfolgen automatisch und müssen nicht vom Entwickler ausgelöst werden. Die Enkodierung von `MyObject` erfolgt beim Aufruf der `send`-Methode.

```
public interface Decoder {  
  
    // Aufruf bei Verbindungsaufbau  
    void init(EndpointConfig config);  
  
    // Aufruf bei Verbindungsabbau  
    void destroy();  
  
    // Sub-Interfaces mit jeweiliger decode- und willDecode-Methode  
    interface Binary<T> extends Decoder {  
        T decode(ByteBuffer bytes) throws DecodeException;  
        boolean willDecode(ByteBuffer bytes);  
    }  
  
    interface Text<T> extends Decoder {  
        T decode(String s) throws DecodeException;  
        boolean willDecode(String s);  
    }  
}
```

Listing 3.14: Java: Decoder-Interface

Ein Endpunkt, der nur über eine `@OnMessage`-Methode verfügt, die als Parameter ein Objekt vom Typ `MyObject` empfängt, muss beim Erhalt einer textuellen Nachricht diese dekodieren. Dies erfolgt über eine Klasse, die das `Decoder.Text<MyObject>`-Interface implementiert hat. Dazu ist es zunächst notwendig zu überprüfen, ob die Dekodierung möglich ist. Dies wird über die Methode `willDecode` sichergestellt. Anschließend wird das erwartete Objekt aus dem empfangenen Text dekodiert.

3. Die WebSocket-API

Damit ein WebSocket-Endpunkt Nachrichten enkodieren oder dekodieren kann, müssen die zugehörigen Klassen bekannt gemacht werden. Dies erfolgt durch deren Angabe als Parameter in der jeweiligen Klassenannotation. Sollen verschiedene Objekte versendet oder empfangen werden, so müssen entsprechende Encoder- und Decoderklassen bereitgestellt werden. Die Auswahl der passenden Klasse erfolgt anhand ihrer Reihenfolge in der Klassenannotation.

```
@ClientEndpoint( encoders = { MyObjectEncoder.class },
                  decoders = { MyObjectDecoder.class })
public class MyWebSocketClient { ... }
```

Listing 3.15: Java: Angabe der Encoder- und Decoder-Klassen

Jede WebSocket-Verbindung erhält eine eigene Instanz der Encoder- und Decoder-Klassen. Dadurch wird sichergestellt, dass niemals mehr als ein Thread in der `decode()`- oder `encode()`-Methode aktiv ist [Cow14, S. 82].

3.2.4. Konfiguration und Session

Eine der zentralen Komponenten der Java-WebSocket-API ist die `EndpointConfig`-Schnittstelle. Durch sie können einem Endpunkt diverse Meta-Daten übermittelt werden, die den gemeinsamen Zustand aller verbundenen Endpunkte repräsentieren. Je nach Art des Endpunktes wird weiterhin zwischen `ClientEndpointConfig` und `ServerEndpointConfig` unterschieden. Die Parameter sind bei der Verwendung von Annotationen in der jeweiligen Klassenannotation anzugeben. Sie werden zum Zeitpunkt der Bereitstellung dazu verwendet, eine entsprechende Instanz der zugehörigen Endpunktkonfiguration zu erstellen. Diese Konfiguration kann als optionaler Parameter von den Event-Handlern empfangen werden.

Die Methoden der übergeordneten `EndpointConfig`-Schnittstelle sind sowohl für Server- als auch für Clientendpunkte verfügbar. Durch sie können die konfigurierten Encoder- und Decoder-Klassen sowie ein `Map`-Objekt abgerufen werden. Die `Map` kann vom Entwickler genutzt werden, um applikationsspezifische Objekte und Informationen abzulegen.

```
public interface EndpointConfig{
    // Liste aller Encoder-Klassen
    List<Class<? extends Encoder>> getEncoders()
    // Liste aller Decoder-Klassen
    List<Class<? extends Decoder>> getDecoders()
    // Editierbare Map
    Map<String, Object> getUserProperties()
}
```

Listing 3.16: Java: EndpointConfig

Die Konfigurationsschnittstelle des Clients ermöglicht den Zugriff auf clientspezifische Eigenschaften wie die bevorzugten Subprotokolle, Erweiterungen und den Konfigurator. Der Konfigurator ist eine Klasse, die Modifikationen an der HTTP-Anfrage sowie eine manuelle Auswertung der zugehörigen Antwort des Servers ermöglicht.

Subprotokolle und Erweiterungen Die Spezifikation des WebSocket-Protokolls ermöglicht die Bereitstellung von Subprotokollen und Erweiterungen. Die verwendeten HTTP-Felder im Opening-Handshake lauten `Sec-WebSocket-Protocol` und `Sec-WebSocket-Extensions`. Subprotokolle werden dabei als einfache `Strings` angegeben und durch Kommata getrennt. Bei der Vergabe eines eigenen Namens für ein Subprotokoll sollte beachtet werden, dass es nicht zu einer Überschneidung mit den bei der IANA registrierten Bezeichnungen kommt. Werden WebSocket-Frames um zusätzliche Informationen erweitert, um beispielsweise den Inhalt der Nutzdaten näher zu beschreiben, so spricht man von `Extensions` [Cow14, S. 102]. Verwendet eine Applikation Subprotokolle oder Erweiterungen, so werden diese ebenfalls in der Klassenannotation hinterlegt.

Für das Hinzufügen oder Ändern der HTTP-Header muss die verwendete Konfiguratorklasse von der Klasse `ClientEndpointConfig.Configurator` abgeleitet werden. Ein häufiger Anwendungsfall ist die Erweiterung der HTTP-Anfrage um applikations- oder frameworkspezifische Informationen für zusätzliche Funktionen [Cow14, S. 104]. In Listing 3.17 ist ein solcher Fall beispielhaft dargestellt.

```
public class MyClientConfig extends ClientEndpointConfig.Configurator {

    @Override
    public void afterResponse(HandshakeResponse hr) {
        // Prüfen, ob Feature aktiviert werden konnte
        boolean activated = hr.getHeaders().get("My-Feature-Field").get(0)
            .equals("activated");
        // Aktivieren des Features oder Fallback
        super.afterResponse(hr);
    }

    @Override
    public void beforeRequest(Map<String, List<String>> headers) {
        // HTTP-Headerfeld fuer "MyFeature" erzeugen
        List<String> featureParam = new ArrayList<String>();
        featureParam.add("enable");
        // HTTP-Header erweitern
        headers.put("My-Feature-Field", featureParam);
        super.beforeRequest(headers);
    }
}
```

Listing 3.17: Java: ClientEndpointConfig-Konfigurationsklasse

Die Konfiguration des Server durch die `ServerEndpointConfig`-Schnittstelle bietet ebenfalls den Zugriff auf einen Konfigurator. Zudem ist es möglich die unterstützten Subprotokolle und Erweiterungen des Servers sowie die Klasse des Endpunkts abzurufen. Die Ermittlung des relativen Pfads des Serverendpunkts wird in Abschnitt 3.2.5 detailliert behandelt.

3. Die WebSocket-API

Der Konfigurator der `ServerEndpointConfig` stellt zusätzlich diverse Methoden bereit, über die der Verbindungsauflauf gezielt gesteuert werden kann. So kann ein passendes Subprotokoll oder eine Erweiterung gewählt, die Erzeugung einer neuen Instanz manuell durchgeführt und in den Handshake selbst eingegriffen werden. Die Methode `checkOrigin()` prüft, ob der für Browser-clients verbindliche HTTP-Origin-Header auf den selben Webserver verweist, auf dem der Serverendpunkt ausgeführt wird [Cow14, S. 107]. Eine Anpassung ist meist nicht notwendig. Listing 3.18 zeigt den Konfigurator des Serverendpunkts, der den modifizierten Opening-Handshake des Clients aus Listing 3.17 bearbeitet. Dazu wird in der Methode `modifyHandshake()` die Anfrage des Clients auf das entsprechende Feld geprüft und die Antwort des Servers vor dem Versenden angepasst.

```
public class MyServerConfig extends ServerEndpointConfig.Configurator {

    @Override // Hier kann die Prüfung des Origin-Headers durchgeführt werden
    public boolean checkOrigin(String originHeaderValue) { ... }

    @Override // Hier kann die Erzeugung einer neuen Instanz angepasst werden
    public <T> T getEndpointInstance(Class<T> endpointClass) throws
        InstantiationException { ... }

    @Override // Hier kann der Standardablauf der Wahl der Erweiterung
    // verändert werden
    public List<Extension> getNegotiatedExtensions(List<Extension> installed,
        List<Extension> requested) { ... }

    @Override // Hier kann der Standardablauf der Protokollwahl verändert
    // werden
    public String getNegotiatedSubprotocol(List<String> supported, List<String>
        requested) { ... }

    @Override
    public void modifyHandshake(ServerEndpointConfig sec, HandshakeRequest
        request, HandshakeResponse response) {
        // Anfrage auf zusaätzliches Header-Feld prüfen
        boolean activateMyFeature = request.getHeaders().get("My-Feature-Field").
            get(0).equals("enable");

        if(activateMyFeature) {
            // HTTP-Headerfeld für "MyFeature" erzeugen
            List<String> featureParam = new ArrayList<String>();
            featureParam.add("activated");

            // HTTP-Antwort-Header erweitern
            response.getHeaders().put("My-Feature-Field", featureParam);
        }
    }
}
```

Listing 3.18: Java: ServerEndpointConfig-Konfigurationsklasse

3. Die WebSocket-API

Durch den Einsatz der `EndpointConfig`-Klassen werden Einstellungen festgelegt, die übergreifend für alle verbundenen Endpunkte gelten. Das `Session`-Objekt hingegen repräsentiert alle Einstellungen und Zustände bezüglich einer einzelnen WebSocket-Verbindung. Für einen WebSocket-Client existiert nur eine `Session`, wohingegen ein WebSocket-Server Zugriff auf die Sessions aller verbundenen Clients hat. Ein `Session`-Objekt wird bei einer erfolgreich aufgebauten Verbindung erstellt und kann über die mit `@OnOpen` gekennzeichnete Methode empfangen werden. Durch den Aufruf der Methode `getId()` lässt sich ein eindeutiger Bezeichner zur Identifizierung einer `Session` ermitteln. Weiterhin können zur Laufzeit die Methoden zur Bearbeitung von Events ermittelt, hinzugefügt oder entfernt werden. Ändert sich im Laufe einer Verbindung beispielsweise die Art, wie ein Client Nachrichten bearbeiten soll, so kann durch den Aufruf der `removeMessageHandler`- sowie `addMessageHandler`-Methode des zugehörigen `Session`-Objekts der `@OnMessage`-Handler ausgetauscht werden. Wie bereits in Abschnitt 3.2.2 dargestellt, erfolgt das Senden von Nachrichten über `Basic`- oder `Async`-Objekte des `RemoteEndpoint`-Interfaces. Diese werden durch die Methoden `getBasicRemote()` respektive `getAsyncRemote()` der zugehörigen `Session` bereitgestellt.

Die Konfiguration verschiedener verbindungsspezifischer Eigenschaften kann ebenfalls über das `Session`-Objekt verändert werden. Auf einige dieser Merkmale ist der Zugriff lediglich lesend möglich, da deren Werte während des Verbindungsaufbaus zwischen Server und Client abgestimmt werden. Hierbei handelt es sich um die Liste der verfügbaren Subprotokolle und Erweiterungen, die verwendete Protokollversion und den authentifizierten Benutzer, repräsentiert durch ein Java-`UserPrincipal`-Objekt. Des Weiteren kann geprüft werden, ob die Verbindung verschlüsselt erfolgt. Der schreibende Zugriff ist auf folgende Eigenschaften möglich:

`userProperties` Ein `map`-Objekt, in dem benutzerspezifische Schlüssel-Werte-Paare gespeichert werden können.

`maxBinaryMessageSize` Beschränkt die Größe einer ankommenden binären Nachricht.

`maxTextMessageSize` Beschränkt die Größe einer ankommenden textuellen Nachricht.

`maxIdleTimeout` Legt den längsten Zeitabstand fest, in dem keine Aktivität zwischen den Endpunkten festgestellt wird, bevor die Verbindung geschlossen wird.

Alle Eigenschaften sind auch übergreifend für alle Endpunkte konfigurierbar. Beispielsweise kann die Beschränkung der Nachrichtengröße auch innerhalb der Annotation `@OnMessage` als Parameter hinterlegt werden. Es gilt daher abzuwagen, ob die Konfiguration nur individuell für bestimmte Clients oder übergreifend zutreffen soll.

Der Lebenszyklus einer WebSocket-Verbindung (siehe Abbildung 3.1) kann durch das `Session`-Objekt der Java-WebSocket-API überprüft und gesteuert werden. Die Methode `isOpen()` zeigt an, ob diese geöffnet ist und `getOpenSessions()` listet alle offenen Verbindungen zu dem Endpunkt auf, mit dem das verwendete `Session`-Objekt verbunden ist. Allerdings können diese

3. Die WebSocket-API

Methoden nicht garantieren, dass die zugrundeliegenden Verbindungen tatsächlich geöffnet sind. Wurde der andere Endpunkt unerwartet beendet oder die Verbindung getrennt, erfolgt keine Benachrichtigung darüber. Um diesbezüglich sicherzugehen, kann eine `PingMessage` an den gegenüberliegenden Endpunkt gesendet und auf eine entsprechende Bestätigung gewartet werden. Soll eine Verbindung geschlossen werden, erfolgt dies über den Aufruf der `close()`-Methode. Optional kann ein `CloseReason`-Argument übergeben werden, in dem ein `CloseCode` und eine textuelle Begründung hinterlegt werden können. Diese können von dem `@OnClose`-Handler des empfangenden Endpunkts abgefragt und interpretiert werden.

3.2.5. Path-Mapping

Die Konfiguration eines WebSocket-Serverendpunkts verlangt die Angabe eines Pfades, unter dem der Endpunkt relativ zum Kontextpfad der Anwendung erreichbar ist. Die Zuordnung dieses Pfads auf einen bestimmten Endpunkt wird als *Path Mapping* bezeichnet. Die Identifikation eines WebSocket-Serverendpunkts erfolgt durch einen *Unique Resource Identifier* (URI). Ein URI für WebSocket-Verbindungen wird nach folgendem Schema aufgebaut:

```
<ws-protocol-scheme>://<host>:<port>/<uri-path>?<query-string>
```

Das Protokollschema ist analog zu HTTP `http(s)` bei WebSocket-URIs `ws(s)`. Dabei entspricht die `wss`-Angabe einer verschlüsselten WebSocket-Verbindung. Die Angabe des Hosts kann sowohl durch dessen IP-Adresse als auch durch seinen kanonischen Namen erfolgen. Die Angabe der Port-Nummer muss lediglich dann erfolgen, wenn der zugehörige Webserver nicht unter den für WebSockets standardisierten Ports 80 bzw. 443 erreichbar ist. Der URI-Pfad entspricht der `value`-Angabe in der Konfiguration des Serverendpunkts. Es gilt zu beachten, dass insbesondere bei der Nutzung von *Java-Web-Archiven* (WAR), der Name der Anwendung häufig als deren Ausgangspunkt verwendet wird. Trägt die WAR-Datei die Bezeichnung `example` und der WebSocket-Serverendpunkt ist unter `/test` erreichbar, so lautet der URI-Pfad `/example/test`.

