Push**Server**

Eine Semesterarbeit zum Thema "Pushing in HTTP"

von Lars Schultz
betreut von Daniel Nydegger, dipl. El. Ing. FH

Hochschule für Technik Zürich

Inhalt

I	**Abstract**	**1**
1	**Zustandbeschreibung**	**2**
1.1	Erste Applikationen	2
1.2	Polling forever?	2
1.3	Warum denn unbedingt HTTP?	3
1.4	Eine Idee	3
2.	**Anforderungen**	**5**
2.1	Anwendungsbereiche	5
2.1.2	Online-Multiplayer-Spiele	5
2.1.3	Multiuser-Applikationen	5
2.1.1	Chats/Nachrichten	6
2.1.3	Serverevent Notification	6
2.1.4	Constantly-changing Data Visualisation	6
2.2	PHP-Entwickler als Zielgruppe	6
2.3	Performance	7
2.4	Zuverlässigkeit	7
2.5	Verfügbarkeit	7
2.6	Skalierbarkeit	8
2.7	Sicherheit	8
2.8	Logik	9
2.8.1	Channel/Subscriber Konzept	9
2.8.2	Synchronisations-Mechanismen	9
2.8.3	Request-Forwarding	9
2.8.3.1	Forwarding Variante	10
2.8.3.2	Subdomain Variante	10
2.8.4	Mandantenfähig	10
3	**Problemanalyse**	**11**
3.1	Server-Applikation	11
3.2	Client-Applikation	11
3.3	PushServer	11
3.3.1	HTTP-Protokoll Parser & Writer	11
3.4	Verbindung Client-Applikation und Server-Applikation	12
3.5	Verbindung Server-Applikation und PushServer	12
3.2	Verbindung PushServer und Client-Applikation	12
3.5.1	Connection-Timeout/-Stabilität	13
4.	**Konkrete Umsetzung**	**14**
4.1	Server-Applikation	14
4.1.1	Schnittstelle zum PushServer	14
4.2	Client-Applikation	14
4.2.1	Content-Frame	15
4.2.2	Receiver-Frame	15
4.2.3	Wrapper-Content	15
4.2.4	Umsetzungsvarianten	15
4.3	PushServer	16
4.3.1	Datenklassen	16

4.3.2 Logik-Klassen.. 17
4.3.3 HTTP-Klassen.. 19
4.3.4 Konfiguration.. 19

5 Verwendete bestehende Libraries........................ 20
5.1 PCREcpp.. 20
5.2 Xerces SAX Parser.. 20

6 Conclusion.. 21
6.1 Ergebnisse.. 22

7 Quellenverzeichnis.. 23

8 Anhang..
8.1 Aufgabenstellung HSZ-T..
8.2 C++ Source..
8.3 Petri-Netz Simulation..
8.4 PHP-Wrapper Source..
8.5 Demo Server-Application..
8.6 Testresultate..
8.6.1 Message-Pushing to one Subscriber..

I Abstract

Diese Semesterarbeit setzt sich mit dem Problem des Pollings in HTTP auseinander und versucht eine echte Alternative zu finden. HTTP ist ein Request/Response-basiertes Protokoll und unterstützt kein Pushing. Für viele web-basierte Anwendungen ist dies jedoch ein Problem. Anwendungen, die eine hohe Interaktion zwischen den Benutzern oder anderen aktiven Komponenten erfordern, verwenden häufig ein zyklisches, automatisiertes Polling. Dieses Prinzip birgt jedoch viele Probleme und ist nicht effizient.

Ein kreatives Konzept, das sich spezielle Eigenschaften des HTTP-Protokolls und der Web-Browser zunutze macht, liefert eine Möglichkeit für echtes Pushing. Dadurch wird es möglich, Nachrichten direkt an den Client zu senden, ohne dass dieser beim Server zyklische Requests absetzt.

Dieses Konzept ist als Webservice in Form eines Push-Servers in C++ umgesetzt. Nutzer des Dienstes sind einerseits Clients, die eine Verbindung zum PushServer aufbauen und andererseits Server-Applikationen, die zum Beispiel in PHP implementiert sein können. Diese Server-Applikationen definieren das genaue Verhalten des PushServer's, indem sie die Businesslogik festlegen und über eine Socket-Verbindung den PushServer steuern.

Auf diese Weise ist es möglich, effizient eine komplexe Server-Applikation in PHP zu implementieren und doch ein echtes zuverlässiges Pushing zu verwenden.

1 Zustandsbeschreibung

Das klassische WorldWideWeb basiert auf dem zustandslosen, polling Protokoll HTTP[1]. Für jede vom Client angeforderte HTML-Seite wird im einfachsten Fall eine TCP-Verbindung aufgebaut und die entsprechende HTML-Seite mit einem HTTP GET-Request vom Server verlangt.

Mit Netscape's Einführung von Cookies[2] 1997, wurde dem Problem des zustandslosen Protokolls entgegengewirkt, da es nun eine einfache und zuverlässige Möglichkeit gab einen Benutzer eindeutig zu identifizieren.

1999 wurde mit dem Update von HTTP/1.0 auf HTTP/1.1[3] ein wichtiger Schritt in Bezug auf die Effizienz des Protokolls gemacht. Es erlaubte das Verwenden derselben TCP-Verbindung für mehrere Requests. Da TCP für längere und Datenintensive Verbindungen entwickelt wurde, ist diese Verbesserung von grossem Vorteil(Slow-Start Problem). Diese Option nennt sich Keep-Alive.

Seit Mozilla Firefox 1.0 die XMLHTTPRequest Schnittstelle von Microsoft's Internet Explorer unterstützt, sind die Grenzen für das AJAX(Asynchronous Javascript and XML) Prinzip stark geschwunden. Spätestens 2005 fand diese Technologie bei vielen, die zuvor daran gezweifelt hatten, Anklang. Zu gross war der Nutzen und die Möglichkeiten.

1.1 Erste Applikationen

Die Entwicklung der ersten Applikationen unterschieden sich von einfachen, statischen Websites in der Weise, dass sie auf Benutzeraktionen reagieren konnten. Ein Besucherzähler oder ein Gästebuch liessen damals die Herzen von Website-Benutzern und vor allem auch den Besitzern höher schlagen. Durch einfache Formulare war es möglich, dass ein Besucher sich in einer Liste eintragen konnte, welche dann wiederum allen Besuchern angezeigt wurde.

