Potenziale von Server Push-Mechanismen im Kontext von Informationssystemen

Herbert Neuendorf

Fakultät Wirtschaft – Studiengang Wirtschaftsinformatik
Duale Hochschule Baden-Württemberg Mosbach
Lohrtalweg 10
74821 Mosbach
neuendorf@dhbw-mosbach.de

Abstract: Ein Aspekt der Gestaltung von Informationssystemen ist die effektive Präsentation und Aktualisierung der Informationsinhalte in Echtzeit. In diesem Beitrag werden innovative Möglichkeiten von Server Push-Technologien beleuchtet, die im Rahmen standardisierter Protokolle und plattformunabhängiger Oberflächentechnologien zur Verfügung stehen. Der Beitrag versteht sich als Hinweis auf neuartige Kommunikations-Muster im Rahmen serverbasierter Anwendungen und umreißt ein offenes Forschungsfeld als work in progress.

1 Motivation und Überblick

Neuartige Technologien und offene Standards im Bereich Server Push unterstützen serverbasierte Anwendungen. Ohne vertraute serverbasierte Programmiermodelle aufgeben zu müssen, können asynchrone, event-getriebene Echtzeit-Benachrichtigungen und –Aktualisierungen der involvierten Clients dargestellt werden. Dabei handelt es sich primär um die Standards Server Sent Events (SSE) und WebSockets (WS), durch die eine polling-freie Kommunikation vom Server zum Client und dadurch eine Echtzeit-Aktualisierung des Clients (plugin-frei) erreicht wird.

Die genannten Standards stellen eine wichtige Komponente zur Realisation von Informationssystemen dar, und ermöglichen die Realisation latenzfreier Kommunikations- und Geschäftsprozesse, ohne auf proprietäre Oberflächentechnologien und Deployment-Verfahren zurückgreifen zu müssen. Dabei ist eine zunehmende Unterstützung der Standards durch gängige Browser zu beobachten.

Indem das reine Request-Response-Modell von HTTP erweitert wird um server-getriebene Benachrichtigungen, lassen sich auch im Rahmen browserbasierter Frontends und serverbasierter Inside-Out-Programmiermodelle Echtzeit-Anwendungen darstellen. Technologische Innovationen im Bereich browserbasierter Clients unterstützen somit auch serverbasierte Informationssysteme.

1.1 Server-Sent Events und Websockets im Überblick

Die Standards SSE und WS sind Teil der HTML5-Spezifikation [BER14] und betreffen die Kommunikation zwischen Server und Client (Browser), wobei die beiden Protokolle auf unterschiedlicher Protokollebene agieren (s.u.). In beiden Fällen handelt es sich um die Möglichkeit, vom Server auf Basis einer persistenten Verbindung wiederholt Daten an den Client zu senden (Server Push), ohne erneute Einzel-Requests der Clients voraus-zusetzen (wie XHR-Polling und -Long Polling). Dabei beruht SSE auf Half-Duplex-Kommunikation, während WS Full-Duplex-Kommunikation unterstützt. Durch den Aufbau einer stehenden Verbindung zwischen Client und Server entfällt die Belastung des Netzwerks durch ständigen aufwendigen Neuaufbau von TCP-Verbindungen. Dadurch reduziert sich auch der Stromverbrauch mobiler Geräte [GRI13].

Innerhalb einer Client-Anwendung können - getrieben durch vom Server ausgelöste Events und gesendete Daten - jederzeit Aktualisierungen verzugslos vorgenommen werden. Speziell SSE lassen sich serverseitig direkt implementieren, ohne dass der Server das Protokoll nativ unterstützen müsste. Somit lässt sich auch mit nicht WS-fähigen Servern ein Push-Modell zur Client-Benachrichtigung implementieren.

Einem serverbasierten Informationssystem ist es somit möglich, einen stets aktuellen Systemzustand in Quasi-Echtzeit auf dem Client abzubilden.