Der URI-Pfad kann dabei exakt oder unter Verwendung von *URI-Templates* angegeben werden. Für die exakte Angabe ist es ausreichend, die URI-Pfadsegmente im `value`-Feld des WebSocket-Servers, getrennt durch `/`, zu hinterlegen. Clients können daraufhin die Verbindung unter diesem Pfad herstellen. Die Verwendung eines *URI-Templates* ermöglicht die Angabe von Pfadvariablen, durch die verschiedene Aufrufe der URI unterschiedlich interpretiert werden können. Die Pfadvariablen werden durch geschweifte Klammern innerhalb des URI-Pfads angegeben und dürfen lediglich einfache Bezeichner ohne reservierte Sonderzeichen enthalten. Ein gültiger URI-Pfad unter Verwendung von Templates wäre demnach:

```
/app/{access-level}/v1
```

3. Die WebSocket-API

Der Aufruf des URI-Pfads mit verschiedenen Werten wie z. B. /app/new/v1 oder /app/gold/v1 ist demnach gültig und kann vom WebSocket-Serverendpunkt unter Verwendung des **Session**-Objekts interpretiert werden. Dazu kann die Methode `getPathParameters()` eingesetzt werden. Das zurückgelieferte **Map**-Objekt beinhaltet die Pfadvariablen, wobei der Schlüsselwert der **Map** dem Variablennamen und der Wert dem übergebenen **String** entspricht. Zudem kann die Annotation `@PathParam` verwendet werden. Durch diese kann in einer der Methoden des WebSocket-Lebenszyklus der angegebene Pfadparameter als Parameter der Methode empfangen werden.

```
@ServerEndpoint(value="/app/{access-level}")
public class MyServer{
    @OnOpen
    public void onOpen(@PathParam("access-level") String level, String msg) {
        // Nachricht differenziert nach Berechtigung bearbeiten.
    }
}
```

Listing 3.19: Java: PathParam

Als Datentypen für `@PathParam`-Parameter können alle primitiven Datentypen verwendet werden. Die Umwandlung in den jeweiligen Zieldatentyp erfolgt automatisch. Kann die Konvertierung nicht durchgeführt werden, so bewirkt dies eine `DecodeException`. Die WebSocket-Verbindung wird jedoch nicht automatisch beendet.

Die Übergabe zusätzlicher Parameter kann durch die Verwendung von *Query Strings* optional erfolgen. Dazu können Schlüssel-Werte-Paare an den URI-Pfad mit ? angehängt und durch & verknüpft werden. Query Strings werden von der WebSocket-Implementierung bei der Auswahl des zu verbindenden Endpunkts ignoriert [Cow14, S. 156]. Sie können jedoch beim Verbindungsauflauf genutzt werden, um beispielsweise benutzerspezifische Einstellungen vorzunehmen. Hierfür kann die Methode `getQueryString()` oder `getRequestParameterMap()` des **Session**-Objekts verwendet werden. Erstere liefert die vollständige Zeichenkette, die nach dem URI-Pfad folgt, wohingegen die zweite Methode ein `Map<String, List<String>>`-Objekt der Parameter erstellt. Gültige Aufrufe mit Parametern in einem Query String wären beispielsweise:

```
/app/{access-level}/v1?username=admin&password=secure
/app/premium?showpictures=1
```

Durch die Verwendung der `getRequestParameterMap()`-Methode können diese Parameter eingesehen und verarbeitet werden. Eine Kombination aus Pfadparametern und Query Strings ist ebenfalls möglich. Bei der Auswahl der Übertragungsart verbindungsspezifischer Parameter sollte beachtet werden, dass Pfadparameter nur einzelne Werte für einen Schlüssel zulassen, Query Strings jedoch eine Listenstruktur ermöglichen. Sollen Parameter ohne einen Neustart der Verbindung übertragen werden, so müssen diese als WebSocket-Nachrichten gesendet werden. Der Autor eines Großteils der Java™ API for WebSocket, Danny Coward, hat neun Regeln zur generellen Verwendung des *Path Mapping* festgelegt [Cow14, S. 146-159]:

3. Die WebSocket-API

Regel 1: “Der URI-Pfad eines WebSocket-Serverendpunkts wird relativ zum Kontextpfad des Webservers angegeben, auf dem er bereitgestellt wird.”

Regel 2: “Innerhalb einer Anwendung dürfen keine zwei Endpunkte denselben URI-Pfad verwenden.”

Regel 3: “Die Interpretation von URI-Pfaden erfolgt unter Beachtung von Groß-/Kleinschreibung.”

Regel 4: “Ein eingehender Opening-Handshake passt genau dann zu dem URI-Template eines Serverendpunkts, wenn er die Rahmenbedingungen des Templates einhält.”

Regel 5: “Es werden nur einfache URI-Templates (Level 1) unterstützt³.”

Regel 6: “Eine eingehende URI trifft nur dann auf ein Template zu, wenn die Pfadparameter mit einem Wert übergeben werden.”

Regel 7: “Analog zu Regel 2 dürfen keine zwei Endpunkte innerhalb einer Anwendung das gleiche URI-Template verwenden.”

Regel 8: “Ein Query String kann nicht zur Auswahl des zu verwendenden Endpunkts eingesetzt werden.”

Regel 9: “Bei der Auswahl des zu verwendenden Endpunkts vergleicht die Java-WebSocket-API Pfadsegment für Pfadsegment beginnend von links. Eine exakte Übereinstimmung wird dabei der Übereinstimmung mit einem Pfadparameter vorgezogen.”

Bei Regel 8 muss angemerkt werden, dass ein übergebener Query String bei der Auswahl des passenden Endpunkts vollständig ignoriert wird. Das folgende Beispiel sorgt für ein besseres Verständnis der Problematik, die in Regel 9 aufgeführt ist:

Endpunkt 1 unter: /{language}/professional

Endpunkt 2 unter: /java/{skill}

Hier stellt eine Anwendung zwei Endpunkte unter der Verwendung von URI-Templates bereit. Trifft nun eine Anfrage an /java/professional ein, so kann nicht garantiert werden, welcher der beiden Endpunkte zuständig ist. Durch die Festlegung des Vorzugs exakter Übereinstimmung in Regel 9 wird Endpunkt 2 bereitgestellt, da java kein Parameter, sondern ein festgelegter Wert ist und mit der angefragten URI übereinstimmt. Zudem erfolgt diese Überprüfung für jedes Segment beginnend von links, wodurch die exakte Übereinstimmung von professional nicht weiter geprüft wird.

³Vollständige Definitionen von URI-Templates sind im zugehörigen RFC einsehbar, siehe [Not+12]

4. Die Chat-Anwendung

Im Rahmen des Bachelorstudiums der Wirtschaftsinformatik an der Hochschule München erlernen Studierende ab dem dritten Fachsemester grundlegende und fortgeschrittene Merkmale der Datenkommunikation. Die zugehörige Modulbeschreibung der Hochschule München erklärt für diesen Kurs die Vermittlung von Prinzipien der Datenkommunikation, der Netzwerkkonzeption sowie der Entwicklung von Kommunikationsanwendungen als Ziel. Die Studierenden sollen dabei Kenntnisse über die Prinzipien der Datenkommunikation und der Netzwerkkonzeption erlangen sowie die zentralen Protokollmechanismen nachvollziehen können. Anschließend sollen sie in der Lage sein, Kommunikationsanwendungen für konkrete Problemstellungen zu entwickeln [siehe Hoc17]. Um die erklärten Ziele des Moduls zu erreichen, liegt der Schwerpunkt auf folgenden Inhalten [aus Man17b]:

- Spezielle Aspekte der Datenkommunikation wie verbindungsorientierte und verbindungslose Kommunikation, gesicherte Datenübertragung, Routing, Segmentierung und Fragmentierung sowie spezielle Aspekte der Netzwerkkonzeption.
- Studium ausgewählter Problemstellungen realer, insbesondere TCP/IP-basierter Datenkommunikationssysteme.
- Verwendung von Transportzugriffsschnittstellen wie Sockets.
- Entwicklung von Kommunikationsanwendungen auf Basis von Transportprotokollen wie TCP und UDP.
- Grundbegriffe und Anwendungsbereiche von Kommunikationsarchitekturen (z. B. Schichtenmodell, Protokolle, Netzarten und -technologien).

Da Studierende nach Abschluss des Moduls in der Lage sein sollen, anhand definierter Anforderungen Kommunikationsanwendungen zu entwickeln, müssen sie unter der Leitung von Herrn Prof. Dr. Mandl⁴ eine Studienarbeit erstellen, in der eine Chat-Anwendung programmiert, evaluiert und vorgestellt werden muss. Der Schwerpunkt der Anwendung liegt auf der Entwicklung von Kommunikationsprotokollen auf Basis heutiger Transportprotokolle [Man17b]. Als Transportprotokoll kommt dabei das verbindungsorientierte *Transmission Control Protocol* (TCP) zum Einsatz. Die verwendete Programmiersprache ist *Java* der Oracle Corporation.

Durch die Einführung des WebSocket-Protokolls sollen Webanwendungen unter möglichst geringem Einsatz von Verwaltungsdaten die Vorteile von TCP nutzen können. Dabei steht insbesondere die Möglichkeit bidirektional zu kommunizieren im Vordergrund. Die Nutzung von WebSockets im Rahmen der Studienarbeit stellt somit einen neuen Anwendungsfall dar, der im Rahmen dieser Bachelorarbeit geplant, umgesetzt und getestet werden soll. Dazu wird in diesem Kapitel zunächst die bestehende Anwendung analysiert, um daraus die Anforderungen der neuen Applikation ableiten zu können. Anschließend wird deren Dokumentation zur gezielten Planung der Umsetzung verwendet. Die Implementierung und die Leistungsmessung der WebSocket-Chat-Anwendung bilden den Abschluss dieses Kapitels.

⁴unter https://www.wirtschaftsinformatik-muenchen.de/?page_id=4 (letzter Zugriff am 14.01.2018)

4.1. Analyse der bestehenden Chat-Anwendung

Die detaillierte Prüfung der bestehenden Anwendung stellt einen wesentlichen Teilaспект der Definition der Anforderungen an das neue Chat-Programm dar. Die Technik der Informationsgewinnung aus der Dokumentation oder Implementierung eines Altsystems wird als *Systemarchäologie* bezeichnet [PR09, S. 38]. Ziel dabei ist die Sicherstellung der Übernahme aller relevanten bereits implementierten Funktionen. Sie gilt als die einzige Technik, die eine vollständige Umstellung von Alt- zu Neusystemen gewährleistet [PR09, S. 38]. Diese Methode kann beim vorliegenden Entwicklungsprojekt angewendet werden, da die Dokumentation sowie die Implementierung der bestehenden Anwendung in ausreichendem Umfang und ausreichender Qualität vorliegen. Des Weiteren unterscheiden sich Neu- und Altsystem nur im Hinblick auf die zugrundeliegende Architektur und nicht bezüglich der fachlichen Funktionalität. Aus diesem Grund wird zunächst die grundlegende Struktur analysiert, die relevanten Klassen und Bibliotheken werden identifiziert und die prinzipielle Funktionsweise wird dargestellt.

4.1.1. Struktur des Projekts

Beim vorliegenden Projekt handelt es sich um ein *Java-Eclipse-Projekt*, unterteilt in sechs *Packages*. Ein Package (Paket) enthält dabei eine oder mehrere Klassen, die sich einen Geltungsbereich für Klassen teilen [Kre06]. Ihre Bezeichnung spiegelt den Aufgabenbereich der enthaltenen Klassen wider.

edu.hm.dako.chat.benchmarking Dieses Package beinhaltet Klassen und Interfaces zur grafischen Darstellung und Bereitstellung der Funktionalität für den Belastungstest der Anwendung.

edu.hm.dako.chat.client Hier sind alle notwendigen Klassen und Interfaces der Client-Anwendung zur grafischen Darstellung und Bereitstellung der Funktionalität enthalten.

edu.hm.dako.chat.common Gemeinsam genutzte Klassen, die beispielsweise Datenobjekte repräsentieren oder in denen Messwerte abgelegt werden können, sind in diesem Package enthalten. Sowohl Server als auch Client können diese Klassen verwenden. Die Übertragung der Nachrichten erfolgt durch Objekte der Klasse `ChatPDU` innerhalb dieses Packages.

edu.hm.dako.chat.connection In diesem Package befinden sich die Klassen, durch die die Verbindungssteuerung sowohl auf Server- als auch auf Clientseite erfolgt. Die Klassen und Interfaces sind dabei klar auf die Verwendung von TCP als Transportprotokoll ausgelegt.

edu.hm.dako.chat.server Hier sind alle notwendigen Klassen und Interfaces der Server-Anwendung zur grafischen Darstellung und Bereitstellung der Funktionalität enthalten.

edu.hm.dako.chat.tcp Diese Klassen implementieren die im `connection`-Package definierten Funktionalitäten zur Verwendung von TCP als Transportprotokoll.

Weiterhin befinden sich im Projektverzeichnis zwei Konfigurationsdateien. Sie werden benötigt, um die `log4j`-Bibliothek zu konfigurieren. Dabei handelt es sich um eine Erweiterung für die Ausgabe von Informationen auf einer Konsole oder in eine Datei.

4.1.2. Auswahl zu übernehmender Klassen

Die strukturellen Änderungen zwischen der bisherigen Chat-Anwendung und dem neuen WebSocket-Chat erfordern eine sorgfältige Auswahl der Klassen, die portiert werden sollen. Insbesondere Klassen zur Steuerung der Verbindung sowie zur Darstellung und Implementierung von Server- und Client-Anwendung sind zu prüfen. Dabei sollen auch nicht verwendete Programmteile des Altsystems identifiziert werden.

Das Package `edu.hm.dako.chat.common` beinhaltet diverse Datenobjekte, die sowohl vom Client als auch vom Server verwendet werden. Die Klassen `MemoryTest` und `TestNanoTime` dienen lediglich zu Testzwecken und sind für den Einsatz in der Studienarbeit nicht relevant. Die Funktionalität aller weiteren Klassen soll auch weiterhin bestehen bleiben und muss daher übernommen werden.

Im Package `edu.hm.dako.chat.benchmarking` sind die Funktionen zur Durchführung der Leistungsmessung gebündelt. Die Klasse `BenchmarkingUserInterfaceSimulation` simuliert jedoch nur eine Benutzeroberfläche und wird weiterhin nicht verwendet. Die Funktionalität aller weiteren Klassen soll bestehen bleiben und muss daher übernommen werden.

Die Packages `edu.hm.dako.chat.server` und `edu.hm.dako.chat.client` repräsentieren alle Klassen zur Darstellung und Implementierung eines Clients bzw. eines Servers. Die grafischen Oberflächen werden mittels JavaFX implementiert, einer Bibliothek zur Erstellung von Benutzeroberflächen durch den Einsatz von FXML. Dabei handelt es sich um eine auf XML basierende Auszeichnungssprache [Ora14b]. Da eine Webanwendung entwickelt werden soll, erfolgt die Darstellung der Client-Anwendung durch einen Webbrower. Die aktuellen Klassen zur Darstellung werden demnach nicht weiter benötigt, wohingegen die Funktionalität aller weiteren Klassen bestehen bleiben soll und daher übernommen werden muss.

Bei den Packages `edu.hm.dako.chat.connection` und `edu.hm.dako.chat.tcp` handelt es sich um die Klassen und Interfaces, die zur Repräsentation und Steuerung der Verbindung eingesetzt werden. Allerdings liegt deren Fokus auf der Implementierung einer Verbindung mit TCP oder dem *User Datagram Protocol* (UDP). Aus diesem Package kann demnach keine Klasse weiterhin verwendet werden.

Die Anwendung wird ergänzt durch einige externe Bibliotheken, die zusätzliche Funktionen einbringen. Die Bibliothek `netty-all-4.0.0.final` wird verwendet, um die übertragenen TCP-Streams in Objekte zu konvertieren und muss daher nicht übernommen werden. Neben der bereits erwähnten Bibliothek `log4j-1.2.15` kommt lediglich `commons-math3-3.6` zur Berechnung der statistischen Werte zum Einsatz und muss daher übernommen werden. Die Verwendung aller weiteren Bibliotheken wird nicht fortgesetzt.