Mit der Einführung von AJAX wurde es erstmals Möglich, mit Hilfe von Client-Logik (in JavaScript), Daten vom Server anzufordern, ohne eine ganze Seite oder zumindest ein Frame austauschen zu müssen. Mit XMLHTTPRequest ist es möglich, Daten vom Server zu requesten und diese weiterzuverwenden. Damit lassen sich viele Probleme bewältigen, die mit traditionellem HTML nicht lösbar waren.

1.2 Polling Forever?

Das Protokoll ist aber noch immer ein Polling-Protokoll. Das heisst, dass der Client für jegliche Initiation von Datentransfers verantwortlich ist. Dieses Verhalten ist durchaus sinnvoll für stark frequentierte Websites, da ein Webserver nicht alle Clients kennen kann (oder soll) und noch immer

effizient funktionieren kann. Der Webserver ist also mit jedem Client nur für kurze Zeit beschäftigt und kann danach diese Ressourcen wieder anderen Clients zur Verfügung stellen, bis zum nächsten Request.

Es gibt jedoch einige Anwendungen, wo es wichtig ist, dass der Server dem Client etwas mitteilen kann. Dies kann durch verschiedene Ereignisse ausgelöst werden, auf die später noch genauer eingegangen wird. Für diese Problematik bietet ein Polling-Protokoll nur eine Lösung: Der Client verlangt in regelmässigen Abständen Daten vom Server. Dieser prüft ob etwas ansteht und gibt dem Client seine Antwort. Dieses Konzept wird bereits seit mehreren Jahren erfolgreich eingesetzt, es gibt jedoch Probleme die sich damit nicht lösen lassen.

Wie kurz ist kurz genug? Je kürzer die Abstände zwischen den Anfragen, desto aktueller die Daten. Es entsteht aber auch umso mehr Last auf dem Server. Der Overhead(TCP-Verbindung, HTTP-Header, Serverapplikation) eines einzelnen Requests ist im Vergleich zur Antwort sehr gross. Ausserdem werden viele Anfragen an den Server gestellt, ohne dass dieser effektiv neue Informationen zur Verfügung hat.

1.3 Warum denn unbedingt HTTP?

Die offensichtliche Lösung ist die Umgehung des Problems mit einer Client-Applikation, typischerweise ein Java-Applet, welches nicht an die Limitierungen des HTTP-Protokolls gebunden ist. Da die Schnittstelle zwischen Applets und dem umgebenden HTML jedoch sehr limitiert ist, müsste ein Applet die gesamte Darstellung übernehmen und HTML wird nur noch als Träger verwendet. Diese Technik hat sich jedoch aus verschiedenen Gründen nicht durchgesetzt. Eine komplett Browser-unabhängige Applikation wäre eine Alternative, jedoch besteht bei den Usern eine immer grösser werdende psychologische und praktische Hemmschwelle gegenüber dem Installieren von neuer, unbekannter Software.

1.4 Ein Idee

Wer sagt denn, dass die Antwort auf ein Request in einem kurzen Datenburst erfolgen muss? Interessante Details:

- Browser beginnen die HTML-Daten zu parsen, bevor die Antwort vollständig angekommen ist.
- Es gibt kein Limit in Bezug auf die Antwortzeit des Servers, sofern dieser den Empfang des Requests einmal bestätigt hat.
- Danach wartet der Browser auf alles was der Server noch sendet, ohne Timeout.

Das heisst also: Eine "Ein-Weg"-Verbindung kann beliebig lange vom

Server zum Client aufrechterhalten werden. Der Server kann jederzeit einen weiteren Teil seiner Antwort liefern, welche vom Browser verarbeitet wird. Auf diese Art und Weise wird ein Polling vom Client zum Server also überflüssig.

Um dem Client kein lokales Polling aufzuerlegen, damit er herausfinden kann, ob der Server weitere Daten gesendet hat, gibt es eine weitere interessante Möglichkeit. Falls die Antwort des Servers nicht nur HTML, sondern auch einen JavaScript-Block beinhaltet, wird dieser sogleich geparst und ausgeführt. Damit wird es möglich dem Client mittels eines Event-Handlers, der ausgeführt wird, sobald alle Daten der Teil-Antwort angekommen sind, mitzuteilen dass neue Daten angekommen sind und dieser nun darauf reagieren kann.

Dies beschreibt genau das Gegenstück zum Polling: Die Push-Methode.

2. Anforderungen

Davon ausgehend, dass diese Idee funktioniert, finden sich leicht verschiedene Anwendungsmöglichkeiten, die mit dem Polling-Prinzip an dessen Grenzen stossen. Technische Anforderungen sind mit Sicherheit abhängig von der Anwendung, es lassen sich jedoch einige Aussagen machen, die auf alle Bereiche zutreffen.

2.1 Anwendungsbereiche

Folgende Anwendungsbereiche könnten von einer Push-Methode profitieren.

2.1.1 Online-Multiplayer-Spiele

Die treibende Kraft im Bereich Home-Computing, den Spielen auf dem PC, macht auch vor dem Web nicht halt. Bereits gibt es viele Spiele, die nicht nur vom Internet als Trägerplattform für die Vernetzung von Teilnehmern Gebrauch machen. Diese bauen direkt auf dem WWW auf und erzeugen mit HTML und JavaScript wahre Wunder. Je aktiver diese Spiele aus verschiedenen Genres auf der Kommunikation zwischen Teilnehmern basieren, desto wichtiger ist die Aktualität derselbigen. Verzögerung kann zur Niederlage führen! Viele dieser Spiele verlieren schnell an Reiz wenn man merkt, dass es nicht vom Spielgeschick abhängig ist ob man gewinnt oder verliert, sondern an der Aktualisierungsrate der Polling-Methode. Wenn alle User über neue Informationen oder die Spielzüge eines Konkurrenten, ohne Verzögerung und ohne Abhängigkeit von einem Zeitinterval, benachrichtigt werden könnten, wäre manche Online-Spiel Idee viel attraktiver lösbar.

2.1.2 Multiuser-Applikationen

Von einfachen Web-Applikationen, wie zum Beispiel einem Online-Mail-Client, bis hin zu komplexen Intranet-Business-Applikationen wäre es sehr hilfreich, wenn man mehr darüber wissen würde, was auf der Client-Seite passiert. Oft bleibt nichts anderes übrig, als darauf zu warten, dass der Client einen neuen Request schickt oder mit AJAX einen sogenannten "Heartbeat" zu realisieren, um herauszufinden, ob der User noch immer aktiv ist, oder ob die Applikation verlassen wurde. Dieses Wissen ist zum Beispiel wichtig, falls Objekte von Benutzern gelockt werden können, so dass nur dieser Benutzer damit arbeiten kann. So müssen andere Benutzer nicht auf einen Timeout warten, bis das Objekt automatisch wieder freigegeben wird, sondern wissen zu jedem Zeitpunkt, dass das Objekt wirklich noch in Bearbeitung ist. Dadurch, dass eine TCP-Verbindung geöffnet bleibt, weiss die Serverapplikation, das der User noch in der Applikation arbeitet. Erst wenn die Verbindung geschlossen wird,

hat der User die Applikation verlassen.