1.2 Anwendungsfelder Server Push

Durch Server Push-Technologien erschließen sich Anwendungsfelder innerhalb lose gekoppelter Systeme, die bislang proprietären Technologien vorbehalten waren. Im folgenden werden einige repräsentative anwendungsbezogene und anwendungs-übergreifende Szenarien genannt.

a) Anwendungsbezogene Szenarien:

- Im betrieblichen Umfeld können dispositive betriebswirtschaftliche Daten zum Monitoring innerhalb von Management-Dashboards ständig aktualisiert werden - relevant z.B. im Bereich des Transportwesens und der mobilen Logistik sowie bei mobilen Dienstleistungen (z.B. im Bereich Facilty Mangement).

- Im Bereich Business Intelligence (BI) erfordern Berechnungen auf großen Datenmengen auch weiterhin eine batch-orientierte Arbeitsweise. Bei Vorliegen berechneter Kennzahlen können browserbasierte BI-Cockpits aktualisiert werden.

- Clients können jederzeit auf Events reagieren, die im Rahmen von ereignis-orientierten Geschäftsprozessen erzeugt werden. Der Status ablaufender Prozesse kann Prozessbeteiligten gemeldet werden. Dies kann zur Unterstützung von Kollaborationsplattformen und Echtzeit-Handelsplattformen eingesetzt werden, sowie mobiles Arbeiten unterstützen.

- Innerhalb von Konferenzsystemen, SocialMedia- und Gaming-Anwendungen ist Instant Messaging realisierbar.

- Im Bereich Mobile Commerce ließen sich Location Based Services (für ortsbezogene aktuelle Informationen) innerhalb von Märkten realisieren, ohne dass Kunden zur Installation von Anwendungen genötigt wären.

b) Anwendungsübergreifende Szenarien:

- Ein Server kann proaktiv und verzugslos Statusmeldungen an Clients streamen, bzw. Probleme oder Schwellwertüberschreitungen melden.

- Messaging-Architekturen können dargestellt werden, da hier größere Datenmengen vom Message Broker (Server) zu einem oder mehreren Clients fließen und Forderungen nach Echtzeitdatenversorgung geringer Latenz bestehen. Messaging-Architekturen können eingesetzt werden, um verschiedene Teile von Informationssystemen lose zu koppeln und zu integrieren. Grundsätzlich lassen sich auch Peer-to-Peer-Ansätze unterstützen, bei denen der Server als Relay-Instanz fungiert. So sind über einen intermediären Websocket-Server 1:n Broadcast-Mechanismen zu zahlreichen Clients realisierbar.

- Innerhalb von Browser-Frontends profitiert die Kombination mehrerer Applikationen zu Meta-Applikationen (Mashups) speziell durch die Möglichkeiten der WS-Technologie.

- Auf Basis von Websockets sind Virtual Network Computing und Remote Desktop Applikationen darstellbar.

- Aktuelle Anwendungspotenziale ergeben sich im Bereich der Machine-to-Machine-Kommunikation. Die Echtzeit-Nachrichtenübermittlung von zu trackenden Objekten (Smart Meter, Fahrzeuge, Schließanlagen, etc.) könnte unterstützt werden, ohne zu propriätären Protokollen greifen zu müssen.

Alle genannten Szenarien setzen eine verzugslose oder sogar kontinuierliche Statusaktualisierung und Synchronisation des Clients voraus. Trotz der Zustands- und Session-Freiheit von HTTP können stateless arbeitende Server mittels Server Push den Benutzerkontext initiativ sofort aktualisieren, ohne diesen selbst längerfristig bis zur nächsten Client-Anfrage vorhalten zu müssen. Insofern stellt Server Push eine grundsätzliche technische und semantische Ergänzung des zustandslosen HTTP-Protokolls dar. Eine zusätzliche Last auf dem Server entsteht nur durch das Offenhalten der Verbindung zum Client. Jedoch wird das Latenzproblem gelöst, das entsteht, wenn der Client Statusaktualisierungen stets aktiv anfordern muss.

In allen Fällen lassen sich browserbasierte und auch mobile Lösungen latenzfrei realisieren, ohne auf spezielle Programmiermodelle angewiesen zu sein, die einem serverbasierten InSide-Out-System einen Outside-In-Programmieransatz aufzwingen würden. Der Vorteil browserbasierter Frontends im betrieblichen und mobilen Umfeld beruht auf deren installationsfreier Aktualität. Die Notwendigkeit eines

Endgerätemanagements (BYOD-Problematik) wird (bis auf die Installation eines einheitlichen Browsertyps) deutlich entschärft. Durch den Verzicht auf proprietäre Plug-In-Technologien erhöht sich die Zukunfts- und Investitionssicherheit von Anwendungen.