4.1.3. Funktionsweise

Die Ermittlung der abzubildenden Funktionsweise ist essentiell für die Implementierung der WebSocket-Chat-Anwendung. Die obligatorische Übernahme der daraus abgeleiteten Anforderungen gewährleistet die bestehende Funktionalität weiterhin. Die Chat-Anwendung setzt sich grundlegend aus drei Teilbereichen zusammen. Der Serverteil ermöglicht die Auswahl zwischen zwei unterschiedlichen Implementierungsvarianten: **Simple** und **Advanced**. Zudem kann der Port eingestellt werden, auf dem der Server auf neue Anfragen wartet. Des Weiteren können Senden- und Empfangspuffergrößen in Byte angegeben werden. Der Teil des Clients besteht aus einer Anmeldemaske, in der der Benutzer seinen Namen sowie die IP-Adresse und den Port des Zielservers eintragen kann. Auch hier muss ein Implementierungstyp angegeben werden. Konnte die Verbindung zum Server hergestellt werden und stimmen die Implementierungstypen überein, so wechselt die Darstellung in die Chat-Ansicht. Diese besteht aus einer Liste der angemeldeten Benutzer sowie einem Bereich, in dem die Chat-Nachrichten dargestellt werden. Zudem befinden sich dort Schaltflächen zum Senden von Nachrichten und zum Abmelden. Der dritte Teilbereich stellt die Funktionalität der Leistungsmessung bereit. Die Benutzeroberfläche dieses Benchmark-Clients bietet eine Vielzahl von Konfigurationsmöglichkeiten. Weiterhin finden sich dort diverse Felder für Laufzeitdaten und Messergebnisse. Der Benchmark-Client kann mit den vorhandenen Schaltflächen gestartet, abgebrochen, neugestartet und vollständig beendet werden. Eine detaillierte Ansicht aller verfügbaren Optionen zeigt die Abbildung im Anhang.

Kern der Anwendung ist die Klasse **ChatPDU**. Eine *Protocol Data Unit* (PDU) repräsentiert die (Protokoll-)Nachrichten, die zwischen Server und Client ausgetauscht werden. Dabei enthalten sie Statusinformationen, durch die der Zustand der Verbindung verändert werden kann.

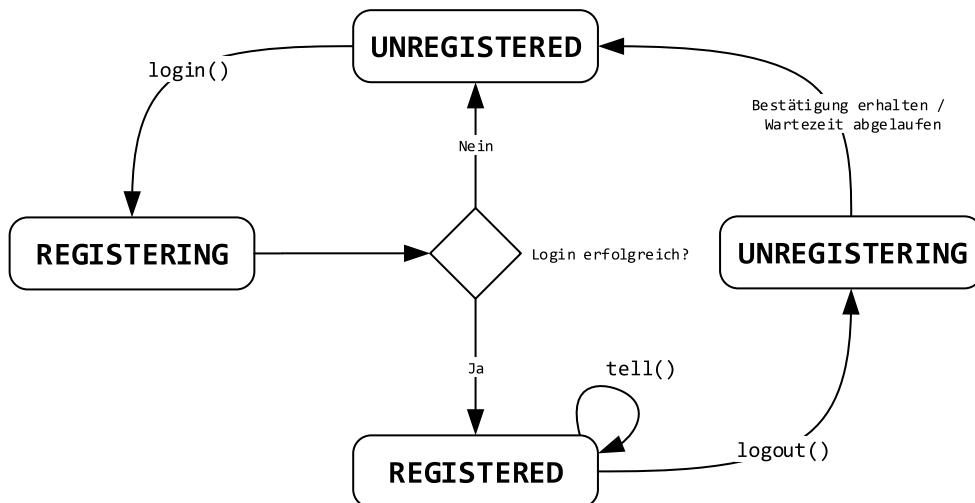


Abbildung 4.1: Zustände in der Chat-Anwendung

Durch den Aufruf der `login()`-Methode wechselt der Client zunächst in den Zustand **REGISTERING**. Bestätigt der Server die korrekte Anmeldung, so wechselt der Client in den Zustand **REGISTERED**.

4. Die Chat-Anwendung

Das Senden von Nachrichten erfolgt mittels `tell()` und ändert dabei nicht den Zustand des Clients. Die Abmeldung wird durch die Methode `logout()` initiiert, woraufhin sich der Client im Zustand **UNREGISTERING** befindet. Wird die Abmeldung vom Server bestätigt bzw. erreicht die Bestätigung den Client nicht innerhalb einer festgelegten Zeitspanne, so wird angenommen, dass die Verbindung geschlossen werden kann und der Client wechselt in den Zustand **UNREGISTERED**.

Jeder Methodenaufruf generiert dabei eine **ChatPDU** mit einer Anfrage der zugehörigen Aktion. Beispielsweise wird bei der Anmeldung der PDU-Typ **LOGIN_REQUEST** und beim Versand einer Nachricht der Typ **MESSAGE_REQUEST** verwendet. Die Verarbeitung der Anfrage ist abhängig von der Implementierung des Servers. Die einfache Implementierung (**Simple**) benachrichtigt alle angemeldeten Clients über das eingetretene Ereignis und antwortet dem anfragenden Client umgehend. Die komplexe Implementierung (**Advanced**) sieht vor, dass alle versendeten Events zunächst bestätigt werden müssen, bevor die Antwort an den initiiierenden Client versendet wird. Der Ablauf der komplexen Implementierung ist beispielhaft für den Anmeldevorgang in Abbildung 4.2 dargestellt.

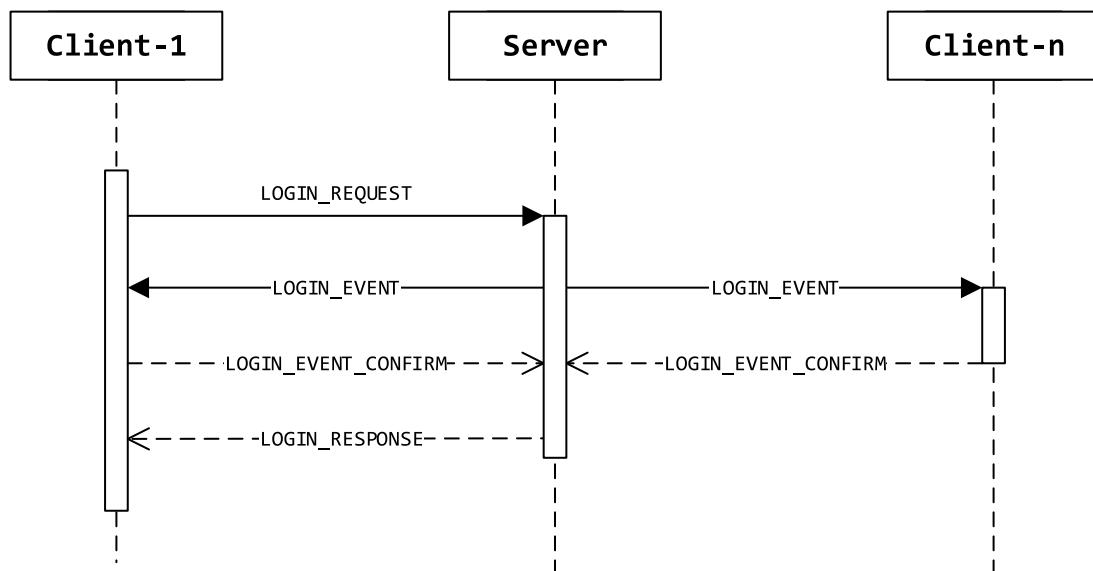


Abbildung 4.2: Ablauf des Login-Vorgangs (AdvancedChat)

Neben den Anfragen, die als **REQUESTS** versendet werden, können die PDUs des Chat-Protokolls von den Typen **EVENT** für neue Ereignisse, **EVENT_CONFIRM** zur Bestätigung eines Ereignisses und **RESPONSE** zur Bestätigung der Anfrage sein. Anfragen und Ereignisbestätigungen werden ausschließlich von einem Client verwendet, wohingegen ein Server lediglich über neue Ereignisse informiert und Anfragen bestätigt. Da die Protokollmechanismen genutzt werden sollen, um die „Programmierung von Kommunikationsanwendungen [...] zu erlernen und Erfahrungen mit Transportprotokollen zu sammeln“ [vgl. Man17a, S. 2], müssen diese im WebSocket-Chat unverändert abgebildet werden.

4. Die Chat-Anwendung

Das Stellen von Anfragen sowie deren Bearbeitung erfolgt nebenläufig durch die Verwendung von *Threads*. Dabei handelt es sich um leichtgewichtige Prozesse innerhalb eines Prozesses, die sich dessen Ressourcen teilen. Da der gemeinsame Zugriff auf die unterliegenden Ressourcen zu unvorhergesehenen Ergebnissen (*Race Conditions*) führen kann, müssen die Threads mittels geeigneter Mechanismen synchronisiert werden. Die Chat-Anwendung verwendet drei Threads pro Verbindung zwischen Server und Client. Der Server stellt für jeden neuen Client einen dedizierten Thread zur Bearbeitung der Anfragen zur Verfügung. Der Client verfügt über zwei Threads, wobei einer für die Bearbeitung der Benutzerinteraktionen zuständig ist und der andere die Nachrichten des Servers annimmt und bearbeitet. Sowohl die Client- als auch die Server-Anwendung werden als eigenständige Prozesse ausgeführt und können unabhängig voneinander auf verschiedene Hosts verteilt werden. Der WebSocket-Chat soll die Anwendungsteile ebenfalls trennen, wobei die Client-Anwendung im Browser und die Server-Anwendung auf einem Webserver ausgeführt wird. Der Teil des Clients wird bei einer Webanwendung ebenfalls durch den Webserver ausgeliefert. Durch die Verwendung der Java-WebSocket-API wird die Verwaltung der Threads von der spezifischen Implementierung übernommen und kann daher nur eingeschränkt angepasst werden.

Die Übertragung der Protokollnachrichten erfolgt synchron durch die kontinuierliche Übertragung von Objektströmen. Eine zu übermittelnde ChatPDU wird zunächst serialisiert und anschließend in den Puffer des TCP-Sockets übertragen. Für die Serialisierung implementiert die Klasse das Interface **Serializable**. Überschreitet die Größe der Nachricht den reservierten Speicherplatz des Puffers nicht, so kann der aktive Thread im Anschluss weitere Aufgaben bearbeiten. Im Gegensatz zum Nachrichten-basierten WebSocket-Protokoll erfolgt die Übertragung mittels TCP-Sockets *Stream-basiert*. Dabei werden die Daten in einem kontinuierlichen Fluss übertragen, dessen Ende vorab nicht ersichtlich ist. Aufgrund der Unterschiede im Aufbau der Protokolle sind nicht alle Eigenschaften der Übertragung von Protokollnachrichten der bestehenden Chat-Anwendung abbildbar. Da jedoch im WebSocket-Chat ebenfalls ChatPDUs übertragen werden sollen, müssen diese entsprechend vor der Übertragung transformiert werden.

4.2. Anforderungen an den WebSocket-Chat

Die gewonnenen Erkenntnisse aus der Analyse der bestehenden Chat-Anwendung sollen im Folgenden dazu verwendet werden, die Anforderungen an den WebSocket-Chat zu spezifizieren. Eine Anforderungsspezifikation entspricht einer systematisch dargestellten Sammlung von Anforderungen an ein System [PR09, S. 43]. Eine Anforderung ist im IEEE-Standard 620.16-1990 definiert als „eine Bedingung oder Fähigkeit, die ein System oder eine Komponente erfüllen oder besitzen muss, um einen Vertrag, einen Standard, eine Spezifikation oder anderweitig formal vorgeschriebene Dokumente zu erfüllen.“ Dabei wird zwischen funktionalen und nicht-funktionalen Anforderungen unterschieden. Funktionale Anforderungen beziehen sich auf ein erwartetes Ergebnis des Verhaltens einer bereitgestellten Funktion des Systems, wohingegen nicht-funktionale Anforderungen an Qualitätsmerkmale des Systems gestellt werden [PR09, S. 16]. Diese Quali-

4. Die Chat-Anwendung

tätsanforderungen können beispielsweise die Geschwindigkeit oder Verfügbarkeit eines Systems betreffen. Die ermittelten Anforderungen dienen als Basis für den Entwurf des Systems sowie die anschließende Implementierung. Des Weiteren werden aus ihnen Testfälle abgeleitet, um die Einhaltung der Systemanforderungen zu überprüfen.

4.2.1. Funktionale Anforderungen

Durch die Analyse der Funktionen der bestehenden Chat-Anwendung wurde eine Vielzahl von Anforderungen an den WebSocket-Chat ermittelt. Im Folgenden werden diese zusammengefasst und in die zugehörigen Bereiche eingeteilt.

System und Systemkontext

Die Chat-Anwendung ist aufgeteilt in Server-, Client- und Benchmarking-Teil. Jeder Teil ist in einem eigenen Prozess ausführbar und kann auf verschiedenen Hostsystemen lokalisiert sein. Der WebSocket-Chat sollte diese Aufteilung weiterhin beibehalten, kann jedoch die Server- und Client-Anwendung in einer Einheit bereitstellen. Der Benchmark-Client muss weiterhin als eigenständige Einheit verfügbar sein, um Leistungsmessungen in verschiedenen Szenarien zu ermöglichen. Die Server-Anwendung soll in Java 8 durch die JavaTM API for WebSocket realisiert werden, wohingegen die Client-Anwendung in TypeScript unter Verwendung des Front-End-Webapplikationsframeworks Angular 4 entwickelt wird. Der Benchmark-Client soll als unabhängige Konsolenanwendung in Java 8 implementiert werden und die bisherigen Auswertungsmöglichkeiten bereitstellen.

Die Kommunikation zwischen den Anwendungsteilen muss auf dem WebSocket-Protokoll basieren. Dabei müssen die in Kapitel 2 und Kapitel 3 beschriebenen Vorgehensweisen und Standards eingehalten werden. Eine WebSocket-Verbindung zwischen Server und Client muss immer über den HTTP-Handshake hergestellt werden.

Die Protokollnachrichten, die zwischen den Kommunikationsendpunkten ausgetauscht werden, müssen den Vorgaben der `ChatPDUs` entsprechen. Dabei sind die Attribute und Methoden vollständig zu übernehmen. Abbildung 4.3 bietet einen Überblick über die wesentlichen Elemente der `ChatPDU`-Klasse. Zur Verbesserung der Lesbarkeit wurde auf *Getter*- und *Setter*-Methoden sowie auf die Parameter der statischen Methoden in der Darstellung verzichtet. Alle statischen Operationen dienen der Erzeugung einer `ChatPDU` eines bestimmten PDU-Typs. Es werden nur die Attribute mit entsprechenden Werten versehen, die im jeweiligen PDU-Typ benötigt werden. So soll beispielsweise keine Liste aller angemeldeten Clients beim Senden einer neuen Chat-Nachricht (`MESSAGE_EVENT`) hinterlegt werden. Die verschiedenen Zustände einer `ChatPDU` sind in der Enumeration `PduType` definiert. Der Status des Clients wird der Enumeration `ClientConversationStatus` entnommen und kann die in Abbildung 4.1 dargestellten Werte annehmen.

4. Die Chat-Anwendung



Abbildung 4.3: Aufbau einer ChatPDU

Die bestehende Chat-Anwendung verfügt über zwei unterschiedliche Implementierungsformen. In der einfachen Variante (**SimpleChat**) wird die Antwort des Servers auf die Anfrage eines Clients umgehend versendet. Die komplexe Variante (**AdvancedChat**) erwartet zunächst eine Bestätigung des Ereignisses durch alle aktiven Clients, bevor die Antwort an den Initiator versendet wird. Ein Client gilt als aktiv, wenn er sich nicht im Zustand **UNREGISTERED** befindet. Beide Implementierungsformen dienen als wichtiges didaktisches Mittel und müssen daher im WebSocket-Chat abgebildet werden. Es soll möglich sein, zwischen beiden Implementierungsformen zu wechseln.

