



Bachelorarbeit im Studiengang B.Sc. Wirtschaftsinformatik

Realisierung einer bidirektionalen Chat-Anwendung durch HTML5 WebSockets mit Java und Angular 4 zum Einsatz im seminaristischen Kontext

Andre Weinkötz

15. Februar 2018

Fakultät für Informatik und Mathematik
Hochschule München

Betreuer: Prof. Dr. Mandl

Abstract

Hier steht am Ende was genau eigentlich mit dem Unsinn hier erreicht werden soll.

Inhaltsverzeichnis

| | |
|--|-----------|
| Abstract | 2 |
| 1 Einleitung | 4 |
| 2 Das WebSocket Protokoll | 5 |
| 2.1 WebSocket-Frames | 5 |
| 2.1.1 Header-Felder | 6 |
| 2.1.2 Non-Control-Frames | 7 |
| 2.1.3 Control-Frames | 9 |
| 2.2 WebSocket Lifecycle | 11 |
| 2.2.1 Verbindungsaufbau | 13 |
| 2.2.2 Datentransfer | 13 |
| 2.2.3 Verbindungsabbau | 13 |
| 2.3 Vor- und Nachteile gegenüber Request | 13 |
| 3 Die WebSocket API | 13 |
| 3.1 Client-seitige Implementierung | 13 |
| 3.2 Server-seitige Implementierung | 13 |
| 4 Fachliche Anforderungen | 13 |
| 4.1 Analyse des DaKo-Frameworks | 13 |
| 4.2 Anforderungen an WebSocket-Chat | 13 |
| 4.3 Prototypische Implementierung | 13 |
| 5 Technische Anforderungen | 13 |
| 5.1 Entwurf des WebSocket-Chats | 13 |
| 5.2 Vergleich der Implementierungsformen | 13 |
| 5.3 Implementierung des WebSocket-Chats | 13 |
| 6 Evaluation | 13 |
| 6.1 Test-Umgebung | 13 |
| 6.2 Leistungsanalyse WebSocket-Chat | 13 |
| 7 Zusammenfassung und Ausblick | 13 |
| 7.1 Einsatzgebiete | 13 |

1 Einleitung

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. Das Hypertext Transfer Protocol (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 mit zugehöriger JavaScript-API.

Die Entwickler des WebSocket Protokolls machten es sich zur Aufgabe, einen Mechanismus zu schaffen, der es browserbasierten Anwendungen ermöglicht bidirektional zu kommunizieren ohne dabei auf mehrere HTTP Verbindungen zu öffnen [FM11]. Dabei sollte auf zusätzliche Methoden - wie beispielsweise den Einsatz von XMLHttpRequests, iframes oder long polling - verzichtet werden.

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 WHATWG¹ der Internet Engineering Task Force (IETF) vorgeschlagen wurde. Bis Mai 2010 wurden noch 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, sowie 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 dem Lebenszyklus und den verschiedenen Frames der WebSockets noch zahlreiche weitere technische Details und Definitionen. In diesem Kapitel sollen die wichtigsten Grundlagen der Protokollspezifikation dargelegt 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, welcher die Nutzdaten beinhaltet. Anhand des Aufbaus ist leicht erkennbar, dass die Kommunikation über WebSockets mit

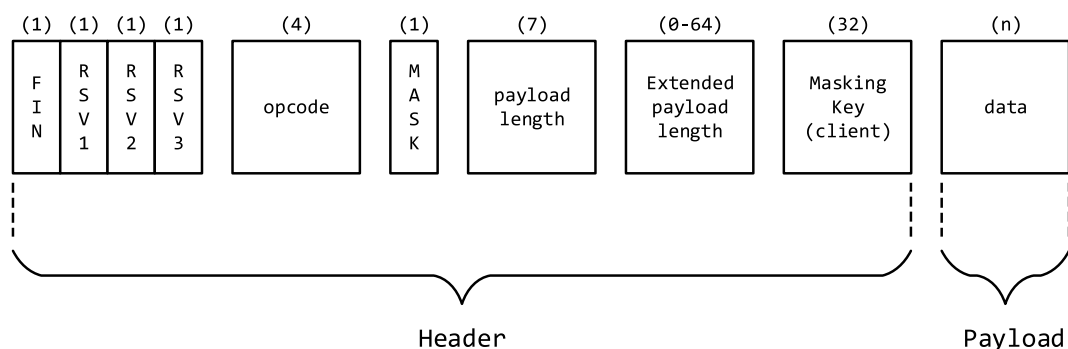


Abbildung 1: Vereinfachte Darstellung eines WebSocket Frames

deutlich 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 noch 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].

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

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.

RSV1, RSV2, RSV3 (jeweils 1 Bit) Diese Flags sind für zukünftige Implementierung reserviert und finden aktuell keine standardisierte Anwendung. Die Ausnahme stellt das RSV1-Bit dar, welches verwendet wird um anzuzeigen, 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, welches das jeweilige Flag interpretieren kann, so muss die WebSocket-Verbindung geschlossen werden [FM11, S. 27].

opcode (4 Bit) Das opcode-Feld enthält die Angabe wie der Payload zu interpretieren ist. Neben den sogenannten Datenframes bzw. Non-Control-Frames, welche 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 ein Fortsetzungs-Frame
- **0x1** definiert ein Text-Frame
- **0x2** definiert ein Binary-Frame
- **0x3-7** reserviert für weitere Non-Control-Frames
- **0x8** definiert ein Verbindung beenden-Frame
- **0x9** definiert ein Ping-Frame
- **0xA** definiert ein Pong-Frame
- **0xB-F** reserviert für weitere Control-Frames

MASK (1 Bit) Ist das MASK-Flag gesetzt, so ist der Payload maskiert. 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 Bytes, 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 Länginfeldern 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 der aufgrund von Sicherheitsproblemen eingeführten Maskierung clientseitig initiiert Nachrichten.