2 Server-Sent Events

Server-Sent Events (SSE) arbeiten auf Basis von HTTP [HIC12b]. Sie werden bereits von gängigen Browsern (Firefox, Chrome, Safari, Opera – nicht jedoch Internet Explorer) unterstützt. SSE beruhen auf einer Half-Duplex-Technologie, d.h. es ist nur vorgesehen, textuelle Daten vom Server zum Client (Browser) zu senden. Voraussetzung ist ein vorausgegangener Verbindungsaufbau des Clients zum Server. Im Anschluss kann dieser Server jederzeit Benachrichtigungen (push notifications) und Informationsaktualisierungen in Echtzeit über eine langlebige offene HTTP-Verbindung zum Client senden, ohne dass dazu periodisches clientseitiges Polling erforderlich wäre.

Dies stellt gegenüber der XHR-Technologie (Ajax) [KES14b] auf Basis von periodischem Client-Polling bzw. Long-Polling („hanging GET“ oder „Comet“) (mit entsprechender Bandbreitenbelastung bzw. verzögerter Benachrichtigung) einen deutlichen Effizienzgewinn dar. Mittels SSE sind im Gegensatz zu XHR auch effiziente Streaming-Lösungen darstellbar, während die geplante Erweiterung des XHR-Protokolls durch eine Stream API [MOU13] momentan noch nicht verfügbar ist [GRI13].

SSE nutzen HTTP und setzen (im Gegensatz zu Websockets) nicht die serverseitige Unterstützung eines weiteren Protokolls voraus. Vielmehr kann die HTTP-Response des Servers in einem konventionellen Servlet ohne direkte serverseitige Protokoll-Unterstützung implementiert werden. Auf Seiten des Clients übernimmt der Browser das Verbindungsmanagement und die Details der Datenextraktion, so dass die Applikation sich auf die Geschäftslogik fokussieren kann.

SSE unterliegen (wie XHR) grundsätzlich der Same-Origin-Policy (SOP) [BAR10], so dass Cross-Origin Requests (ursprungsübergreifende Kommunikation) nicht zugelassen sind. Somit können Benachrichtigungen nur von dem Server an den Client gesendet werden, an den der eröffnende Request des Clients adressiert war. Inzwischen wurde die Spezifikation jedoch zugunsten größerer Flexibilität unter Berücksichtigung möglichst hoher Sicherheit erweitert, so dass Cross-Origin Resource Sharing (CORS) [KES14a] erlaubt wird, um Webclients Cross-Origin Requests gegen fremde Server zu ermöglichen. Allerdings wird diese SSE-Erweiterung erst von wenigen Browsern unterstützt. Sicherheit wird also je nach Browsertyp durch SOP oder die CORS-API gewährleistet.

2.1 Verbindungsaufbau und Datentransfer

Clientseitig beruht die SSE-API ausschließlich auf dem `EventSource`-Objekt. Zum Verbindungsaufbau zum Server genügt es, ein `EventSource`-Objekt mittels URL zu erstellen. In beschränktem Maße kann der Client dabei Daten zum Server senden, indem

der URL ein Query-String angefügt wird. Daraufhin baut der Browser eine Verbindung zum Server auf und sendet einen HTTP-Get-Request.

Im Client sind Handler-Funktionen für zu behandelnde Events zu implementieren. Diese werden asynchron aufgerufen und aktualisierte Daten übergeben, wenn vom Server eine Benachrichtigung gesendet und ein entsprechendes Event im Client ausgelöst wird – ohne jegliches (potenziell verzögertes) Polling durch den Client. Im Client stehen die Events `open`, `close`, `error` und `message` zur Verfügung.

Auf Seiten des Servers sind bezüglich des EventStream-Datenformats nur wenige Anforderungen zu beachten: Der Content-Type der Response muss explizit text/event-stream lauten und die textuelle Response in UTF-8 kodiert sein. Die eigentlichen Nutzdaten sind Werte eines oder mehrerer aufeinander folgender `data:`-Felder. Auch das Versenden im JSON-Format ist möglich. Für syntaktische Details verweisen wir auf die Spezifikation [HIC12b].