Funktionen des Servers

Der Server stellt die zentrale Komponente der WebSocket-Chat-Anwendung dar. Er muss die Nachrichten der Clients empfangen und verarbeiten können. Dazu muss der Server in der Lage sein, Informationen zu jedem verbundenen Client zu speichern und dem Initiator entsprechende Anfragen zuzuordnen. Der Server muss daher jederzeit den Zustand und die Adresse aller verbundenen Clients abrufen können. Treffen Anfragen eines Clients ein, die eine Statusänderung zur Folge haben, so muss der Server den gespeicherten Zustand des Clients umgehend ändern können. Die Benutzernamen innerhalb des WebSocket-Chats müssen eindeutig sein. Verwendet ein Client beim Verbindungsauflauf einen bereits existierenden Namen, so muss der Server die Anmeldung verweigern und einen entsprechenden Fehlercode in seiner Antwort hinterlegen. Chat-Nachrichten dürfen ausschließlich von Clients gesendet werden, die sich im Status **REGISTERED** befinden. Eintreffende Chat-Nachrichten von Clients mit einem anderen Status müssen verworfen werden. Der

4. Die Chat-Anwendung

Verbindungsabbau wird clientseitig gestartet und muss vom Server bestätigt werden. Befindet sich der Client im Status **UNREGISTERED**, kann eine neue Verbindung aufgebaut werden. Bei der Verwendung der **Simple-Chat**-Implementierung muss der Server bei einer eintreffenden Anfrage alle aktiven Clients über das Ereignis informieren und anschließend eine Bestätigung an den initiierenden Client senden. Die komplexere **AdvancedChat**-Implementierung verlangt, dass der Server nach der Benachrichtigung aller aktiven Clients zunächst auf deren Empfangsbestätigung wartet, bevor die Antwort an den Initiator erfolgt.

Funktionen des Clients

Die Client-Anwendung soll einem Anwender ermöglichen, am WebSocket-Chat teilzunehmen. Dazu muss die Client-Anwendung über eine grafische Oberfläche verfügen, über die der Anwender sich anmelden, Nachrichten versenden und sich abmelden kann. Für die Anmeldung muss der Anwender einen Benutzernamen eingeben können. Existiert dieser Name bereits in der Chat-Anwendung, so wird dem Anwender die Anmeldung verweigert und er wird darüber benachrichtigt. Der Anwender muss anschließend in der Lage sein, die Anmeldung mit einem neuen Benutzernamen zu wiederholen. Nach einer erfolgreichen Anmeldung müssen Anwender untereinander Nachrichten austauschen können. Sie sollen dabei jederzeit die Benutzernamen aller angemeldeten Anwender sehen können. Alle Nachrichten, die seit dem Zeitpunkt der Anmeldung eines Anwenders gesendet wurden, sollen für diesen einsehbar sein. Die Abmeldung von der Chat-Anwendung muss über eine Schaltfläche möglich sein.

Beim Start der Anwendung befindet sich der Client im Zustand **UNREGISTERED**. Ausschließlich Clients mit diesem Status sollen eine Anmeldeanforderung angezeigt bekommen. Durch die Anmeldung wechselt der Client in den Zustand **REGISTERING** und insofern diese erfolgreich ist in **REGISTERED**. Durch diesen Status erhält der Anwender Zugriff auf den Chat. Meldet sich der Anwender ab, so wechselt der Client in den Zustand **UNREGISTERING** und wartet auf die Bestätigung durch den Server. Beim Erhalt dieser Bestätigung oder nach einer vordefinierten Wartezeit wechselt der Client in den Status **UNREGISTERED**. Der Client muss in der Lage sein, den eigenen Zustand zu speichern und zu ändern. Die Änderung des Status wird entweder durch eine Aktion des Anwenders ausgelöst oder durch eine Antwort des Servers.

Funktionen des Benchmark-Clients

Der Benchmark-Client nutzt die Funktionsweise der Client-Anwendung. Für die Leistungsmessung soll er eine große Anzahl simultan agierender Clients bereitstellen. Eine grafische Oberfläche wird dazu nicht benötigt. Der Benchmark-Client muss in der Lage sein, die Messergebnisse aus Abbildung A.1 zu erheben. Die Eingabeparameter des bestehenden Benchmark-Clients müssen als Konsolenparameter abgebildet werden. Folgende Parameter müssen konfigurierbar sein:

- Implementierungsart (Simple/Advanced)
- Serveradresse (IP-Adresse oder Hostname)

- Anzahl der Clients
- Anzahl der Nachrichten pro Client
- Länge einer Nachricht in Byte
- Wartezeit eines Clients vor dem Senden der nächsten Nachricht

4.2.2. Nicht-funktionale Anforderungen

Betrifft eine Anforderung mehrere oder alle funktionalen Anforderungen oder überschneidet sie, so handelt es sich um eine nicht-funktionale Anforderung [BL11, S. 109]. Für viele Anforderungen dieser Art lassen sich Qualitätskriterien definieren. Einer Studie von Forschern der University of Technology Sydney zufolge sind die am häufigsten berücksichtigten Qualitätsmerkmale Leistung (performance) (89 %), Zuverlässigkeit (reliability) (68 %), Benutzbarkeit (usability) (62 %), Sicherheit (security) (60 %) und Wartbarkeit (maintainability) (55 %) [MZN10, S. 314]. Die nicht-funktionalen Anforderungen an den WebSocket-Chat werden im folgenden Abschnitt diesen Kategorien zugeteilt. Da die zu entwickelnde Anwendung im Rahmen einer Lehrveranstaltung zu didaktischen Zwecken eingesetzt werden soll, werden Sicherheitsaspekte jedoch nicht betrachtet.

Leistung

Anforderungen bezüglich der Leistung einer Anwendung spezifizieren die Fähigkeit der Software, eine angemessene Geschwindigkeit im Verhältnis zu den eingesetzten Ressourcen zu gewährleisten [MZN10, S. 315]. So soll die Oberfläche der Anwendung beim Senden einer Nachricht nicht blockieren, sondern weitere Aufgaben bearbeiten können. Die Messergebnisse des Leistungstests des WebSocket-Chats sollen maximal um das 2,5-Fache über den Werten der bestehenden Chat-Anwendung liegen. Dieser Faktor wird gewählt, da Forschungsergebnisse zeigen, dass die Kommunikation über WebSockets im Vergleich zu TCP einen Overhead von 150-250 % hat [SHS14, S. 1007].

Zuverlässigkeit

Anforderungen bezüglich der Zuverlässigkeit einer Anwendung spezifizieren die Fähigkeit der Software, ein definiertes Leistungsniveau unter regulären Bedingungen für einen vorgegebenen Zeitraum fehlerfrei einzuhalten [MZN10, S. 315]. Der WebSocket-Chat soll im Falle einer abgebrochenen Clientverbindung weiterhin für alle anderen Clients funktionieren. Der Client, dessen Verbindung fehlerhaft beendet wurde, wird aus der aktiven Chat-Sitzung entfernt und soll sich anschließend erneut anmelden können.

Benutzbarkeit

Anforderungen bezüglich der Benutzbarkeit einer Anwendung betreffen die Interaktionen eines Benutzers mit dem System. Dies betrifft auch die Erlernbarkeit und Bedienbarkeit des Systems [MZN10, S. 315]. Der WebSocket-Chatclient soll durch die Verwendung von Angular 4 als *Single-Page-Webanwendung* entwickelt werden. Eine solche Webanwendung besteht aus einer einzigen

4. Die Chat-Anwendung

HTML-Seite, deren Inhalte dynamisch nachgeladen werden, sodass die Notwendigkeit von Unterseiten entfällt. Des Weiteren soll durch eine moderne Benutzeroberfläche sichergestellt werden, dass Benutzer intuitiv im System navigieren können. Die passende dynamische Skalierung für mobile Endgeräte soll ebenfalls gewährleistet sein.

Wartbarkeit

Anforderungen bezüglich der Wartbarkeit einer Anwendung spezifizieren, in welchem Maße der Quellcode einer Software verständlich, korrigierbar und verbesserbar ist [MZN10, S. 315]. Der Quellcode des WebSocket-Chats ist in ausreichendem Maß zu dokumentieren. Es soll sichergestellt werden, dass Studenten des Fachbereichs (Wirtschafts-)Informatik nach einer angemessenen Einarbeitungszeit in der Lage sind, den Quellcode zu verstehen und zu erweitern. Da die Erweiterung der internen Abläufe im Fokus der Lehrveranstaltung steht, sollen die Klassen der jeweiligen Implementierungsform (**Simple** und **Advanced**) getrennt voneinander realisiert werden.

4.3. Entwurf und Implementierung des WebSocket-Chats

Der *Entwurf* ist der Prozess, in dem die Architektur, Komponenten, Schnittstellen sowie weitere Eigenschaften eines Systems hinsichtlich spezifizierter Anforderungen definiert werden [Ins90, S. 25]. Der Entwurf wird für die Implementierung der Anwendung benötigt und dokumentiert anhand formeller Darstellungen die Architektur der Software.

Die *Implementierung* beschreibt den Prozess, in dem der Entwurf in die Komponenten einer Software umgesetzt wird [Ins90, S. 38]. Dabei werden die abstrakten Modelle durch die Verwendung geeigneter Datenstrukturen und Algorithmen in eine konkrete Programmiersprache übersetzt. Die Entscheidungen, die während der Implementierungsphase getroffen werden, müssen dokumentiert werden.

4.3.1. Entwurf

Neben dem Entwurf der internen Software-Architektur müssen auch die physische Architektur und die Distribution auf der technischen Infrastruktur beachtet werden [BL11, S. 6]. Der zu entwickelnde WebSocket-Chat wird nicht produktiv eingesetzt. Das bedeutet, dass keine Archivierung von Nachrichten oder eine anderweitige Persistierung von Zuständen der Anwendung erfolgt. Die Software-Architektur des WebSocket-Chats kann daher vereinfacht werden.

Die Server-Anwendung besteht lediglich aus einem funktionalen Teil, der die Anfragen der Clients bearbeitet. Die Client-Anwendung im Browser des Anwenders verfügt zusätzlich über einen Darstellungsteil in Form von HTML-Seiten. Der Anwendungsteil zur Durchführung der Lasttests soll als Konsolenanwendung ohne grafische Oberfläche zur Verfügung stehen.

Der WebSocket-Chat wird als Client-Server-System entworfen. Die einzelnen Anwendungsteile lassen sich dabei auf verschiedenen Hostsystemen ausführen. Als Webanwendung besteht zudem

4. Die Chat-Anwendung

die Möglichkeit, die Client-Anwendung durch den Webserver bereitstellen zu lassen, der auch die Server-Anwendung ausführt. Die URI des WebSocket-Serverendpunkts wird in diesem Fall relativ zum Kontextpfad der Client-Anwendung angegeben.

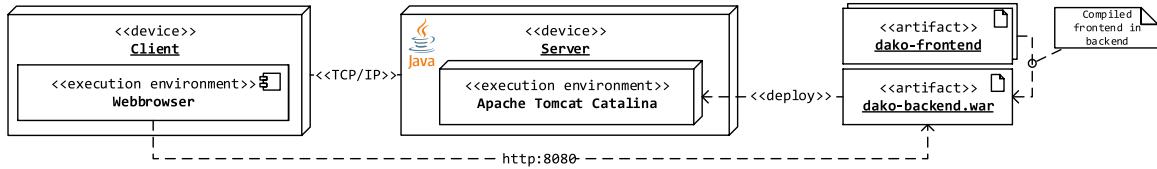


Abbildung 4.4: Bereitstellung von Front- und Back-End durch Apache Tomcat Catalina

Diese Flexibilität wird durch den Einsatz von Java und dem TypeScript-Framework Angular 4 gewährleistet. Die Server-Anwendung wird als *Web-Application-Archive-(WAR)-Servlet* von einem Webserver bereitgestellt, der die Spezifikation für Java-Servlets implementiert und die Java-WebSockets-API unterstützt. Der Servlet-Container *Catalina* des *Apache Tomcat* stellt diese Funktionalitäten ab Version 7.0.56 zur Verfügung.

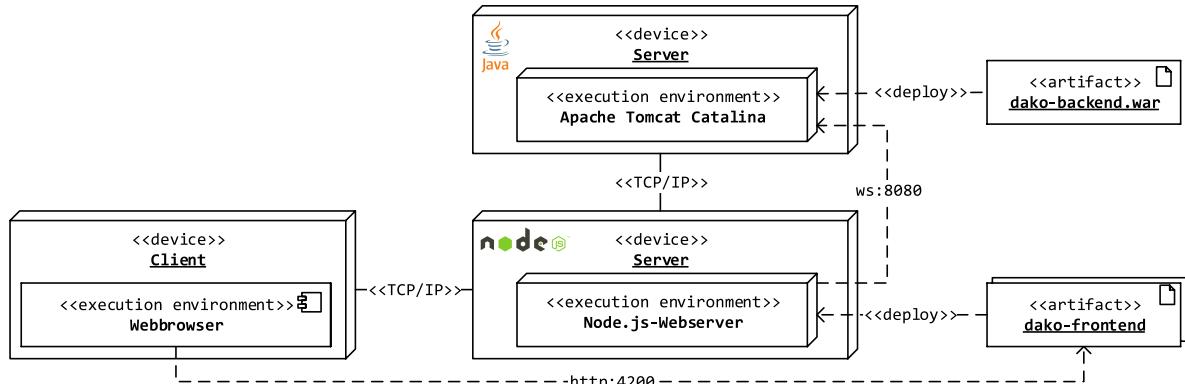


Abbildung 4.5: Bereitstellung des Front-Ends durch Node.js-Webserver

Der Einsatz von Angular 4 ermöglicht die dedizierte Bereitstellung des kompilierten Front-Ends auf einem *Node.js*-Webserver. *Node.js* ist ein JavaScript-Framework, das auf der Chrome-V8-JavaScript-Engine basiert und die Entwicklung von serverseitigen JavaScript-Anwendungen ermöglicht [Nod18]. Da Client- und Server-Anwendung auf dedizierten Hostsystemen ausgeführt werden, muss der Client-Anwendung die vollständige URI des Serverendpunkts bekannt sein.

Da der WebSocket-Chatserver ohne grafische Benutzeroberfläche implementiert werden soll und die zusätzlichen Klassen der TCP-Verbindungssteuerung entfallen, reduziert sich der Umfang in den Packages deutlich. Ein detailliertes Klassendiagramm des WebSocket-Chatservers ist in den Abbildungen A.2 und A.3 im Anhang zu finden. Der WebSocket-Benchmarkclient verfügt ebenfalls über das Package `edu.hm.dako.chat.common` und dessen Inhalt. Auf die formelle Darstel-

4. Die Chat-Anwendung

lung der `Main`-Klasse wurde verzichtet. Das Klassendiagramm des WebSocket-Benchmarkclients ist in Abbildung A.4 dargestellt.

Die Architektur des WebSocket-Chatclients wird durch die Verwendung des Angular Frameworks um zusätzliche Elemente erweitert, die in Java Anwendungen nicht zu finden sind. Dabei handelt es sich um Komponenten (*Components*) mit einem Anzegebereich (*View*) und *Services*. Jede Angular-Anwendung setzt sich aus verschiedenen Komponenten eines bestimmten Aufgabenbereichs zusammen. Sie sind hierarchisch aufgebaut, wobei alle Komponenten einer Hauptkomponente (*Root Component*) untergeordnet sind. Der Anzegebereich einer Komponente dient der Darstellung eines HTML-*Templates*, mit dem der Benutzer interagieren kann [Woi+17, S. 66].