Fragmentierung

In Abschnitt 5.4. des RFC6455 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 werden hierbei außer Acht gelassen:

- Unfragmentierte Nachrichten werden mit einem einzelnen Frame übertragen (FIN-Bit = 1, opcode \neq 0)
- Die Übertragung fragmentierter Nachrichten erfolgt über mindestens einen Frame mit FIN-Bit = 0 und opcode \neq 0 und wird mit einem Frame FIN-Bit = 1 und opcode \neq 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 Fragemente einer Nachricht in der gleichen Reihenfolge erhalten, in welcher sie vom Sender aufgegeben wurden.
- 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, von einem Client zu einem Server gesendet, müssen maskiert 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 deren 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 verhindert, 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.

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

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

Tabelle 1: Maskierung durch bitweise XOR-Verknüpfung

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

Tabelle 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 RFC6455 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 RFC6455 unterscheidet dabei drei Arten von Control-Frames, welche 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 RFC6455 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].

| | 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: Ping- und Pong-Frames zur Verbindungskontrolle

Bei den in Abbildung 2 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.

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.

| | Client |
|---------------------------|---|
| 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 3: Close-Frame initiiert von Client

Abbildung 3 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*, welche besagt,

| | Server |
|----------------------|-------------------------------------|
| 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 4: Close-Frame Antwort des Servers

dass die Verbindung ordnungsgemäß beendet wurde. Der RFC6455 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 RFC6455 sowie für zukünftige Überarbeitungen und Erweiterungen reserviert.

3000-3999 Status-Codes in diesem Bereich sind für 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 einem gewissen Wildwuchs 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 3 zeigt einen kurzen Auszug der wichtigsten Status-Codes. Es fällt auf, dass alle Neuregistrierungen durch Alexey Melnikov erfolgten, welcher 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 denn überhaupt genutzt werden und ob noch ein Interesse an deren Registrierung besteht [Mel12].

2.2 WebSocket Lifecycle

Lebenszyklus woop woop

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

| Status-Code | Bedeutung | Beschreibung | Spezifiziert durch |
|-------------|------------------|--|--------------------|
| 1000 | Normal Closure | Zeigt einen ordnungsgemäßen Verbindungsabbau an. | RFC6455 |
| 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. | RFC6455 |
| 1006 | Abnormal Closure | Dieser Status darf nicht in einem Close-Frame verwendet werden. Anwendungen können durch diesen signalisieren, dass die Verbindung unerwartet und auf unbekannte Weise beendet wurde. | RFC6455 |
| 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. | RFC6455 |
| 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 3: Auszug der registrierten Status-Codes

2.2.1 Verbindungsaufbau

2.2.2 Datentransfer

2.2.3 Verbindungsabbau

2.3 Vor- und Nachteile gegenüber Request

3 Die WebSocket API

3.1 Client-seitige Implementierung

3.2 Server-seitige Implementierung

4 Fachliche Anforderungen

4.1 Analyse des DaKo-Frameworks

4.2 Anforderungen an WebSocket-Chat

4.3 Prototypische Implementierung

5 Technische Anforderungen

5.1 Entwurf des WebSocket-Chats

5.2 Vergleich der Implementierungsformen

5.3 Implementierung des WebSocket-Chats

6 Evaluation

6.1 Test-Umgebung

6.2 Leistungsanalyse WebSocket-Chat

7 Zusammenfassung und Ausblick

7.1 Einsatzgebiete

Literatur

- [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: 978-3-446-44371-6 978-3-446-44438-6.
- [Hic10] Ian Hickson. *The WebSocket Protocol*. 6. Mai 2010. URL: <https://tools.ietf.org/search/draft-hixie-thewebsocketprotocol-76> (besucht am 19.12.2017).
- [Hua+11] Lin-Shung Huang u. a. „Talking to Yourself for Fun and Profit“. In: *Proceedings of W2SP* (1. Jan. 2011), S. 1–11.
- [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).
- [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).
- [Mel12] Alexey Melnikov. *Additional WebSocket Close Error Codes*. 16.5.12. URL: <http://www.ietf.org/mail-archive/web/hybi/current/msg09670.html> (besucht am 25.12.2017).
- [She17] Eric Shepherd. *Writing WebSocket Servers*. 16.8.17. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API/Writing_WebSocket_servers (besucht am 22.12.2017).

Abbildungsverzeichnis

| | | |
|---|---|----|
| 1 | Vereinfachte Darstellung eines WebSocket Frames | 5 |
| 2 | Ping- und Pong-Frames zur Verbindungskontrolle | 10 |
| 3 | Close-Frame initiiert von Client | 10 |
| 4 | Close-Frame Antwort des Servers | 11 |

Tabellenverzeichnis

| | | |
|---|---|----|
| 1 | Maskierung durch bitweise XOR-Verknüpfung | 9 |
| 2 | Demaskierung durch bitweise XOR-Verknüpfung | 9 |
| 3 | Auszug der registrierten Status-Codes | 12 |