Nachrichten können mit einem frei wählbaren Eventnamen versehen werden. Serverseitig wird dazu der Benachrichtigung eine Zeile mit `event:<name>` hinzugefügt. Auf diese Weise können innerhalb eines einzigen Server Push mehrere verschiedene benamte Nachrichten gleichzeitig an den Client gesendet und von diesem separat behandelt werden. Benamte Nachrichten lösen entsprechend benamte Events aus, die durch spezielle Event Handler des Clients separat behandelt werden können. Auf diese Weise lassen sich Verhaltensweisen des Clients feingranular durch serverseitige Benachrichtigungen auslösen – z.B. für unterschiedliche Statusmeldungen des Servers.

Zur Realisation von Streaming-Lösungen muss die Verbindung permanent offen gehalten werden. Deshalb sendet das `EventSource`-Objekt des Clients automatisch einen neuerlichen Get-Request zum Server, sobald die Verbindung erfolgreich beendet oder durch einen Verbindungsfehler unterbrochen wurde. Konfigurierbar ist das zeitliche Client-Verhalten durch die `retry`-Option der Server-Response. Die client-seitige API abstrahiert von den Details des Aufbaus der Verbindung, deren automatischer Aufrechterhaltung und Wiederherstellung, was die Implementierung clientseitiger Anwendungslogik erleichtert.

Der Neuaufbau einer geschlossenen Verbindung durch den Client kann im Client durch Aufruf der Methode `close()` des `EventSource`-Objekts beendet werden. Dieser Aufruf schließt sofort die aktuelle Verbindung zum Server. Auch servergesteuert lässt sich die `EventSource` stoppen, indem eine HTTP-Response ohne Content-Type Header oder mit Status-Code ungleich 200 gesendet wird. Letzteres ist bei Streaming-Szenarien nicht möglich, da Header-Informationen bereits zu Streaming-Beginn zum Client gesendet wurden. In diesem Fall muss der Server ein speziell benamtes Event senden, auf das der Client mit Schließen des `EventSource`-Objekts reagiert.

Mittels des Nachrichtenelements `id` kann der Server jeder Nachricht einen beliebigen Identifikations-String zuweisen, der im Attribut `lastEventID` des `event`-Objekts abgelegt wird. Auf diese Weise kann der Client überprüfen, ob er alle Nachrichten erhalten hat. Wird die Verbindung zum Server beendet oder unterbrochen, so sendet der

Client beim nächsten Request den zusätzlichen HTTP-Header `Last-Event-ID` zum Server, der den letzten vom Client erhaltenen `id`-Wert enthält. Der Server kann den Header abfragen und entsprechend reagieren - z.B. alle Nachrichten seit dieser ID zum Client senden oder den Verlust ignorieren.

Zur Erhöhung der Sicherheit steht auf dem `event`-Objekt das Attribut `origin` zur Verfügung. Somit kann der Ursprung jeder Servernachricht im Client überprüft werden.

2.2 Bewertung

SSE sind das Mittel der Wahl, wenn textuelle Echtzeitdaten vom Server geliefert werden sollen, ohne dass der User einen Rückmeldungskanal zum Server benötigt. Markante Eigenschaften des SSE-Protokolls sind seine einfache und flexible Handhabbarkeit. Es wird HTTP als Trägerprotokoll genutzt, das von CORS unterstützt werden kann. Da der Rahmen des HTTP-Protokolls nicht verlassen wird, treten bei Proxy-Servern keine Akzeptanzprobleme beim Verbindungsaufbau auf.

Durch Verwendung von HTTP können SSE serverseitig im Rahmen jedes Webframeworks dargestellt werden, dass eine Manipulation der HTTP-Response erlaubt. Eine native Unterstützung durch spezielle Server oder Webframeworks ist nicht erforderlich. Browserseitig sind SSE relativ einfach durch Javascript-Bibliotheken (Polyfills) abbildbar (wie im Falle IE9), falls sie nicht nativ unterstützt werden.