Services sind Klassen, die ihre Funktionen für andere Klassen bereitstellen. Sie enthalten mehrfach verwendbaren Code und werden über das Entwurfsmuster *Dependency Injection* (DI) bereitgestellt [Woi+17, S. 117]. DI folgt dem Prinzip der *Inversion of Control* (IoC), das besagt, dass die Erzeugung einer Abhängigkeit nicht von dem abhängigen Objekt durchgeführt wird, sondern alle Abhängigkeiten an einem zentralen Ort (DI-Container) verwaltet werden. Der Aufbau des WebSocket-Chatclients ist in Abbildung A.5 im Anhang dargestellt. Die Klassen des *common*-Packages sind nicht enthalten, da ihre Darstellung mit denjenigen in Abbildung A.2 identisch ist.

4.3.2. Implementierung

Die Implementierung aller Java-Module wird in der Entwicklungsumgebung *Eclipse Java EE IDE for Web Developers* in Version Oxygen.2 Release (4.7.2) durchgeführt. Die installierte Version des *Java Development Kits* (JDK) ist 1.8.0_144. Zur Verwaltung der Abhängigkeiten wird Apache Maven 3.5.0 eingesetzt. Die Entwicklung des WebSocket-Chatclients erfolgt in *Visual Studio Code* Version 1.18 und *TypeScript* 2.6.2. Für die Nutzung des Angular-4-Frameworks wird die Paketverwaltung *npm* in Version 3.10.10 eingesetzt. Die Entwicklung wird auf dem Betriebssystem *macOS High Sierra* 10.13.3 durchgeführt. Die Versionierung der Codebase erfolgt durch *git* unter Verwendung der gleichnamigen Plattform von *Github, Inc.*

Zur Umsetzung der Architektur des WebSocket-Chatservers wird ein Maven-Projekt verwendet, um alle benötigten Bibliotheken bereitzustellen. Neben den Bibliotheken, die aus der bestehenden Anwendung übernommen wurden, werden die Java-WebSocket-API sowie Bibliotheken zur Transformation von Java-Objekten in das *JSON*-Format benötigt. Dieses dient der leichtgewichtigen textuellen Repräsentation von Objekten in einer lesbaren Form. Die `ChatPDUs` werden in diesem Format zwischen den Endpunkten übertragen.

4. Die Chat-Anwendung

```
@ServerEndpoint(value = "/advancedchat", encoders = { ChatPDUEncoder.class },
    decoders = { ChatPDUDecoder.class })
public class AdvancedWebSocketServer extends AbstractWebSocketServer { ... }
```

Listing 4.1: AdvancedWebSocketServer-Annotation

Durch die Deklaration der Encoder- und Decoder-Klassen des Serverendpunkts erfolgt die Umwandlung der ChatPDUs in JSON-Objekte automatisch anhand des Vorgehens, das in den jeweiligen Methoden definiert wurde. Die `@OnClose`-Methode muss speziell für den Fall angepasst werden, dass Clients die Verbindung durch das Beenden des Browsers oder das Verlassen der Seite schließen. Der Client wird bei Bedarf aus noch vorhandenen Wartelisten und der Liste der angemeldeten Clients entfernt, da jeder weitere Versuch eine Nachricht an diesen Client zu senden mit einer Exception (*broken pipe*) abgebrochen wird.

```
@Override
public void sendPduToClient(ClientListEntry client, ChatPDU pdu) {
    try {
        Session session = client.getSession();
        session.getBasicRemote().sendObject(pdu);
    } catch (IOException | EncodeException e) {
        log.error(e.getMessage());
    }
}
```

Listing 4.2: Senden einer ChatPDU

Das Senden einer ChatPDU erfolgt über die zugehörige `Session` eines Clients. Die Übertragung findet synchron an das `BasicRemote`-Objekt statt, da die asynchrone Alternative bei Nachrichten von geringer Größe keinen Vorteil bietet und die zusätzliche Verwaltung von Future-Objekten oder Callback-Methoden sowohl die Komplexität als auch den Verwaltungsaufwand erhöht. Da zwischen den verschiedenen Implementierungsformen eine Trennung erfolgen soll, werden zwei Endpunkte auf unterschiedlichen URIs⁵ simultan bereitgestellt. Die Wahl der zu verwendenden Implementierung erfolgt durch die Client-Anwendung.

Die Realisierung des Benchmark-Clients setzt die Verwendung einer Bibliothek voraus, die die Java-WebSocket-API implementiert. Dazu wird die Referenzimplementierung *Project Tyrus* der Oracle Corporation eingesetzt. Die Anzahl gleichzeitiger Chatclients sollte nebenläufig durch Threads abgebildet werden. Jedoch werden durch die Bibliothek pro Client drei Threads erzeugt, wohingegen in der bestehenden Chat-Anwendung nur zwei Threads pro Client eingesetzt wurden. Trotz zusätzlicher Synchronisierungsmechanismen war es nicht möglich die Funktionalität zu gewährleisten. Die Leistungsmessung wird aus diesem Grund in mehreren Prozessen ausgeführt. Die Abläufe und Funktionen des bestehenden Benchmark-Clients wurden übernommen.

⁵/simplechat und /advancedchat

4. Die Chat-Anwendung

Die Basisstruktur der Client-Anwendung wird mit Hilfe der Angular-CLI erstellt. Für die Nutzung von WebSockets in einer Angular-Anwendung wird die Bibliothek *RxJS* benötigt. RxJS ist der JavaScript-Teil des Open-Source-Projekts *ReactiveX.io*, das den Einsatz asynchroner Programmierung und des **Observable**-Entwurfsmusters erleichtert. Sie bietet zudem eine Unterstützung für WebSockets. Die API wird durch `npm install rxjs` dem Projekt hinzugefügt und im Service `websocket.service.ts` importiert.

```
import * as Rx from 'rxjs/Rx';

@Injectable()
export class WebsocketService {
    private subject: Rx.Subject<MessageEvent>;
    (...)

    private create(url): Rx.Subject<MessageEvent> {
        let webSocket = new WebSocket(url);
        let observable = Rx.Observable.create(
            (obs: Rx.Observer<MessageEvent>) => {
                webSocket.onmessage = obs.next.bind(obs);
                webSocket.onerror = obs.error.bind(obs);
                webSocket.onclose = obs.complete.bind(obs);
                return webSocket.close.bind(webSocket);
            })
        let observer = { next: (data: Object) => {
            if (webSocket.readyState === WebSocket.OPEN) {
                webSocket.send(JSON.stringify(data));
            }
        }}
        return Rx.Subject.create(observer, observable);
    } (...)
```

Listing 4.3: WebSocket-Client: WebSocket-Service

Dieser Service stellt zur Anbindung der bidirektional kommunizierenden WebSockets ein **Subject**-Objekt bereit, das Events empfangen und senden kann. Dazu wird sowohl ein **Observable**, das Events an mehrere Abonnenten senden kann, als auch ein **Observer** benötigt, der Daten empfängt. Ein Subject kann demnach sowohl eingesetzt werden, um eingehende Events von einem WebSocket an die Benutzeroberfläche weiterzuleiten als auch um Daten von dieser an den WebSocket zu übergeben. Dazu wird ein Observable-Objekt erstellt und die Callback-Methoden der abonnierenden Observer werden an die jeweiligen Event-Handler des WebSockets gebunden. Der Rückgabewert gibt an, dass die WebSocket-Verbindung geschlossen wird, wenn der abonnierende Observer sich abmeldet. Der Observer-Teil des Subjects definiert lediglich die Methode `next`, die dem Observer mitteilt, dass neue Daten vorliegen. In dieser wird zunächst geprüft, ob der zugrundeliegende WebSocket geöffnet ist, bevor die übergebenen Daten in einen JSON-String transformiert und versendet werden. Der WebSocket-Service wird vom Chat-Service verwendet. Dieser wandelt eingehende JSON-Strings in ChatPDUs und übernimmt Funktionen wie Verbindungsauflaufbau und -abbau.

4. Die Chat-Anwendung

Der Ablauf und die Algorithmen des WebSocket-Chats sind in der Klasse `AbstractChatClient` bzw. in deren konkreten Ausprägungen implementiert. Diese werden von den Komponenten App, Chat und Login verwendet. Die `AppComponent` ist die Hauptkomponente, von der aus die anderen gesteuert werden. Beim Aufruf der Website wird diese zuerst geladen und liefert die `LoginComponent`, falls der Benutzer noch nicht angemeldet ist. Andernfalls wird die `ChatComponent` bereitgestellt.

```
<div *ngIf="!loggedIn; then thenBlock else elseBlock"></div>
<ng-template #thenBlock>
  <app-login (clientEvent)="receiveClientService($event)"></app-login>
</ng-template>
<ng-template #elseBlock>
  <app-chat [client]="client" (closedEvent)="receiveClosedEvent($event)">
  </app-chat>
</ng-template>
```

Listing 4.4: WebSocket-Client: AppComponent-View

Der Austausch von Objekten zwischen den Komponenten erfolgt entweder durch die Übergabe der übergeordneten Komponente (*Parent*) an die aufgerufene Komponente (*Child*) oder durch `EventEmitter`. Hier empfängt die `AppComponent` ein Objekt des `AbstractChatClient`-Services durch einen `EventEmitter` der `LoginComponent`. Dieses Event wird erst bei einer erfolgreichen Anmeldung ausgelöst. Der Benutzername des angemeldeten Clients wird dabei im *SessionStorage* gespeichert. Liegt hier ein Wert vor, so gilt der hinterlegte Benutzer als angemeldet und die `ChatComponent` wird bereitgestellt. Diese empfängt den zu verwendenden Client-Service von der `AppComponent`. Meldet sich ein Client ab oder lädt die Webseite neu, so wird der Benutzername aus dem SessionStorage entfernt und der Client muss sich erneut anmelden. Da im Server entsprechende Vorkehrungen getroffen wurden, kann sich der Client unverzüglich erneut anmelden. Kann der Benutzer nicht angemeldet werden, wird eine Fehlermeldung über dem Login-Bereich ausgegeben. Das Design des WebSocket-Chatclients ist responsive, das bedeutet, dass die Darstellung an die Größe des Displays des verwendeten Endgeräts angepasst wird. Somit ist der WebSocket-Chat sowohl von einem Smartphone als auch von einem hochauflösenden Computerbildschirm bedienbar. Dazu wird das CSS-Framework *Semantic UI* eingebunden.

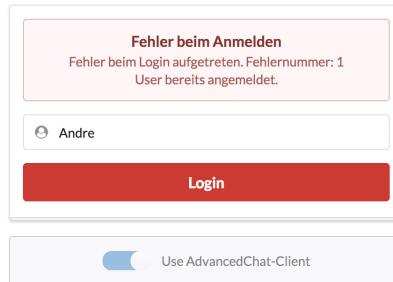


Abbildung 4.6: WebSocket-Client: Fehlermeldung bei Login

4. Die Chat-Anwendung

Im unteren Bereich des Login-Views wird angezeigt, welche Implementierungsform die Client-Services nutzen. Diese Einstellung kann nicht zur Laufzeit verändert, sondern muss in der Konfiguration der Umgebung angepasst werden. Die zugehörigen Konfigurationsdateien lauten `environment.prod.ts` für die Einstellungen in der Produktivumgebung und `environment.ts` für Testumgebungen. Als Konfiguration für den Einsatz als eigenständige Anwendung dient die Datei `environment.ts`. Hier müssen neben der Implementierungsform auch die IP-Adresse bzw. der Hostname des WebSocket-Servers hinterlegt werden. Soll der WebSocket-Client hingegen ebenfalls von der Server-Anwendung ausgeliefert werden, so wird die Produktivkonfiguration verwendet. Dazu muss die durch den Befehl `ng build -prod -base-href /NAME DES JAVA PROJEKTS/` kompilierte Client-Anwendung in den `webapp`-Ordner des WebSocket-Serverprojekts verschoben werden.

Abschließend lässt sich feststellen, dass die definierten funktionalen Anforderungen ausnahmslos erfüllt wurden. Die bestehende Funktionalität der TCP-Chat-Anwendung konnte vollständig übernommen werden. Die Übertragung zwischen den Anwendungsteilen basiert nun auf HTML5 WebSockets. Bei der Umsetzung der nicht-funktionalen Anforderungen konnten die definierten Kriterien der Kategorien Zuverlässigkeit, Benutzbarkeit und Wartbarkeit erfüllt werden. Die Kriterien zur Leistung des WebSocket-Chats werden im nachfolgenden Kapitel geprüft. Diese konnten nicht erfüllt werden.

5. Evaluation

Die Durchführung der Leistungsmessung soll einen Vergleich zwischen der bestehenden Chat-Anwendung und dem entwickelten WebSocket-Chat ermöglichen. Dazu sollen die Implementierungen der jeweiligen Benchmark-Clients eingesetzt werden. Zur Realisierung einer geeigneten Umgebung wird ein dediziertes Netzwerk verwendet, in dem die Clients und der Server ausgeführt werden können. Der Aufbau des Netzwerks ist in Abbildung 5.1 dargestellt.

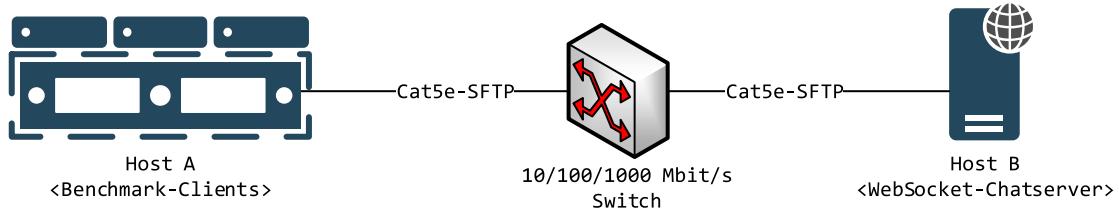


Abbildung 5.1: Netzwerk der Test-Konfiguration

Der WebSocket-Chatserver befindet sich auf einem dedizierten Host, der die Anwendung mittels des Webservers *Apache Tomcat* (Version 9.0.0.M26) bereitstellt. Die Benchmark-Clients werden als virtuelle Maschinen auf einem Hostsystem ausgeführt. Es kommen drei dieser Instanzen zum Einsatz. Die physikalischen Hosts verfügen jeweils über eine 1000 Mbit/s-Verbindung. Die technische Konfiguration der Testsysteme ist in Tabelle 5.1 abgebildet.

| | Benchmark-VM | WebSocket-Chatserver |
|-----------------------|---|----------------------|
| OS | Ubuntu 16.04 LTS | macOS HighSierra |
| CPU | 1 Core | Intel Core i5-7360U |
| RAM | 4GB | 8GB |
| Virtualization | VMWare Workstation Pro 12 | - |
| Hypervisor | Typ2 (OS: Windows 10 x64, CPU: Intel Core i5-4460, RAM: 16GB) | - |

Tabelle 5.1: Konfiguration der Test-Umgebung

Die Messungen der bestehenden Anwendung werden in der gleichen Umgebung durchgeführt. Jedoch wird hier nur eine der virtuellen Maschinen für die Client-Anwendung eingesetzt. Die IP-Adressen der Systeme werden manuell konfiguriert. Die Server-Anwendungen sind über die TCP-Ports 8080 (WebSocket-Chatserver) bzw. 50000 (TCP-Chatserver) erreichbar.