2.1.3 Chats/Nachrichten

Kaum eine Anwendung hat schon so früh im Zeitalter des Inernets die Massen fasziniert, wie all die unzähligen Chatrooms in denen 24 Stunden am Tag mehr oder weniger soziale Gespräche geführt wurden. Frühe Varianten dieser Anwendungen waren gar nicht in Web-Browsern integriert, sondern benutzten das Telnet-Protokoll als Träger für die Chat-Botschaften. Schon bald aber entstanden Varianten, die vollständig in HTML-Browsern umgesetzt waren. Diese basierten meist auf dem Meta-Tag "Refresh" um die Nachrichten in einigermassen kurzen Abständen zu empfangen.

Zusätzlich zu den Chats kamen die Messaging-Clients wie ICQ. Diese Client-Applikationen vernetzten tausende von Benutzern. Der Nachteil daran war, das man Software installieren musste, die zumindest in den Anfängen voll von Sicherheitslöchern waren.

Mit einer Client-Applikation, die im Browser integriert funktioniert und sich nicht auf die aufwändige Polling-Methodik verlässt, wäre eine klare Alternative.

2.1.4 Serverevent Notification

Ein Systemadministrator ist unter anderem zuständig für das Monitoring von einzelnen Systemen oder ganzen Netzwerken. Ein Webinterface, welches all diese Komponenten vereinigen und wichtige Veränderungen sofort anzeigen könnte, wäre ein wertvolles Werkzeug.

2.1.5 Constantly-changing Data Visualisation

Gewisse Informationen können niemals aktuell genug sein. Wie zum Beispiel die aktuellen Börsenkurse des eigenen Portfolios oder Messwerte von Sensoren. Mit der Push-Methode könnte man auf die Entwicklung aufwändiger Client-Applikationen verzichten und stattdessen ein schlankes Webinterface zur Verfügung stellen. Eine weitere Anwendung sind verteilte Präsentationen, bei denen die Zuhörer automatisch zur nächsten Folie geführt werden oder auf bestimmte Punkte speziell aufmerksam gemacht werden.

2.2 PHP-Entwickler als Zielgruppe

Serverscriptsprachen wie PHP erlauben eine rasche Entwicklung komplexer Business-Logik und Benutzerschnittstellen. Was aber nicht zur ohne weiteres zur Verfügung steht, sind Threads und Applikationsweite Daten. Falls Daten Applikationsübergreifend zur Verfügung stehen sollen, muss dies anhand von Files oder Datenbanken realisiert werden. Da keine

Threading-Funktionalität zur Verfügung gestellt wird, ist es schwierig und ineffizient eine Serverapplikation zu implementieren, die mehrere Verbindungen akzeptiert und Daten verteilen kann.

Die Komplexität einer Multi-Thread Anwendung soll auf das Wesentliche reduziert werden, um eine effiziente Integration zu gewährleisten. Dies bedeuted, dass keinerlei Business-Logik integriert und nur die notwendige Logik für das verteilen der Nachrichten implementiert werden soll.

Die Aufteilung der Aufgaben erfordert eine effiziente Kommunikationsmöglichkeit zwischen diesen Anwendungen.

Im Folgenden wird die PHP-Applikation Server-Applikation genannt und die Verteiler-Applikation PushServer.

2.3 Performance

Ansprüche an die Performance variieren stark, je nach Anwendung, insbesondere was die Anzahl gemeinsamer Nutzer betrifft. Bei Chats und Online-Spielen kann die Zahl von ein paar 100 bis auf ein paar 1000 steigen wobei bei den anderen Anwendungsbereichen die Benutzerzahlen unter 100 bleiben.

Die Daten, die übermittelt werden sollen, sind typischerweise in kleine Pakete aufgeteilt. Es ist also mit vielen Push-Requests mit kleiner Datenmenge zu rechnen. Um dies effizient zu behandeln, muss ein schnelles "Demultiplexing" ermöglicht werden.

2.4 Zuverlässigkeit

Je nach Anwendung ist es wichtig, dass jede Nachricht übermittelt wird. Auch Nachrichten, die nicht sofort übermittelt werden können weil keine Verbindung zum Empfänger besteht. Diese sollten idealerweise zwischengespeichert werden bis die Verbindung wieder steht.

Falls wegen Überlastung (Überschreiten der maximal zulässigen Verbindungen) eine Nachricht nicht übermittelt werden kann, so muss dies der aufrufenden Server-Applikation mitgeteilt werden, damit diese darauf reagieren kann.

2.5 Verfügbarkeit

Damit zu jeder Zeit genügend Verbindungen zur Verfügung stehen, wäre es von Vorteil, die Anzahl Push-Verbindungen, die für längere Zeit offenbleiben, pro Teilnehmer zu limitieren. Dies stellt im Normalfall der Browser sicher, da dieser aus Gründen der Fairness in der Standardeinstellung nur 2 gleichzeitige Verbindungen zum selben Host zulässt. Trotzdem sollten Vorkehrungen getroffen werden, damit nicht ein einzelner Client zu viele offene Verbindungen hält. Bei einer DOS-Attacke

würde dies helfen die Verbindungen sofort wieder zu schliessen, ohne diese zu behandeln.

2.6 Skalierbarkeit

Die Skalierbarkeit ist stark verknüpft mit der Performance. Da die Businesslogik auf eine andere Applikation ausgelagert wird, wird der PushServer stark entlastet. Dadurch sollte es möglich sein, eine grosse Anzahl von Verbindungen ohne Probleme zu verarbeiten. Auf eine verteilte Architektur des PushServers wird verzichtet.

2.7 Sicherheit

Der direkte Zugriff auf den PushServer soll nur von der Server-Applikation erfolgen. Jegliche administrative Kommunikation soll vom Client an die Server-Applikation geschickt werden. Einzig die Push-Verbindung wird direkt vom Client zum PushServer aufgebaut. Die Server-Applikation muss sich beim PushServer mit einem Passwort identifizieren können.