Somit sollten SSE zur Realisation textueller Server Push-Szenarien geringer Latenz in Betracht gezogen werden, ohne im Vornherein auf die Websocket-Technologie zu fokussieren.

3 Websockets

Das Websocket-Protokoll wird spezifiziert durch die IEFT (RFC6455) [FET11], die Websocket-API durch das W3C [HIC12a]. Websockets ermöglichen eine bidirektionale Full-Duplex-Kommunikation textueller oder binärer Daten zwischen einem beliebigen Client (u.a. Browser) und einem Server auf Basis des WS-Protokolls. Somit werden (im Gegensatz zu SSE) WS-fähige Server zur Realisation benötigt. Eine beispielhafte Java-basierte Server-API ist spezifiziert in [COW13].

Das WS-Protokoll wurde speziell zur bidirektionalen Kommunikation entworfen, statt (wie bei SSE) eine zusätzliche Protokollschicht oberhalb von HTTP zu definieren. Dadurch entfällt zusätzlicher HTTP-Overhead nach Aufbau der Verbindung.

3.1 Verbindungsaufbau

Das WS-Protokoll [FET11] verwendet einen initialen HTTP-Request über HTTP-Standard-Ports und den HTTP-Upgrade-Mechanismus, um beim Opening-Handshake eine HTTP-Verbindung zu einer WS-Verbindung aufzuwerten. Das WS-Protokoll wird

mittlerweile von allen gängigen Browsern unterstützt. Neben dem WS-Protokoll existiert auch das WSS-Schema, das zur Verschlüsselung auf TLS (SSL) aufsetzt.

Bislang unterstützen WS noch kein Verbindungs-Multiplexing - d.h. jede WS-Verbindung basiert auf einer dedizierten TCP-Verbindung. Dies kann problematisch sein angesichts der begrenzten Zahl offener Browser-Verbindungen. Aktuell ist eine Multiplex-Erweiterung für WS in Entwicklung [TAM12], durch die eine physische TCP-Verbindung als Träger mehrerer separater logischer WS-Verbindungen fungieren kann, indem jede einzelne WS-Verbindung durch eine eigene Kanal-ID ausgezeichnet wird. Mehrere WS-Verbindungen (channels) können somit über dieselbe geteilte TCP-Verbindung dargestellt werden.

Grundsätzlich geht die Verbindungsaufnahme stets vom Client aus. Webclients können keine WS-Verbindungen akzeptieren, die sie nicht selbst initiiert haben. Somit sind Clients vor ungewollten Verbindungsaufnahmen geschützt.

Eine wichtige Neuerung ist die Cross-Domain-Fähigkeit von WS, die nicht den Beschränkungen der Same-Origin-Policy (wie SSE) unterworfen sind, so dass eine Kommunikation zwischen verschiednen Domänen möglich ist. Innerhalb einer WS-Applikation können somit z.B. separate Server für die Kern-Applikation einerseits und Streaming-Aufgaben andererseits eingesetzt werden (client side aggregation). Somit entfällt die Notwendigkeit, die gesamte Applikation auf einem zentralen Server zu hosten (server side aggregation) und der damit verbundene Engpass. Eine Verbindung kann zu jedem beliebigen Server aufgebaut werden, sofern das Handshake-Protokoll korrekt ausgeführt wird.

Dem Schutz des Servers vor DenialOfService-Attacken dient das Header-Feld `Origin` innerhalb der WS-Client-Anfrage. Durch dieses werden Schema, Host und Port des anfragenden Clients spezifiziert. Der `Origin`-Header ermöglicht es dem Server, WS-Verbindungen zu akzeptieren oder abzulehnen. Der Browser ist verpflichtet den `Origin`-Header beim Handshake mitzusenden. Der Server kann anhand des Wertes entscheiden, ob er den Verbindungsaufbau bestätigt oder durch eine Fehlermeldung ablehnt. Serverseitig kann vorgegeben werden, ob die Serverdienste Clients jeglicher Domäne oder nur definierter Domänen zugänglich sind.