5.1. Komplikationen

Die Anwendungen sollten sich lediglich durch die Verwendung des Übertragungsprotokolls unterscheiden, um die Vergleichbarkeit der Messergebnisse zu gewährleisten. Die Simulation mehrerer gleichzeitiger Benutzer soll nebenläufig durch Threads erfolgen. Dafür wurde zur Realisierung des Benchmark-Clients die Bibliothek *Tyrus 1.12* eingesetzt. Diese verwendet für jede Verbindung eines WebSocket-Clients drei Threads, wohingegen die ursprüngliche Implementierung die Verwendung von zwei Threads vorsieht. Trotz zusätzlicher Synchronisierungsmechanismen konnte die fehlerfreie Ausführung der WebSocket-Leistungsmessungen nicht gewährleistet werden. Aus diesem Grund wird der WebSocket-Benchmark-Client in mehreren Prozessen gestartet. Der zusätzliche Aufwand zur Verwaltung der Prozesse durch das Betriebssystem wirkt sich negativ auf die Leistung des WebSocket-Chats aus. Durch die Verwendung einer anderen WebSocket-Bibliothek oder eines browserbasierten Benchmarkings könnte eine erhebliche Verbesserung der Messergebnisse erzielt werden.

Die Prozesse der WebSocket-Clients zur Leistungsmessung werden auf die drei virtuellen Maschinen verteilt. Damit diese in möglichst kurzen Zeitabständen starten können, wird deren Ausführung durch zwei Bash-Skripte gesteuert.

Das erste Skript trägt die Bezeichnung `start_benchmark.sh` und empfängt folgende Werte als Parameter:

| Parameter | Beschreibung | Beispiel |
|-------------------------------|---|--------------------|
| Adresse des WebSocket-Servers | Die Angabe muss den verwendeten Port und den Namen der Anwendung enthalten, unter der der Endpunkt bereitgestellt wird. | 10.0.0.1:8080/chat |
| Anzahl der Clients pro Host | Im Skript können die IP-Adressen der Systeme hinterlegt werden, die als Hosts für die Benchmark-Clients dienen. Es wird auf jedem Host die übergebene Anzahl Clients gestartet. | 10 (x3 = 30) |
| Anzahl Nachrichten | Die Anzahl der Nachrichten, die pro Client gesendet werden sollen. | 10 |
| Nachrichtenlänge | Die Länge einer Nachricht in Byte. | 10 |
| Denkzeit des Clients | Die Zeit in ms, die ein Client zwischen dem Senden von zwei Nachrichten wartet. | 100 |
| Benutzen des AdvancedChats | Anhand dieser Angabe wird der zu verwendende Implementierungstyp festgelegt. | y oder n |

Tabelle 5.2: Parameter des Benchmark-Skripts

5. Evaluation

Dieses Skript stellt bei seiner Ausführung eine SSH-Verbindung zu den im Skript hinterlegten Hosts her und führt dort das Skript `run_client_benchmark.sh` im Hintergrund aus. Hier findet der Aufruf des Benchmark-Clients statt, der ebenfalls als Hintergrund-Prozess gestartet wird. Der jeweils letzte Aufruf blockiert die weitere Ausführung, damit Skript- und Benchmark-Prozess zeitgleich enden. Der Benchmark-Client sammelt alle Messwerte und legt sie als JSON-String in einer Textdatei ab. Wurden alle Instanzen des Benchmark-Clients beendet, werden die Resultate der einzelnen Testläufe durch das Startskript gesammelt und von einer Java-Konsolenanwendung ausgewertet. Diese importiert die Informationen aller Clients und ermittelt die statistischen Ergebnisse des gesamten Durchlaufs.

```
{  
    "userName": "Client - Thread - 1", "implementation": "Advanced",  
    "totalEvents": 6, "sendMessages": 2, "loginEvents": 2, "logoutEvents": 2,  
    "chatPduRtt": [28270117128, 17249212499],  
    "serverTime": [25951704704, 17245024235]  
}
```

Listing 5.1: Resultat eines Benchmark-Clients

Die Leistungsmessung der bestehenden Chat-Anwendung kann jedoch nicht auf mehrere Hosts aufgeteilt werden, da jede Instanz die Benutzernamen der Clients gleich nummeriert. Das System verhindert jedoch die Anmeldung gleichnamiger Benutzer. Aus diesem Grund wird die Leistungsmessung der TCP-Chat-Anwendung auf nur einer virtuellen Maschine durchgeführt.

5.2. Leistungsanalyse des WebSocket-Chats

Die Metrik zur Messung der Leistung des WebSocket-Chats ist die *Round Trip Time* (RTT). Dabei handelt es sich um die Zeit, die zwischen dem Stellen einer Anfrage und dem Eintreffen der zugehörigen Antwort vergeht. Die Zeit, die der Server zur Bearbeitung der Anfrage benötigt, wird als *Server Time* bezeichnet. Die Dauer der reinen Kommunikation errechnet sich aus der Differenz zwischen RTT und Server Time ($T_k = RTT - \text{Server Time}$).

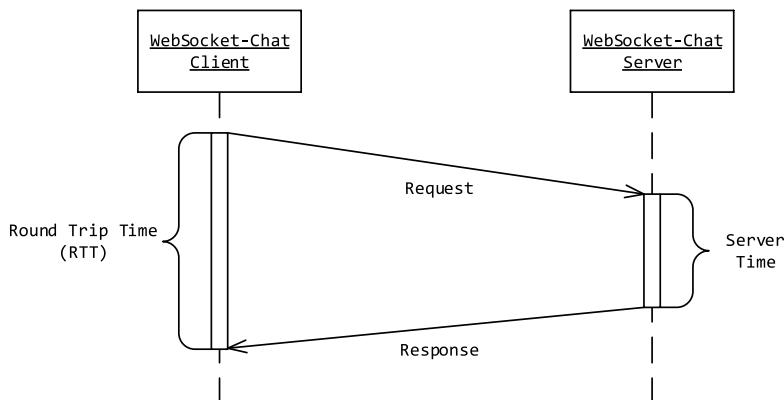


Abbildung 5.2: Metriken der Leistungsmessung des WebSocket-Chats

5. Evaluation

Für die Leistungsmessungen wird die Anzahl der aktiven Clients variiert. Jeder Client versendet zehn Nachrichten mit einer Länge von zehn Byte. Die Clients haben eine maximale Denkzeit von 100 ms. Jede Messung wird dabei fünf Mal wiederholt. Die Ergebnisse der Advanced-Benchmarks sind in den Tabellen A.1 und A.2 im Anhang zusammengefasst. Insbesondere bei den Zeitmessungen der Round Trip Time und der Serverzeit ist zu erkennen, dass die Messergebnisse des WebSocket-Chats die Werte des TCP-Chats signifikant übersteigen. Die Ursachen dieser Abweichungen sind zahlreich. Zum einen wirken sich die in Abschnitt 5.1 beschriebenen Unterschiede negativ auf die Leistungsfähigkeit des WebSocket-Chats aus, zum anderen liegen die Ursachen auch in der Verschiedenartigkeit der Protokolle selbst begründet.

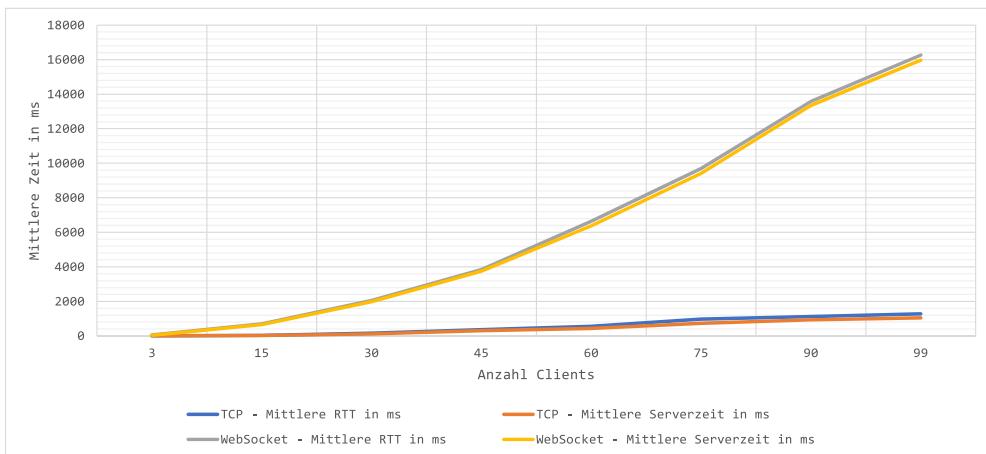


Abbildung 5.3: Messergebnisse RTT und Serverzeit



Abbildung 5.4: WebSocket-Overhead für reine Kommunikationsdauer

Das WebSocket-Protokoll agiert auf einer höheren Schicht des TCP/IP-Referenzmodells und benötigt zusätzliche Zeit und Rechenleistung für die Übertragung von der TCP-Schicht in die eigene Ebene. Zudem verwendet das WebSocket-Protokoll eventgesteuerte Callback-Mechanismen, die zusätzlichen Verwaltungsaufwand auf Anwendungsebene erfordern [SHS14, S. 1007]. Der zeitliche

5. Evaluation

Aufwand für die Maskierung clientseitiger Nachrichten kann nicht dediziert erfasst werden und erhöht ebenfalls die erfassten Messwerte. Darüber hinaus könnten auch die Java-Implementierungen der Protokolle ins Gewicht fallen. Das WebSocket-Protokoll wurde erst vor kurzer Zeit entwickelt und die verwendete API ist eine frühe Referenzimplementierung, wohingegen die Java-TCP-Sockets bereits seit Jahrzehnten eingesetzt wird und kontinuierlich optimiert wurde [SHS14, S. 1007].

Auch die Dauer der reinen Kommunikation des WebSocket-Chats übersteigt die des TCP-Chats, jedoch reduziert sich der Overhead (Berechnung anhand Gleichung 5.1) mit steigender Anzahl verbundener Clients.

$$O_{T_k} = \frac{T_{k(\text{WebSocket})} - T_{k(\text{TCP})}}{T_{k(\text{TCP})}} * 100\% \quad (5.1)$$

Ausschlaggebend für diese Entwicklung sind die unterschiedlichen Nachrichtengrößen und die Verwendung der Callback-Mechanismen. Die übrigen Faktoren wirken sich erst beim Einsatz mehrerer Clients wesentlich auf die reine Kommunikationszeit aus. Beispielsweise ist der zusätzliche Aufwand für Kontextwechsel bei nur drei Threads gering, bzw. bei nur einem Prozess muss kein Kontextwechsel durchgeführt werden.

Die bestehende Chat-Anwendung zeigt im direkten Leistungsvergleich einen signifikanten Geschwindigkeitsvorteil. Die Kommunikation über TCP ist leichtgewichtiger und schneller als über WebSockets. Zur besseren Vergleichbarkeit sollten jedoch weitere Anwendungen und Bibliotheken zur Leistungsmessung herangezogen werden. Beispielsweise birgt der Einsatz einer Java-WebSocket-Bibliothek, die eine Parallelisierung durch Threading ermöglicht, erhebliches Verbesserungspotential. Zudem sollten sich WebSocket-Leistungsmessungen auf Webtechnologien wie JavaScript oder Typescript fokussieren, da diese den primären Anwendungsfall für einen Client darstellen.

6. Fazit und Ausblick

In dieser Arbeit wurde die praktische Einsetzbarkeit von HTML5-WebSockets anhand der Migration der Kommunikationsstruktur einer auf TCP-Sockets basierenden Chat-Anwendung geprüft. Dazu wurden in Kapitel 2 zunächst die wesentlichen Aspekte der neuen Protokollspezifikation detailliert dargelegt, um daraus die Vorteile der Technologie gegenüber dem Request/Response-Verfahren von HTTP abzuleiten. Durch das WebSocket-Protokoll können Informationen zwischen Server und Client bidirektional ausgetauscht werden, ohne dass der Client eine neue Anfrage initiieren muss. Des Weiteren reduziert es den durch Zusatzinformationen verursachten Overhead erheblich, da dieser nach dem erfolgreichen Aufbau einer Verbindung lediglich 2-14 Byte beträgt. Die Implementierung des Protokolls durch die WebSocket-API des W3C und das entsprechende Java-Pendant wurden in Kapitel 3 vorgestellt.

Anhand der daraus gewonnenen Erkenntnisse wurde in Kapitel 4 die Überarbeitung der Chat-Anwendung geplant und umgesetzt. Hierzu wurde zunächst eine Anforderungsanalyse anhand der Funktionalität des Altsystems durchgeführt. Auf dieser Grundlage wurde die Architektur des WebSocket-Chats entworfen und anschließend implementiert. In Kapitel 5 werden die Leistung der WebSocket-Chat-Anwendung und die des Altsystems gegenübergestellt. Die Messergebnisse zeigen, dass der WebSocket-Chat bis zu 13-mal länger für die Bearbeitung einer Anfrage benötigt als der TCP-Chat bei gleicher Anzahl angemeldeter Benutzer. Diese Werte resultieren zum einen aus der Beziehung zwischen den Protokollen: Das WebSocket-Protokoll agiert auf der Anwendungsebene und setzt auf TCP auf. Die Übertragung zwischen den Schichten erfordert zusätzliche Zeit sowie Rechenleistung und führt folglich zu schlechteren Leistungswerten. Auch der Einsatz von Callback-Mechanismen und die clientseitige Maskierung wirken sich negativ auf die Gesamtleistung aus. Zum anderen werden die Ergebnisse von diversen Komplikationen bei der Umsetzung beeinflusst, die nicht im Zusammenhang mit den verwendeten Protokollen stehen.

Die Technologie der HTML5-WebSockets überzeugt sowohl durch die funktionale, leichtgewichtige Architektur des Protokolls als auch durch die komfortable Handhabung der zugehörigen APIs. Entwickler moderner Webanwendungen, die Informationen zwischen Server und Client bidirektional austauschen sollen, verfügen dadurch über einen standardisierten Weg zur Umsetzung dieser Anforderung. Die Unterstützung von WebSockets ist inzwischen durch nahezu alle aktuellen Webbrower und -server gegeben. Dennoch kann aus Leistungsgründen von der Migration bestehender Systeme abgeraten werden, insofern deren Kommunikation auf TCP basiert und innerhalb des eigenen Netzwerks stattfindet. Jedoch werden, mit zunehmender Akzeptanz und Verbreitung, diverse Erweiterungen und Optimierungen das WebSocket-Protokoll verbessern. Daher sollten weitere Untersuchungen die Leistungsfähigkeit der unterschiedlichen WebSocket-Bibliotheken einer Programmiersprache vergleichen. Zudem sollten die WebSocket-Implementierungen unterschiedlicher Programmiersprachen analysiert und bewertet werden.