Die Daten müssen nicht zwingend verschlüsselt übertragen werden, da diese meist nur von kurzer Bedeutung sind und keine sensitiven Informationen enthalten. Dies variiert je nach Anwendung. Bei Bedarf müsste ein SSL-Tunnel aufgebaut werden.

2.8 Logik

Die Business-Logik wird vollständig in der Server-Applikation implementiert. Der PushServer stellt minimale Funktionalität zur Verfügung.

2.8.1 Channel/Subscriber Konzept

Um die Verteilung von Nachrichten an mehrere Teilnehmer möglichst effizient zu gestalten, wird ein Channel/Subscriber Konzept verwendet. So ist ein Teilnehmer ein Subscriber, also ein Abonnent, von verschiedenen Informationskanälen, den Channels. Diese Kanäle dienen der Server-Applikation zur einfachen Verteilung von Nachrichten, die an viele Teilnehmer gleichzeitig gerichtet sind. Nachrichten die nicht sofort versendet werden können, werden zwischengespeichert bis der Teilnehmer sich wieder beim PushServer anmeldet.

2.8.2 Synchronisations-Mechanismen

Der PushServer verschiedene Funktionen über die Administrations-Schnittstelle an, um Informationen über dessen Business-Objekte abzurufen.

Ausserdem kann die Server-Applikation sogenannte Notifier registrieren. Diese werden aufgerufen sobald Veränderungen auftreten, die nicht von der Server-Applikation ausgelöst wurden. Dazu gehört zum Beispiel das Verbinden und das Trennen der Verbindung vom Client zum PushServer.

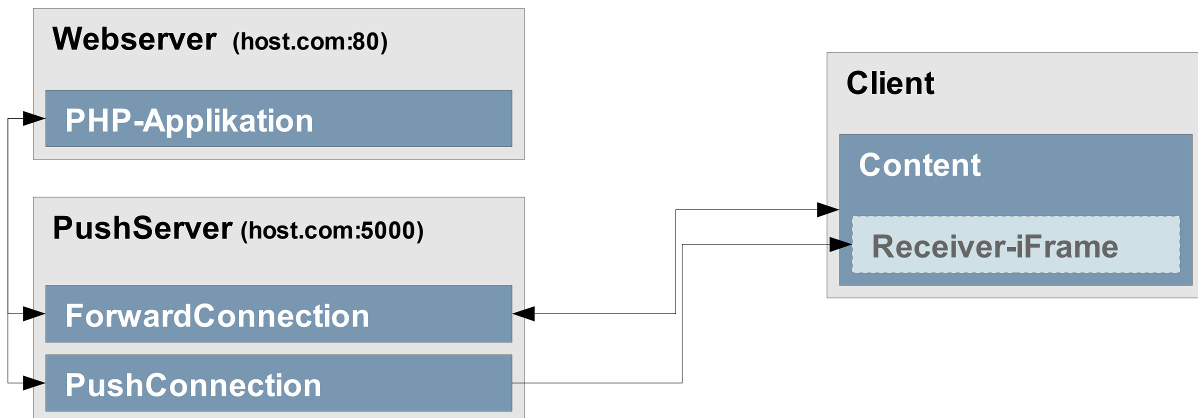
2.8.3 Request-Forwarding

Aus Sicherheitsgründen erlauben die Browser (HTML-)Frameübergreifende Javascript-Interaktion (lesender und schreibender Zugriff auf benachbarte oder übergeordnete Frames) nur, falls die Quellen(URL) der Frames vom selben Host stammen. Diese Einschränkung nennt sich "Same Origin Policy"[4].

Wenn also die Server-Applikation in PHP implementiert auf dem Webserver unter Port 80 laufen würde und der PushServer auf demselben Host unter Port 5000 auf Requests warten würde, dann kann ein Frame, das eine Verbindung zum PushServer aufgebaut hat, nicht mit einem Frame kommunizieren, dessen Inhalt vom Webserver stammt.

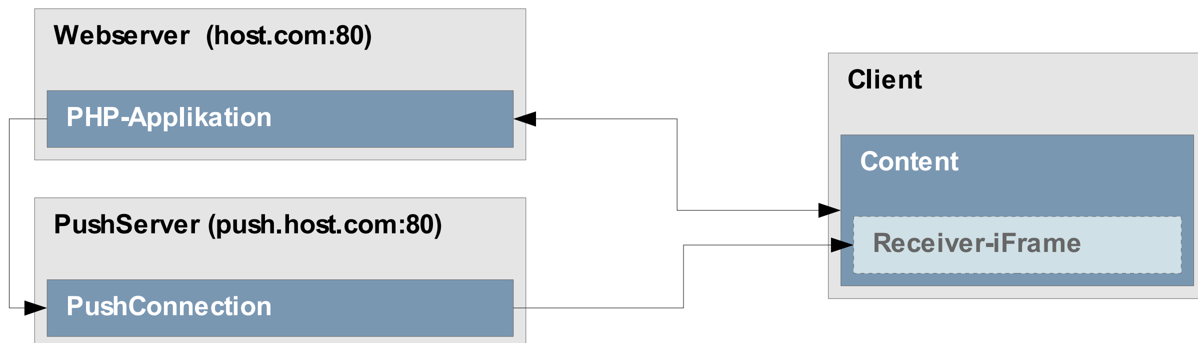
Um diese Einschränkung zu Umgehen gibt es zwei Möglichkeiten. Die Subdomain Variante ist technisch aufwändiger. Darum erlaubt der PushServer das Forwarding von Requests auf den eigentlichen Webserver, so dass Requests in allen Frames an den PushServer zu gehen scheinen.

2.8.3.1 Forwarding Variante



Der Client bezieht die Daten bei beiden Frames vom PushServer. Dieser leitet Requests des Clients für das Content-Frame an den Webserver weiter. Damit wird die Policy erfüllt.

2.8.3.2 Subdomain Variante



Diese Variante basiert auf dem Prinzip, das Subdomains von der Same-Origin-Policy akzeptiert werden. Ein DNS hat ein explizites Mapping von host.com auf eine IP und für push.host.com auf eine andere IP. Da beide Server auf demselben Port hören, ist es möglich die Einschränkung zu Umgehen.

2.8.4 Mandantenfähig

Um nicht mehrere Instanzen des Servers auf verschiedenen Ports starten zu müssen, können mehrere Mandanten(Mandators) konfiguriert werden. Pro Mandator werden einzelne Channels und Subscriber verwaltet. Zudem kann sich die Notifier- und Forwarding-Konfiguration unterscheiden. Die restlichen Einstellungen, wie zum Beispiel die maximale Anzahl Verbindungen, sind Serverübergreifend gültig.