3.2 Unterstützung von Subprotokollen

Das clientseitige `Websocket`-Objekts wird durch Angabe der Server-URL im Client erzeugt [HIC12a]. Als optionaler Konstruktor-Parameter können Namen von Subprotokollen (als String-Array) an den Server übermittelt werden, die auf Basis der WS-Kommunikation verwendet werden sollen. Das WS-Protokoll kennzeichnet die übermittelten Daten als textuell oder binär, macht selbst aber keine weiteren Vorgaben über deren Format und das ihrer Nutzung zugrunde liegende Protokoll. Der Server wählt ein von ihm unterstütztes Protokoll aus, so dass Client und Server mittels des ausgehandelten Zusatzprotokolls interagieren können. Die Mitteilung der erfolgreichen Protokollauswahl durch den Server und der somit erreichten Protokollvereinbarung

zwischen Client und Server geschieht über den WS-Header `Sec-WebSocket-Protocol` der Server-Antwort und das gefüllte Attribut `protocol` des `WebSocket`-Objekts. Kann der Server das vom Client vorgeschlagene Protokoll nicht unterstützen, so schließt er die Verbindung und das `error`-Event wird im Client ausgelöst. Bei den Protokollen kann es sich um offiziell registrierte WS-Subprotokolle handeln, weit verbreitete standardisierte Protokolle oder um selbst definierte Protokolle, z.B. auch Kompressionsverfahren bei textuellem Datenverkehr [GRI13].

Vorteil der WS-Technologie ist somit, dass Protokollerweiterungen und zusätzliche Protokolle spezieller Semantik und Funktionalität auf der WS-Protokollschicht aufbauen können. WS kann als Transportschicht für höhere Protokolle dienen. WS baut auf TCP/TLS auf, wird aber selbst auch als Transportprotokoll betrachtet, da Protokolle der Applikationsschicht auf WS fußen können. Typische Beispiele sind das eXtensible Messaging and Presence Protocol (XMPP) und das Simple Text Oriented Messaging Protocol (STOMP) [WAN13]. Im Rahmen einer WS-Anwendung wird HTTP jedoch weiterhin genutzt werden, um statische Ressourcen (HTML, CSS, JS) zu laden. Insgesamt ergibt sich somit der Protokollstapel gemäß Abbildung 1.

<table>
<tr><td colspan="3">Applikation</td></tr>
<tr><td>SSE</td><td>XMPP</td><td>STOMP</td></tr>
<tr><td>HTTP</td><td colspan="2">WS</td></tr>
<tr><td colspan="3">TLS</td></tr>
<tr><td colspan="3">TCP</td></tr>
</table>

Abbildung1: Protokollstapel im Kontext der Protokolle SSE und WS.

3.3 Kommunikation und Datenaustausch

Das Kommunikationsmodell folgt clientseitig dem Eventhandler-Muster; es können Behandler (asynchrone Callback-Funktionen) für die Events `open`, `message` und `close` definiert werden. Die WS-API arbeitet rein eventbasiert, jegliches Polling entfällt. Events und Benachrichtigungen werden asynchron empfangen und verarbeitet.

Meldet der Server einen Fehler, so wird auf dem Client das `error`-Event ausgelöst und die Verbindung geschlossen. Ein automatischer Neuaufbau der Verbindung durch den Client (wie bei SSE) findet nicht statt. Der Client kann durch Aufruf der `close()`-Methode des `Websocket`-Objekts die Verbindung zum Server schließen.

Sobald die Verbindung zum Server etabliert wurde (sichtbar durch Auslösen des `open`-Events im Client), kann der Client Daten an den Server mittels der Methode `send()` des `Connection`-Objekts senden. Ebenso kann der Server jederzeit Daten an den Client senden, wodurch in diesem das `message`-Event ausgelöst wird und die Daten im

`data`-Attribut des `event`-Objekts zur Verfügung stehen. In beiden Fällen kann es sich um textuelle Daten oder um Binärdaten handeln. Es besteht für den Server die Möglichkeit, alle verbundenen Clients zu adressieren und Aktualisierungen an diese gemeinsam zu versenden [COW13].

Eine aktive WS-Verbindung kann vor serverseitiger Überlastung geschützt werden, indem der Client die Datenübertragungsrate anpasst (throttling). Das `Websocket`-Attribut `bufferedAmount` legt die Zahl der Bytes fest, die auf dem Client vor der Versendung zum Server zwischengepuffert werden.