Literatur

- [BL11] Helmut Balzert und Peter Liggesmeyer. *Lehrbuch der Softwaretechnik. 2: Entwurf, Implementierung, Installation und Betrieb.* 3. Aufl. Lehrbücher der Informatik. OCLC: 750951360. Heidelberg: Spektrum, Akad. Verl, 2011. 596 S. ISBN: 978-3-8274-1706-0.
- [Cab17] Pierre Cabonnele. *PYPL PopularitY of Programming Language Index.* 1. Dez. 2017. URL: <http://pypl.github.io/PYPL.html> (besucht am 31.12.2017).
- [Cow14] Danny Coward. *Java WebSocket Programming.* New York: McGraw-Hill, 2014. 230 S. ISBN: 978-0-07-182719-5.
- [Fet10] Ian Fette. *The WebSocket Protocol.* 31. Aug. 2010. URL: <https://tools.ietf.org/html/draft-ietf-hybi-thewebsOCKETprotocol-01> (besucht am 19.12.2017).
- [FM11] Ian Fette und Alexey Melnikov. *The WebSocket Protocol.* 1. Dez. 2011. URL: <https://tools.ietf.org/html/rfc6455> (besucht am 19.12.2017).
- [GLN15] Peter Leo Gorski, Luigi Lo Iacono und Hoai Viet Nguyen. *WebSockets: moderne HTML5-Echtzeitanwendungen entwickeln.* OCLC: 901007083. München: Hanser, 2015. 269 S. ISBN: 978-3-446-44438-6.
- [Hic09a] Ian Hickson. *Server-Sent Events.* 23. Apr. 2009. URL: <https://www.w3.org/TR/2009/WD-eventsource-20090423/> (besucht am 28.12.2017).
- [Hic09b] Ian Hickson. *The Web Sockets API.* 23. Apr. 2009. URL: <https://www.w3.org/TR/2009/WD-websockets-20090423/> (besucht am 30.12.2017).
- [Hic10] Ian Hickson. *The WebSocket Protocol.* 6. Mai 2010. URL: <https://tools.ietf.org/search/draft-hixie-thewebsOCKETprotocol-76> (besucht am 19.12.2017).
- [Hoc17] Hochschule München. *Modulhandbuch IB 2017.* 1. Okt. 2017. URL: <https://w3.o.cs.hm.edu:8000/public/module/59/> (besucht am 14.01.2018).
- [Hua+11] Lin-Shung Huang u.a. „Talking to Yourself for Fun and Profit“. In: *Proceedings of W2SP* (1. Jan. 2011), S. 1–11.
- [Ins90] Institute of Electrical and Electronics Engineers. *IEEE 620.12-1990.* 28. Sep. 1990.
- [Int11] Internet Assigned Numbers Authority. *WebSocket Protocol Registries.* 21. Okt. 2011. URL: <https://www.iana.org/assignments/websocket/websocket.xhtml#opcode> (besucht am 25.12.2017).
- [Kre06] Heinz Kredel. *JavaPackages.* 23. Jan. 2006. URL: <http://krum.rz.uni-mannheim.de/pk1/sess-606.html> (besucht am 14.01.2018).
- [Lea+99] Paul J. Leach u.a. *Hypertext Transfer Protocol – HTTP/1.1.* 1. Juni 1999. URL: <https://tools.ietf.org/html/rfc2616> (besucht am 20.12.2017).
- [Lee11] Trustin Lee. *Netty.News: WebSockets in Netty.* 17. Nov. 2011. URL: <https://netty.io/news/2011/11/17/websockets.html> (besucht am 04.01.2018).

Literatur

- [Man17a] Peter Mandl. *Aufgabenstellung Studienarbeit Datenkommunikation WS1718*. 1. Sep. 2017. URL: https://www.wirtschaftsinformatik-muenchen.de/wp-content/uploads/Peter%20Mandl/Lehrveranstaltungen/WiSe%2017-18/Datenkommunikation/Aufgabenstellung_Studienarbeit_Datenkommunikation_WS_17_18.pdf (besucht am 25.01.2018).
- [Man17b] Peter Mandl. *Datenkommunikation / Competence Center Wirtschaftsinformatik (CCWI)*. 1. Okt. 2017. URL: https://www.wirtschaftsinformatik-muenchen.de/?page_id=595 (besucht am 14.01.2018).
- [Mel12] Alexey Melnikov. *Additional WebSocket Close Error Codes*. 16. Mai 2012. URL: <http://www.ietf.org/mail-archive/web/hybi/current/msg09670.html> (besucht am 25.12.2017).
- [MZN10] Dewi Mairiza, Didar Zowghi und Nurie Nurmuliani. „An Investigation into the Notion of Non-Functional Requirements“. In: *Proceedings of the 2010 ACM Symposium on Applied Computing*. ACM, 2010, S. 311–317.
- [Nod18] Node.js Foundation. *Node.Js*. 1. Jan. 2018. URL: <https://nodejs.org/en/> (besucht am 30.01.2018).
- [Not+12] Mark Nottingham u. a. *URI Template*. 1. März 2012. URL: <https://tools.ietf.org/html/rfc6570> (besucht am 13.01.2018).
- [Oak99] Chris Oakes. *Interview with the Web's Creator*. 23. Okt. 1999. URL: <https://www.wired.com/1999/10/interview-with-the-webs-creator/> (besucht am 09.02.2018).
- [Ora14a] Oracle Corporation. *GlassFish Server Open Source Edition Release Notes 4.1*. 1. Sep. 2014. URL: <https://javaee.github.io/glassfish/doc/4.0/release-notes.pdf> (besucht am 04.01.2018).
- [Ora14b] Oracle Corporation. *JavaFX Overview (Release 8)*. 1. Aug. 2014. URL: <https://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-overview.htm> (besucht am 14.01.2018).
- [Ora17a] Oracle Corporation. *Project Grizzly - WebSockets Overview*. 6. Sep. 2017. URL: https://javaee.github.io/grizzly/websockets.html#/Overview_of_Grizzlys_WebSocket_Implementation (besucht am 04.01.2018).
- [Ora17b] Oracle Corporation. *The Java Community Process(SM) Program - Introduction - Timeline View*. 1. Jan. 2017. URL: <https://jcp.org/en/introduction/timeline> (besucht am 31.12.2017).
- [Pey17] Sebastian Peyrott. *A Brief History of JavaScript*. 16. Jan. 2017. URL: <https://auth0.com/blog/a-brief-history-of-javascript/> (besucht am 31.12.2017).

Literatur

- [PR09] Klaus Pohl und Chris Rupp. *Basiswissen Requirements Engineering: Aus- und Weiterbildung zum "Certified Professional for Requirements Engineering"; Foundation Level nach IREB-Standard*. 1. Aufl. OCLC: 319867958. Heidelberg: dpunkt-Verl, 2009. 172 S. ISBN: 978-3-89864-613-0.
- [Rus+07] Alex Russell u. a. *The CometD Reference Book – 3.1.3*. 1. Jan. 2007. URL: https://docs.cometd.org/current/reference/#_bayeux (besucht am 28.12.2017).
- [Rus06] Alex Russell. *Comet: Low Latency Data for the Browser – Infrequently Noted*. 3. März 2006. URL: <https://infrequently.org/2006/03/comet-low-latency-data-for-the-browser/> (besucht am 28.12.2017).
- [She17] Eric Shepherd. *Writing WebSocket Servers*. 16. Aug. 2017. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API/Writing_WebSocket_servers (besucht am 22.12.2017).
- [SHS14] D. Skvorc, M. Horvat und S. Srbljic. „Performance Evaluation of Websocket Protocol for Implementation of Full-Duplex Web Streams“. In: IEEE, Mai 2014, S. 1003–1008. ISBN: 978-953-233-077-9 978-953-233-081-6. DOI: 10.1109/MIPRO.2014.6859715. URL: <http://ieeexplore.ieee.org/document/6859715/> (besucht am 21.10.2017).
- [Smy09] Tamara Smyth. *Brief History of Java*. 1. März 2009. URL: https://www.cs.sfu.ca/~tamaras/javaIntro/Brief_History_Java.html (besucht am 31.12.2017).
- [The14] The Apache Foundation. *Apache Tomcat 7 (7.0.56) - Changelog*. 6. Okt. 2014. URL: <http://tomcat.apache.org/tomcat-7.0-doc/changelog.html> (besucht am 04.01.2018).
- [The16] The Eclipse Foundation. *Jetty - Servlet Engine and Http Server*. 1. Jan. 2016. URL: <http://www.eclipse.org/jetty/about.html> (besucht am 04.01.2018).
- [WHA06] WHATWG. *XMLHttpRequest Standard*. 1. Jan. 2006. URL: <https://xhr.spec.whatwg.org/> (besucht am 27.12.2017).
- [WHA10] WHATWG. *HTML Standard WebSockets*. 2010. URL: <https://html.spec.whatwg.org/multipage/web-sockets.html> (besucht am 19.12.2017).
- [Woi+17] Gregor Woivode u. a. *Angular: Grundlagen, fortgeschrittene Techniken und Best Practices mit TypeScript - ab Angular 4, inklusive NativeScript und Redux*. 1. Auflage. iX Edition. OCLC: 939112124. Heidelberg: dpunkt.verlag, 2017. 551 S. ISBN: 978-3-86490-357-1 978-3-96088-205-3 978-3-96088-206-0 978-3-96088-207-7.
- [Yos15] Takeshi Yoshino. *Compression Extensions for WebSocket*. 24. Aug. 2015. URL: <https://tools.ietf.org/html/draft-ietf-hybi-permessage-compression-28#section-4> (besucht am 26.12.2017).

Abbildungsverzeichnis

| | | |
|-------|--|----|
| 2.1. | WebSocket-Protokoll im ISO/OSI-Modell | 7 |
| 2.2. | Vereinfachte Darstellung eines WebSocket-Frames | 8 |
| 2.3. | Ping- und Pong-Frames zur Verbindungskontrolle | 13 |
| 2.4. | Close-Frame initiiert von Client | 13 |
| 2.5. | Close-Frame-Antwort des Servers | 13 |
| 2.6. | Opening-Handshake des Clients | 16 |
| 2.7. | Opening-Handshake des Servers | 16 |
| 2.8. | Datentransfer von Server zu Client | 17 |
| 2.9. | Clientseitiger Closing-Handshake | 18 |
| 2.10. | HTTP-Streaming im Bayeux-Protokoll | 20 |
| 3.1. | UML-Zustandsdiagramm des WebSocket-Lifecycle | 23 |
| 3.2. | JSR Approval Timeline | 25 |
| 4.1. | Zustände in der Chat-Anwendung | 43 |
| 4.2. | Ablauf des Login-Vorgangs (AdvancedChat) | 44 |
| 4.3. | Aufbau einer ChatPDU | 47 |
| 4.4. | Bereitstellung von Front-und Back-End durch Apache Tomcat Catalina | 51 |
| 4.5. | Bereitstellung des Front-Ends durch Node.js-Webserver | 51 |
| 4.6. | WebSocket-Client: Fehlermeldung bei Login | 55 |
| 5.1. | Netzwerk der Test-Konfiguration | 57 |
| 5.2. | Metriken der Leistungsmessung des WebSocket-Chats | 59 |
| 5.3. | Messergebnisse RTT und Serverzeit | 60 |
| 5.4. | WebSocket-Overhead für reine Kommunikationsdauer | 60 |

Tabellenverzeichnis

| | |
|--|------|
| 2.1. Maskierung durch bitweise XOR-Verknüpfung | 11 |
| 2.2. Demaskierung durch bitweise XOR-Verknüpfung | 12 |
| 2.3. Auszug der registrierten Status-Codes | 15 |
| 5.1. Konfiguration der Test-Umgebung | 57 |
| 5.2. Parameter des Benchmark-Skripts | 58 |
| A.1. Ergebnisse der WebSocket-Leistungsmessung | VIII |
| A.2. Ergebnisse der TCP-Leistungsmessung | VIII |

Listings

| | | |
|-------|---|----|
| 3.1. | WebSocket anlegen und verbinden | 22 |
| 3.2. | WebSocket-Lebenszyklus | 23 |
| 3.3. | WebSocket-Event-Handler | 24 |
| 3.4. | WebSocket-Nachrichten senden | 24 |
| 3.5. | Java: programmatischer Ansatz | 27 |
| 3.6. | Java: Annotationen | 27 |
| 3.7. | Java Client: Verbindungsauflbau | 27 |
| 3.8. | Java: Event-Handler | 28 |
| 3.9. | Java: Ping-/Pong-Nachrichten | 29 |
| 3.10. | Java: synchrone Nachrichtenübertragung | 30 |
| 3.11. | Java: Future und Handler | 30 |
| 3.12. | Java: Callback-Interface SendHandler | 31 |
| 3.13. | Java: Encoder-Interface | 32 |
| 3.14. | Java: Decoder-Interface | 32 |
| 3.15. | Java: Angabe der Encoder- und Decoder-Klassen | 33 |
| 3.16. | Java: EndpointConfig | 33 |
| 3.17. | Java: ClientEndpointConfig-Konfigurationsklasse | 34 |
| 3.18. | Java: ServerEndpointConfig-Konfigurationsklasse | 35 |
| 3.19. | Java:PathParam | 38 |
| 4.1. | AdvancedWebSocketServer-Annotation | 53 |
| 4.2. | Senden einer ChatPDU | 53 |
| 4.3. | WebSocket-Client: WebSocket-Service | 54 |
| 4.4. | WebSocket-Client: AppComponent-View | 55 |
| 5.1. | Resultat eines Benchmark-Clients | 59 |