3 Problemanalyse

Vor der Umsetzung sind bei der Analyse und der Implementierung verschiedener Prototypen technische Probleme aufgetreten. Diese sind hier zu jeder Komponente aufgeführt und erklärt.

3.1 Server-Applikation

Die Server-Applikation sollte keine Schwierigkeiten haben. Sie muss die sporadischen Polling-Requests der Client-Applikation beantworten. Die Verwaltung der Benutzer kann in einem gewissen Masse dem PushServer überlassen werden. Falls die Applikation es jedoch erfordert gewisse Meta-Daten, wie z.B. den Namen, zu einem Client zu speichern, dann muss dies in der Server-Applikation anhand der eindeutigen Subscriber-ID, die dem Client zugeordnet wird, geschehen.

3.2 Client-Applikation

Je nach Implementation der Client-Applikation bestehen gewisse Probleme im Bezug auf das aufrechterhalten der Verbindung zum PushServer. Falls die Client-Applikation den Content und das Receiver-Frame nicht in einem umgebenden Frameset verpackt wird die Verbindung immer wieder ab- und nach dem Reload wieder aufgebaut.

3.3 PushServer

3.3.1 HTTP-Protokoll Parser & Writer

Es wurden verschiedene bestehende Libraries evaluiert. Die Haupt-probleme damit waren:

- Eine Antwort konnte oft nur als ganzes an den Client zurück-geschickt werden. Die PushConnection verwendet aber das Transfer-Encoding "chunked".
- Die Struktur eines Requests konnte nicht vollständig wieder-hergestellt werden. Dies beeinflusste das Forwarden eines Client-Requests.
- Die Einflussnahme auf den Parseprozess und die frühe Erkennung der Request-Art. Dies ist für den *RequestSwitch* wichtig, da dieser bereits nach dem lesen der ersten Requestzeile wichtige Entscheidungen treffen kann.

Die Entscheidung einer Neu-Implementation des HTTP Protokolls und des Sockets wurde auch aus persönlichen Gründen getroffen. Zum einen war es der Reiz der Herausforderung, zum anderen um das Verständnis zu fördern, wie das Protokoll funktioniert.

3.4 Verbindung Client- und Server-Applikation

Bei dieser Verbindung sollte es keine Probleme geben. Diese Polling-Requests werden mittels HTTP-Keep-Alive performant gehalten. Die Kommunikation erfolgt vor allem über GET-Parameter und POST-Formularwerte.

Diese Verbindung ist beliebig skalierbar, da der Webserver bei Überlastung einfach per Load-Balancer verstärkt werden kann. Mit einem zentralen Datenspeicher bei dem auch die serialisierten Sessions geshared werden, ist das kein Problem.

3.5 Verbindung Server-Applikation und PushServer

Das Hauptproblem bei dieser Verbindung besteht darin, das die Serverapplikation jedesmal eine neue Verbindung aufbauen muss, wenn ein Client-Request oder ein Server-Event einen Push erfordert. Die Server-Applikation wird danach serialisiert und die Verbindung getrennt, bis ein neuer Request der Client-Applikation wieder einen Push auslöst.

PHP bietet sogenannte persistent-connect Funktionalität für verschieden Verbindungen an. Unter anderem auch für TCP-Socket Verbindungen.

Ein echtes Connection-Pooling mit Keep-Alive Möglichkeit ist wünschenswert. Apache bietet dafür eine Möglichkeit: apr_reslist[5]. Die Implementation dieses Interface und der Umsetzung für PHP ist jedoch nicht geplant. Was aber als Option berücksichtigt wird, ist die Möglichkeit des Keep-Alive.

Die curl-Library, welche auf einfache Weise mit PHP verwendet werden kann, erlaubt Keep-Alive innerhalb eines PHP-Skriptes. Sobald dieser Skript beendet wird, wird jedoch auch die Verbindung geschlossen.

3.2 Verbindung PushServer und Client-Applikation

Der Browser erlaubt nur eine gewisse Anzahl an Verbindungen zum selben Server pro Port. Dies ist je nach Implementation unterschiedlich und ist aus Gründen der Fairness eingeführt worden. So wird ein Server nicht mit einer hohen Anzahl gleichzeitigen Verbindungen überflutet. Im Falle des PushServers ist des jedoch ein Problem, da z.B. Firefox nur eine und Internet Explorer maximal 2 Verbindungen gleichzeitig auf dem selben Port öffnet. Das heisst: Wenn in einem Fenster die Applikation bereits läuft, kann in einem anderen unter Umständen keine Verbindung mehr aufgebaut werden. Diesem Problem kann auf der Client-Seite unter Umständen durch Anpassen der Einstellungen behoben werden, ist aber eher unerwünscht. Eine serverseitige Lösung ist das akzeptieren von Verbindungen auf mehreren Ports oder virtuellen Sub-Domains, was die Server-Applikation dann über eine Art Balancing kontrollieren und dem Client entsprechend kommunizieren muss.

Dies erschwert das Anbieten eines Push-Dienstes für eine Server-Applikation im Sinne eines WebServices, da mehrere Server- und somit auch Client-Applikationen auf denselben Service-Provider zurückgreifen könnten.

Eine Erweiterungsoption für den PushServer wäre das Verwalten der bereits verbundenen Ports mit einem Client. Diese Information soll dann der Server-Applikation zur Verfügung stehen. Problem: Server-Applikation übergreifende identifizierung des Clients. IP kann wegen NAT(mehrere verschiedene Clients "hinter" der selben IP) nicht verwendet werden.

Eine Möglichkeit ist ein standardisiertes Cookie, welches von der Server-Applikation abgefragt und/oder an den PushServer weitergeleitet werden kann. Falls die Server-Applikationen aber auf verschiedenen Hosts laufen, funktioniert diese Lösung nicht, da die Cookies nur an den ausstellenden Host zurückgesendet werden.

Eine weitere Möglichkeit wäre ein HTTP-basiertes Balancing mittels Redirects. Dies erfordert, das der PushServer immer auf demselben Port von der Client-Applikation angesprochen wird, damit der PushServer entscheiden kann, auf welchen Port er den Client umleitet. Dies ist wohl die interessanteste Variante, da so mittels eines Cookies der Client eindeutig identifiziert werden kann, ohne das die Server-Applikation etwas dazu beitragen muss.