3.4 Proxy-Problematik

WS-Verbindungen können längere Zeit geöffnet bleiben. Dies kann zum Erreichen von Timeouts in zwischengeschalteten intermediären (forward oder reverse) Proxy-Servern oder Routern führen, durch die die (scheinbar untätige TCP- und somit auch WS-) Verbindung geschlossen wird. Um dies zu vermeiden, unterstützt das WS-Protokoll browser- und serverseitig initiierte `ping`- und `pong`-Aufrufe in frei wählbaren Intervallen, um die Verbindung offen zu halten.

Ebenso wie HTTP kann auch WS den Schutz durch Tunneling via TLS nutzen; dies entspricht dem Schema WSS (Websocket Secure) und kommt durch einen HTTPS-Handshake-Upgrade von HTTPS zu WSS zustande. Neben reinen Sicherheitsaspekten gestaltet sich dadurch die Durchleitung des WS-Verkehrs durch intermediäre Proxies unproblematisch, da verschlüsselter Datenverkehr durch Proxies nicht inspiziert und somit durchgeleitet wird. Ferner sehen Proxies für Tunnel typischerweise großzügigere Timeout-Einstellungen vor.

3.5 Bewertung

Durch WS wird ein neues symmetrisches Kommunikationspattern für serverbasierte Anwendungen eröffnet und das HTTP-Request-Response-Dogma durchbrochen. Dazu müssen WS-Server jedoch zahlreiche gleichzeitig geöffnete WS-Verbindungen bewältigen. Dies setzt eine performante und hochgradig nebenläufige Serverarchitektur voraus, typischerweise auf Basis von Thread-Pooling und nichtblockierenden Prozessen.

Insgesamt ermöglichen WS erstmals eine bidirektionale textuelle und binäre asynchrone Echtzeit-Kommunikation zwischen Client und Server mit minimaler Benachrichtigungsverzögerung.

Besondere Stärke des Websocket-Protokolls ist seine grundsätzliche Fähigkeit, als Träger beliebiger bidirektionaler Subprotokolle zwischen Client und Server zu fungieren, die innerhalb des Eröffnungs-Handshakes zwischen Client und Server vereinbart werden können.

4 Praktische Umsetzungen

Praktische Erfahrungen mit den Standards SSE und WS konnte im Rahmen von studentischen Projekten des Studiengangs Wirtschaftsinformatik der DHBW Mosbach gesammelt werden. Hierbei wurde eine hochschulbezogene Standort-WebApp realisiert, die aktuelle Informationen zu Themen des studentischen Alltags browserbasiert darstellt. Im Rahmen der Anwendung wurde u.a. bereits eine Chat-Möglichkeit auf Basis von WS realisiert sowie ein News-Ticker auf Basis von SSE. Dabei wurden SSE auf aktuellen Versionen der Browser Chrome, Firefox, Opera und Chrome für Android betrieben, WS auch auf Internet Explorer. Als Server kamen Tomcat und Jetty zum Einsatz

Momentan wird eine Firmenbefragung vorbereitet, um potenzielle betriebliche Nutzungsmöglichkeiten der Standards SSE und WS zu evaluieren und im Rahmen kooperativer Forschungsprojekte zu pilotieren.

5 Fazit

In dem Maße, in dem sich das Internet wandelt von einer dokumentenorientierten zu einer ereignisorientierten Nutzungs- und Wahrnehmungsweise (augmented web), werden auch standardisierte Server Push-Technologien an Bedeutung gewinnen. Speziell durch SSE und WS erschließen sich serverbasierten Informationssystemen weitere Kommunikationskanäle und Kommunikationsmuster. In Tabelle 1 werden charakteristische Protokolleigenschaften vergleichend gegenüber gestellt.