A. Anhang

A. Anhang

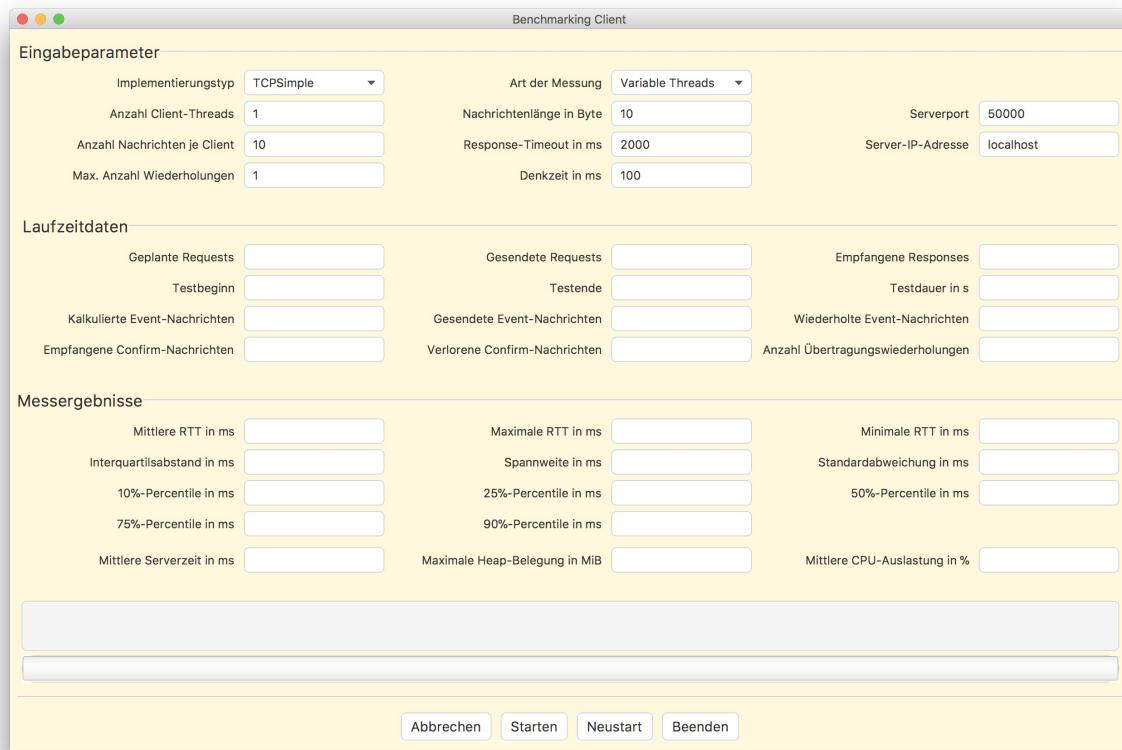


Abbildung A.1: Benutzeroberfläche des Benchmark-Clients

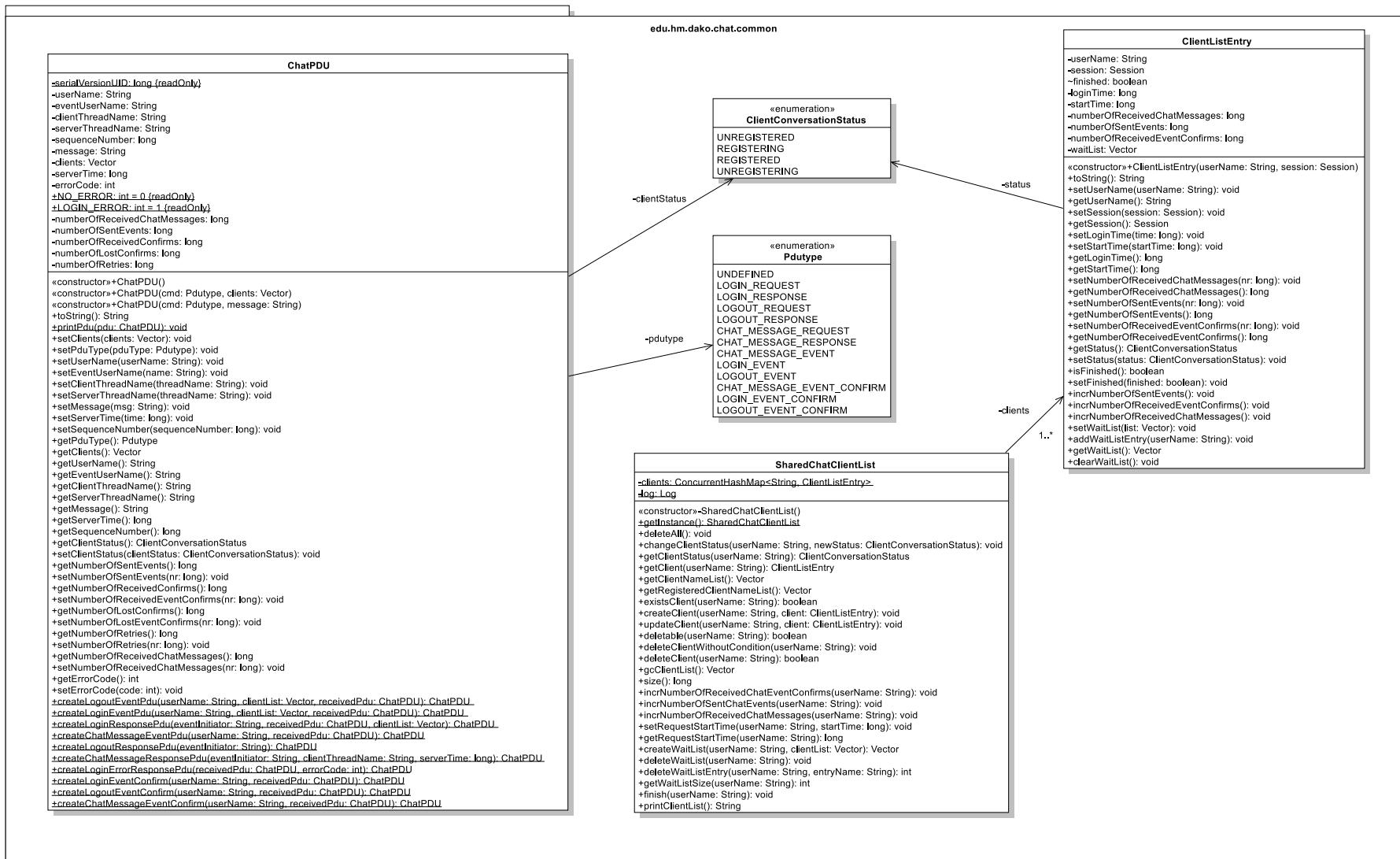
A. Anhang

| Anzahl Clients | 3 | 15 | 30 | 45 | 60 | 75 | 90 | 99 |
|---------------------|----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
| Min. RTT | 27 | 9 | 19 | 18 | 28 | 26 | 51 | 64 |
| Max. RTT | 154 | 1991 | 5618 | 10321 | 18260 | 25881 | 35720 | 43344 |
| Mittlere RTT | 58 | 700 | 2052 | 3838 | 6642 | 9698 | 13580 | 16264 |
| Mittlere Serverzeit | 32 | 657 | 1975 | 3745 | 6369 | 9406 | 13354 | 15970 |
| T _k | 26 | 43 | 77 | 93 | 273 | 192 | 226 | 294 |

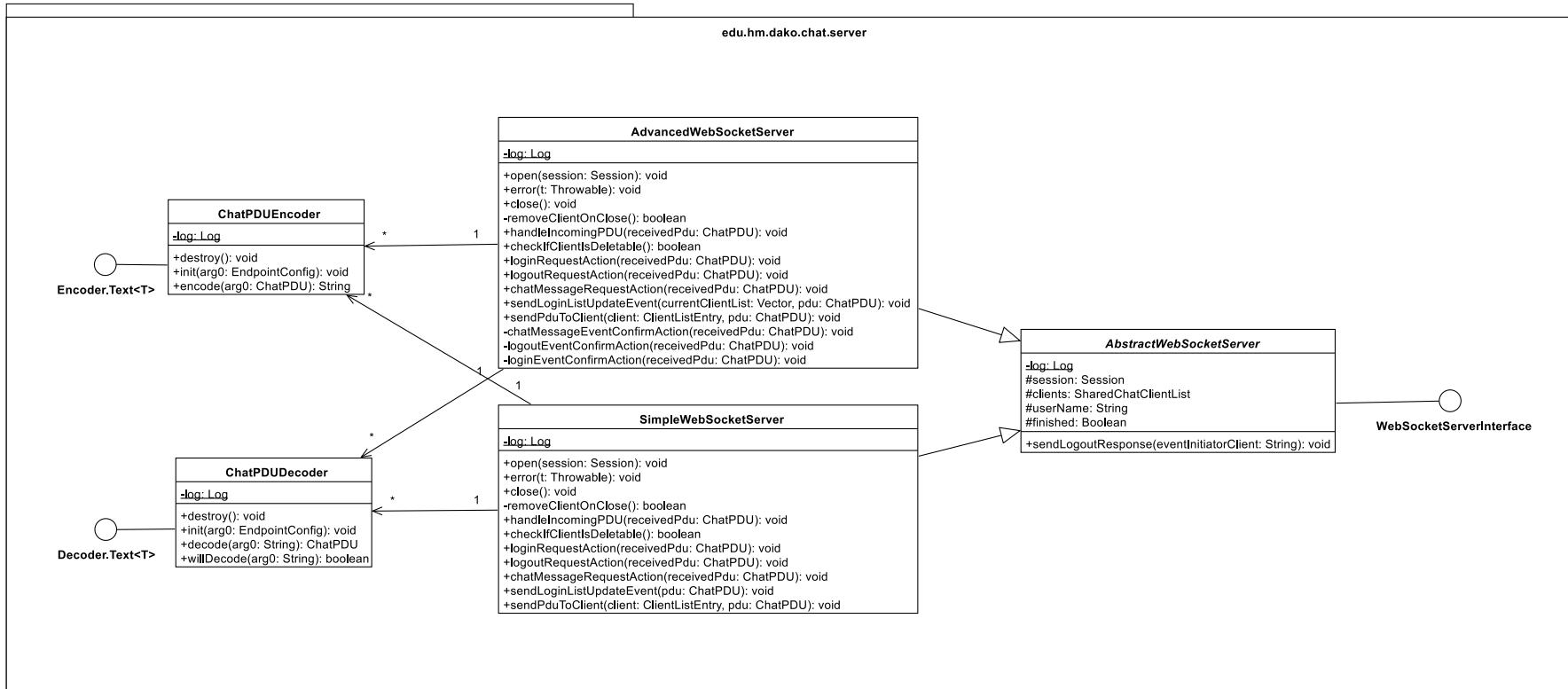
Tabelle A.1: Ergebnisse der WebSocket-Leistungsmessung

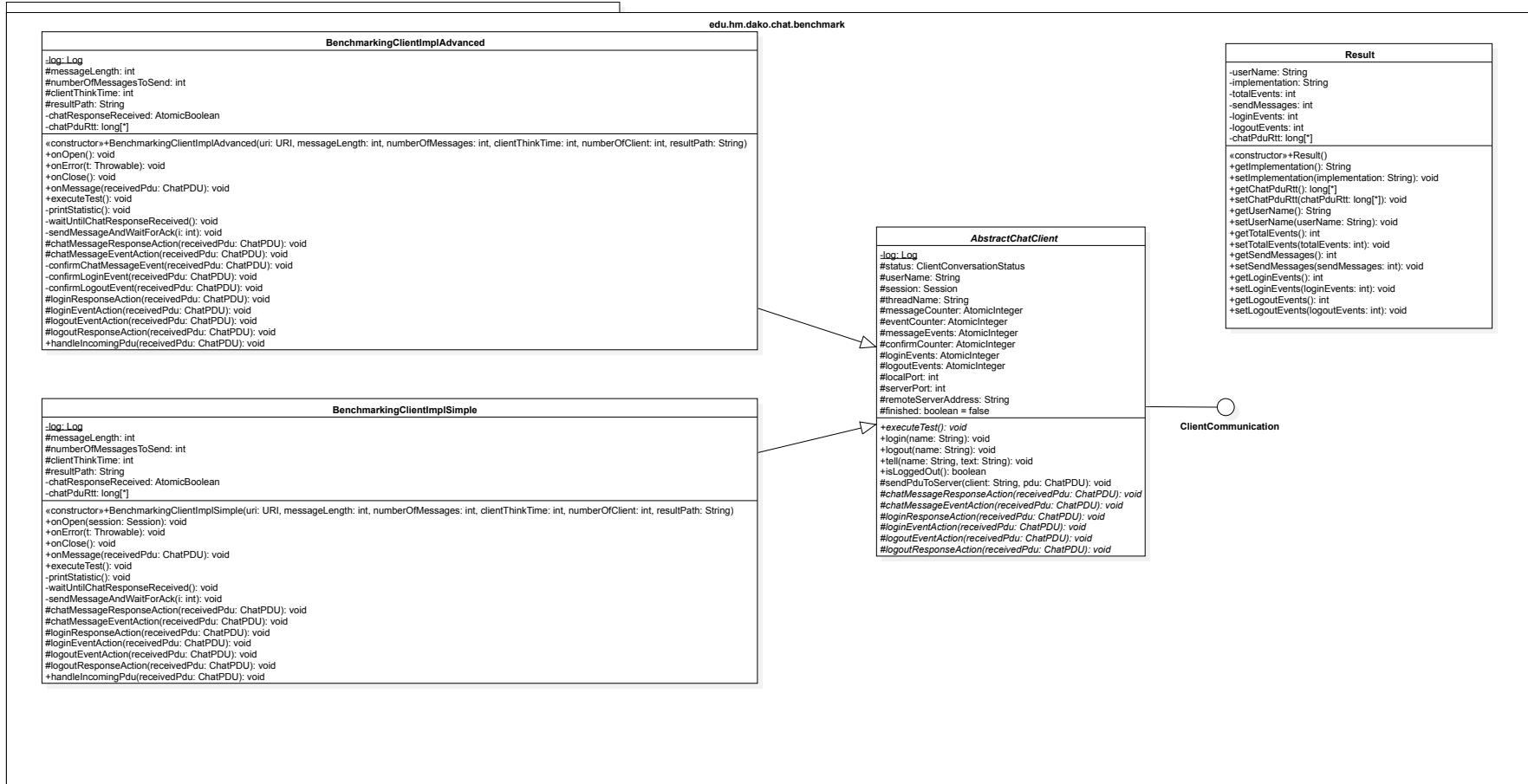
| Anzahl Clients | 3 | 15 | 30 | 45 | 60 | 75 | 90 | 99 |
|---------------------|----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
| Min. RTT | 4 | 7 | 13 | 21 | 20 | 28 | 26 | 28 |
| Max. RTT | 17 | 105 | 379 | 793 | 1409 | 2460 | 2414 | 2974 |
| Mittlere RTT | 8 | 32 | 158 | 364 | 557 | 978 | 1127 | 1280 |
| Mittlere Serverzeit | 4 | 26 | 118 | 274 | 441 | 734 | 933 | 1047 |
| T _k | 4 | 6 | 40 | 90 | 116 | 244 | 194 | 233 |

Tabelle A.2: Ergebnisse der TCP-Leistungsmessung

Abbildung A.2: Klassendiagramm: WebSocket-Server *common*-Package

X

Abbildung A.3: Klassendiagramm: WebSocket-Server *server*-Package

Abbildung A.4: Klassendiagramm: WebSocket-Benchmark *benchmark*-Package

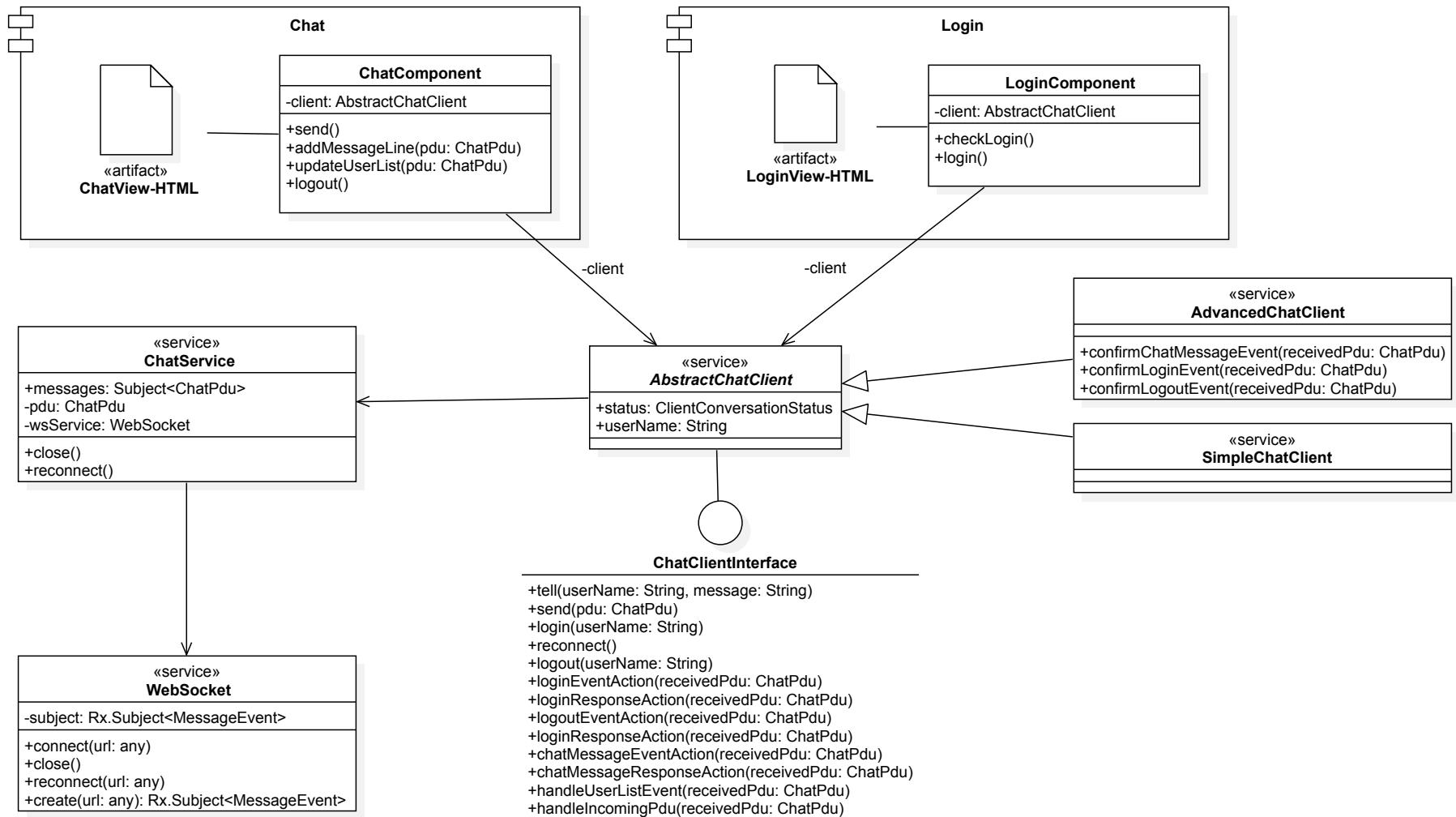


Abbildung A.5: Klassendiagramm: WebSocket-Client