Auf dieses Problem wurde bei der Umsetzung nicht eingegangen.

3.5.1 Connection-Timeout/-Stabilität

Bei anderen, ähnlichen Projekten gab es mit der Verbindung gewisse Schwierigkeiten. Einige Browser schliessen die Verbindung ohne Aufforderung des Servers nach einer gewissen Zeit. Eine mögliche Lösung dafür ist das senden eines Daten-Pulses, einer kleinen Datenmenge, die in einem Interval zwischen den echten Push-Daten an den Client gesendet werden, damit dieser die Verbindung offen hält.

Dieses Problem muss noch genauer getestet und analysiert werden.

4. Umsetzung

4.1 Server-Applikation

4.1.1 Schnittstelle zum PushServer

Die Server-Applikation soll über eine TCP-Verbindung mit dem PushServer kommunizieren. Das zugrundeliegende Protokoll soll HTTP sein, wobei für Funktionsaufrufe das POST-Format verwendet wird. Auf der Basis des Keep-Alive Parameters soll es möglich sein, mehrere Kommandos abzusetzen über dieselbe Verbindung. Die Authentifizierung der Server-Applikation erfolgt jedoch für jeden Request.

Die Wahl des HTTP-Protokolls liegt nahe, da es bereits viele Libraries gibt, die dieses Protokoll unterstützen und das aufrufen von Funktionen ähnlich dem Requesten von Dokumenten auf einem Webserver ist. Eine reduzierte Variante davon ist einfach zu implementieren.

Die Applikations-Identifizierung/Authentifizierung soll, falls nötig, über eine HTTP-Authentifizierung[6] gewährleistet sein.

Für die Sicherstellung des Datenschutzes während der Übertragung bietet sich das SSL Protokoll an. Die Realisierung ist aber nicht geplant.

Eine PHP Wrapper-Klasse soll die Implementierung der Schnittstelle vereinfachen. Für die Verbindung wir die curl-Library verwendet werden.

4.2 Client-Applikation

Die effektive Umsetzung der Client-Applikation ist abhängig von der benötigten Flexibilität, Komplexität und Beständigkeit der Anwendung. Eine Implementation besteht aus mindestens 2 Komponenten: Einem Content-Frame und einem Receiver-Frame.

Der Begriff Frame bezieht sich nicht zwingend auf ein effektives HTML-Frame sondern als Teil der Applikation, der einen eigenen Request an den Server sendet und dessen Antwort empfängt. Mögliche Umsetzungen sind Framesets oder iFrames, aber auch eigenständige Windows, die per JavaScript geöffnet wurden.

4.2.1 Content-Frame

Dieses Frame übernimmt die effektive Darstellung des GUI's. Je nach Umsetzung, ist diese Aufgabe auf mehrere Content-Frames aufgeteilt. Daten vom PushServer sind in diesem Frame visualiserbar oder lösen eine Aktion aus. Der HTTP-Request ist immer an die Server-Applikation gerichtet.

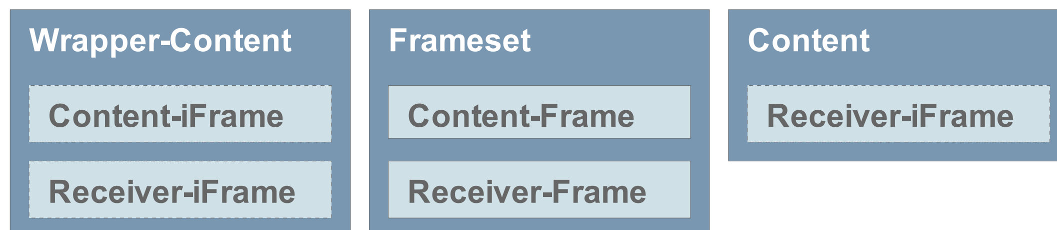
4.2.2 Receiver-Frame

Die Client-Applikation baut in diesem Frame eine Verbindung zum PushServer auf. Daten werden in dieses Frame gepusht. Um ohne lokales Polling auf diese Daten zuzugreifen, werden die Daten von der Server-Applikation in Javascript-Blöcke gepackt, die eine Handler-Funktion auslösen, sobald der Browser den Block verarbeitet.

4.2.3 Wrapper-Content

Ein Wrapper-Content mit zwei iFrames oder auch ein Frameset mit den jeweiligen Frames ermöglicht es, die Receiver-Frame Verbindung aufrechtzuerhalten selbst wenn die Daten im Content-Frame durch einen erneuten Request komplett ausgetauscht werden.

4.2.4 Umsetzungsvarianten



Grundsätzlich gibt es drei verschiedene Varianten wie eine Client-Applikation aufgebaut sein kann.

In der ersten Variante beinhaltet ein Wrapper-Content das Content-iFrame, in welchem die Daten dargestellt werden und das Receiver-iFrame welches Daten vom PushServer empfängt.

Die zweite Variante ist ähnlich aufgebaut wie die erste, nur werden normale Frames verwendet. Dabei muss aufgepasst werden, das die Seite nicht vollständig neu geladen wird, sondern nur die Daten im Content-Frame ausgetauscht werden, da sonst die Push-Verbindung bei jedem Content-Update ab- und wieder neu aufgebaut wird.

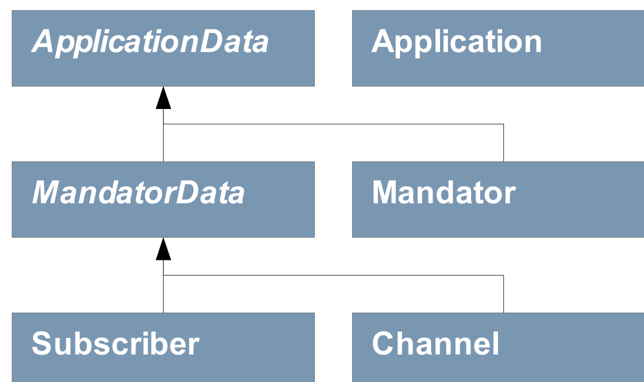
In der dritten Variante gibt es keinen Wrapper-Content sondern die Daten werden direkt dargestellt.

Das Receiver-Frame ist jeweils unsichtbar, da darin im Normalfall keine Daten dargestellt, sondern nur empfangen werden.