Eigenschaft	**TCP**	**UDP**	**HTTP**	**XHR**	**SSE**	**WS**
Adressierung	IP:Port	IP:Port	URL	URL	URL	URL
Übertragung	Full Duplex	Datagramms	Half Duplex	Half Duplex	Half Duplex	Full Duplex
Inhalt	Bytestream	Bytestream	Mimes	Mimes	Text	Text + Binär
Basisprotokoll	IP	IP	TCP	HTTP	HTTP	TCP
Verbindungs-orientiert	Ja	Nein	Nein	Nein	Ja	Ja
Verlässlichkeit	Ja	Nein	Ja	Ja	Ja	Ja
Cross Domain Messaging	--	--	(Nein) (CORS)	(Nein) (CORS)	(Nein) (CORS)	Ja

Tabelle 1: Eigenschaften verschiedener Web-Protokolle im Umfeld von Server Push.

Server Push-Technologien unterstützen und stärken vertraute serverbasierte Architekturen und Inside-Out-Programmiermodelle, durch die Geschäfts- und Prozesslogik auf dem Server zentralisiert, stabil und einheitlich bereitgestellt werden.

Hinz und Kessel [HIN13] weisen darauf hin, dass webbasierte Frontendsysteme zunehmend auf dedizierte Usergruppen und native Frontendtechnologien zugeschnitten wurden, so dass durch parallele Mehrfachentwicklung ähnlicher Abläufe siloartige, monolithische Infrastrukturen mit hohen Kosten der Neuerstellung und Wartung entstanden. Die konsequente Nutzung neuartiger, standardisierter Kommunikationskanäle kann dazu beitragen, einer architekturellen Zersplitterung und Schaffung redundanter Zugriffskanäle und Systemstrukturen vorzubeugen.

Literaturverzeichnis

[BAR10] Barth, A. (IETF) (2010): The Web Origin Concept (Draft), November 2010. http://tools.ietf.org/html/draft-abarth-origin-09 (Abgriff 22.04.2014)

[BER14] Berjon, R. et.al. (W3C) (2014): HTML5 – A vocabulary and associated APIs for HTML and XHTML. W3C Candidate Recommendation, 04 February 2014. www.w3.org/TR/html5/ (Abgriff 22.04.2014)

[COW13] Coward, D. (2013): Java API for WebSocket. JSR 356, Version 1.0, 26 April 2013, https://jcp.org/en/jsr/detail?id=356 (Abgriff 22.04.2014)

[FET11] Fette, I.; Melnikov, A. (IETF) (2011): RFC 6455 - The WebSocket Protokoll, December 2011. http://tools.ietf.org/html/rfc6455.txt (Abgriff 22.04.2014)

[GRI13] Grigorik, I. (2013): High Performance Browser Networking. O'Reilly Media. Sebastopol.

[HIC12a] Hickson, I. (W3C) (2012): The WebSocket API. W3C Candidate Recommendation, 20 September 2012. www.w3.org/TR/websockets/ (Abgriff 22.04.2014)

[HIC12b] Hickson, I. (W3C) (2012): Server-Sent Events. W3C Candidate Recommendation, 11 December 2012. www.w3.org/TR/eventsource/ (Abgriff 22.04.2014)

[HIT13] Hitz, M.; Kessel, K. (2013): Einfluss der Nutzung und Auswahl von Open Source Software beim Entwurf einer Multikanal-Architektur. Workshop ISOS 2013. GI-Tagung 2013.

[KES14a] van Kesteren, A. (W3C) (2014): Cross-Origin Resource Sharing. W3C Recommendation, 16 January 2014. www.w3.org/TR/cors/ (Abgriff 22.04.2014)

[KES14b] van Kesteren, A. (W3C) (2014): XMLHttpRequest Level 1. W3C Working Draft, 30 January 2014. www.w3.org/TR/XMLHttpRequest/ (Abgriff 22.04.2014)

[MOU13] Moussa, F; Yoshino, T (2013): Streams API. W3C Working Draft, 05 November 2013. www.w3.org/TR/streams-api/ (Abgriff 22.04.2014)

[TAM12] Tamplin, J.; Yoshino, T. (IETF) (2012): A Multiplexing Extension for WebSockets (Internet-Draft), 28 February 2012. https://tools.ietf.org/html/draft-tamplin-hybi-google-mux-03 (Abgriff 22.04.2014)

[WAN13] Wang,W.; Salim,F.; Moskovits,P. (2013): The Definitive Guide to HTML5 WebSockets. Apress. New York.