4.3 PushServer

Die Umsetzung des PushServers erfolgte mit C++ und ist mit dem GNU-Compiler g++ unter Linux kompilierbar. Eine Kompilierung unter Windows ist nicht ohne weiteres möglich, da gewisse Komponenten anders gebaut werden müssten. Dazu gehören zum Beispiel die auf BSD-Sockets basierenden Funktionen oder die pthread-Library. Als Entwicklungs-umgebung diente ein einfacher Editor, wobei nur die Syntax-Highlighting Funktionalität verwendet wurde. Jede Klasse ist in ein Header- und ein Code-File aufgeteilt. Der Namensraum "PushServer" dient der Differenzierung zu anderen Bibliotheken, wie dem verwendeten Regular-Expression-Wrapper (pcrecpp) von Google und dem SAX-XML-Parser aus der Xerces-Bibliothek des Apache-XML-Projects. Zudem werden verschiedene Funktionen aus dem eigenen "Widescreen" Namespace verwendet, welche verschiedenste Funktionalität zur Verfügung stellt.

4.3.1 Datenklassen



Bei den Datenklassen handelt es sich um Container für Business-Logik Daten.

Die *Application* Klasse ist als Singleton implementiert. Die Klasse instanziert und ver-waltet die *Mandators* die in der Konfiguration spezifiziert wur-den. Ausserdem ist sie für die instanzierung der verschiede-nen *ThreadPools* und deren Limits zuständig. Die Limits beziehen sich auf die maximal zulässigen und der minimal, zu Beginn bereits instanzierten, *WorkerThreads*. Diese Limits lassen sich in der Konfiguration einstellen. Auch startet sie den *ServerSocketListener* der auf eingehende Requests wartet.

In der *Mandator* Klasse werden alle zugehörigen *Channels* und *Subscribers* verwaltet. Zudem sind darin die Einstellungen gespeichert, welche per Mandant konfigurierbar sind(Forwarding & Notifiers). Ein Mandator besitzt ausserdem eine eindeutige ID welche in der Konfiguration festgelegt wird. Diese ID wird bei allen Requests als Prefix verwendet. Damit ist klar für welchen Mandator ein Request gilt.

Die *Channel* Klasse führt eine Liste aller Subscriber die diesen Datenkanal abonniert haben. Ausserdem muss für jeden Channel eine *Mandator*-übergreifend eindeutige ID definiert werden.

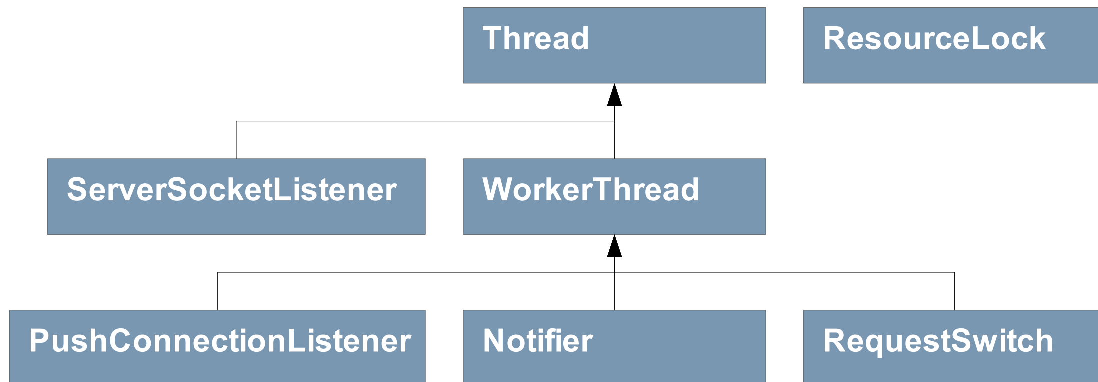
Die Subscriber Klasse führt eine Liste aller abonnierten Channels. Zudem weiss eine Instanz ob der Client zur Zeit tatsächlich und mit welcher

PushConnection mit dem PushServer verbunden ist.

Channel und *Subscriber* sind von der abstrakten Klasse *MandatorData* abgeleitet. Damit wird jeder Instanz ein *Mandator* zugeordnet.

MandatorData und *Mandator* leiten von der abstrakten Klasse *ApplicationData* ab. Diese speichert eine Referenz auf die *Application* und legt ausserdem das ID Interface fest.

4.3.2 Logik-Klassen



Die *Thread* und *ResourceLock* Klasse bilden einen Wrapper für die POSIX pthread-Library.

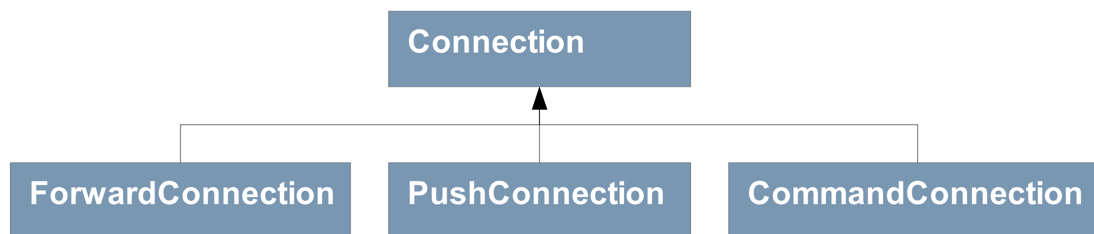
Ein *WorkerThread* bietet eine allgemeine Schnittstelle um eine Aufgabe nach Bedarf zu erfüllen. Jeder WorkerThread wird einem ThreadPool zugeordnet, bei dem er sich wieder registriert sobald seine Aufgabe erledigt ist.

Die Klasse *ServerSocketListener* hört auf einem bestimmten Port auf HTTP-Requests. Sobald ein Request akzeptiert wird, übergibt der Listener den Request an einen *RequestSwitch* aus dem *RequestSwitchPool*.

Der *RequestSwitch* verwendet einen *HTTPRequest* um herauszufinden welcher Art der Request ist. Danach wird die *HTTPRequest*-Instanz an ein entsprechend instanziertes *Connection*-Objekt übergeben.

Mit einem *PushConnectionListener* wird geprüft, ob der Client die *PushConnection* noch immer offen hält oder ob diese geschlossen wurde. Dies kann zum Beispiel beim Verlassen der Website oder dem Schliessen des Browsers geschehen. Sobald die Verbindung geschlossen wird, teilt der Listener dies der *PushConnection* mit.

Wenn ein Client die Verbindung zum PushServer auf oder abbaut, wird dies mittels eines *Notifiers* der Server-Applikation mitgeteilt.

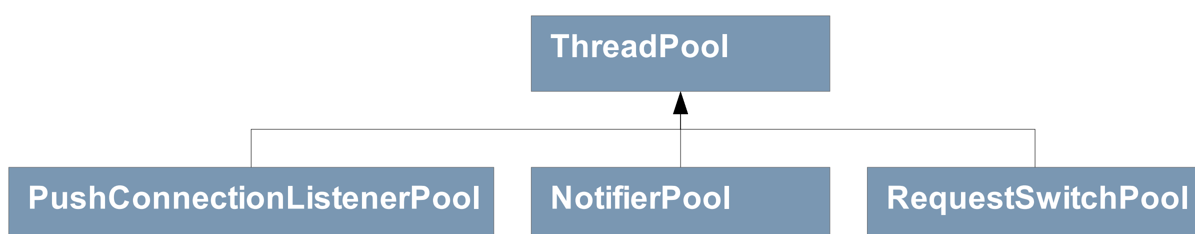


Eine *ForwardConnection* wird verwendet um den eingehenden Request an einen Webserver weiterzuleiten und dessen Antwort wiederum an den Client zurückzusenden.

Die *CommandConnection* nimmt Requests von der Server-Applikation entgegen und verarbeitet verschiedene Kommandos. Diese werden zum Beispiel verwendet um einen Subscriber oder einen Channel hinzuzufügen. Erfolgreiche Verarbeitung wird mit einer "200 OK"-HTTP-Status Meldung beantwortet. Zusätzlich wird als Inhalt eine detaillierte XML-Antwort gesendet.

Sobald ein Client eine Verbindung zum PushServer aufbaut und diese vom Typ "psh" ist, wird ein *PushConnection*-Objekt instanziert. Dieses Objekt benachrichtigt die Server-Applikation mit einem *Notifier* vom Auf- und später vom Abbau der Verbindung. Die *Connection* wartet darauf das Daten an den Client gesendet werden sollen. Falls die Verbindung vom Client geschlossen wird, weckt der *PushConnectionListener* die PushConnection. Falls der Subscriber von einer *CommandConnection* entfernt wird, wird die *PushConnection* geschlossen.

Jede *Connection* wird von einem *RequestSwitch* instanziert.



Ein *ThreadPool* verwaltet WorkerThreads. Beim start werden die als Minimum angegebenen Threads instanziert. Falls mehr Threads benötigt werden, werden diese instanziert bis das angegebene Maximum erreicht wird.

4.3.3 HTTP-Klassen

Die *Socket* Klasse dient als objektorientierte Schnittstelle zu den BSD-Sockets. Unter anderem wird damit das korrekte Verarbeiten des Buffers und das Lesen pro Zeile ermöglicht. Diese Klasse ist HTTP-unabhängig insofern, als das sie keine HTTP spezifischie Funkionalität implementiert.

HTTP	HTTPHeader
HTTPRequest	HTTPRequestData
HTTPResponse	Socket

Die statische Klasse *HTTP* bietet Zugriff auf die Protokoll-spezifischen Eigenschaften: *Protocol*, *Method* und *Status*. Damit ist eine saubere Definition der Schnittstelle für die anderen Klassen möglich.

Die Klassen *HTTPRequest* und *HTTPResponse* verwenden die Klasse *HTTPHeader* um eine HTTP-Anfrage und eine HTTP-Antwort gemäss Protokollspezifikation zu verarbeiten. Ein Request oder Response Objekt kann zum lesen oder schreiben mit einem Socket oder verwendet werden.

Die Klasse *HTTPRequestData* wird verwendet um POST/GET und Cookie Daten aufzunehmen und auszugeben.

4.3.4 Konfiguration

Die Konfiguration des PushServers sowie der Mandators erfolgt über eine XML-Datei(config.xml), das sich im Startverzeichnis befunden muss. Das Format ist dem im Anhang beigefügten Beispiel zu entnehmen.

5 Verwendete bestehende Libraries

5.1 PCREcpp

Der PCREcpp-Wrapper von Google Inc. eignet sich hervorragend um ein textbasiertes Protokoll wie HTTP zu parsen. Der Code bleibt übersichtlich dank kompakter Formulierung von Regular Expressions. Ausserdem können Änderungen schnell und einfach vollzogen werden.

5.2 Xerces SAX Parser

XML eignet sich hervorragend um strukturierte Daten in einem einfachen Textfile zu bearbeiten. Dieser Vorteil wurde für das Konfigurationsfile verwendet.

Xercesc[7] ist ein Teil des Apache XML Projects und bietet eine standardisierte Schnittstelle zu verschiedenen XML-Parser. Ein SAX-Parser ist für den einmaligen Zugriff auf XML-Daten optimiert. Die XML-Konfiguration wird vom SAX-Parser eingelesen und in eine C++-Datenstruktur (Widescreen::NodeMap) gefüllt welche einen einfachen und direkten Zugriff erlaubt.

6 Conclusion

Abschliessend möchte ich über meine persönliche Meinung und Erfahrungen zu diesem Projekt berichten.

Die grössten Schwierigkeiten bei der Implementation ergaben sich durch meine geringe Erfahrung mit C++. Viele Stunden habe ich investiert um ein einfaches Problem zu lösen, nur weil ich diesem in C++ noch nie zuvor begegnet bin. So zum Beispiel der Umgang mit umfangreichen Klassenstrukturen und dem dazugehörigen Makefile, die Feinheiten von Namespaces oder dem Einsatz von externen Libraries. Nach vielen frustrierenden Versuchen hat es dann aber schlussendlich immer geklappt. Ich habe mir viel Mühe gegeben, die Qualität des Codes nicht durch meine geringe Erfahrung zu beeinträchtigen. So habe ich verschiedene Tips aus Büchern[8][9][10] und auch Erfahrungen mit anderen Programmiersprachen einfliessen lassen.

Durch den gezielten und bewussten Umgang mit Pointers und Referenzen konnten Memory-Leaks hoffentlich verhindert werden.

Der Einsatz der pthread-Library und des PCREcpp-Wrappers ist sehr intuitiv und vollständig Dokumentiert. Um die Konfigurationsdatei mit dem SAX-Parser von Xercesc einzulesen, verbrachte ich frustrierende Stunden, welche sich aber zum Ende gelohnt haben.

Der Zeitaufwand lag im Rahmen der für die Semesterarbeit vorgeschriebenen 100 Stunden, wobei durch die geringe Erfahrung mit C++ ein paar zusätzliche Stunden angefallen sind. Diese waren jedoch in der Planung der Arbeit berücksichtigt und so konnte ich diese nach gut 2 Monaten für mich zufriedenstellend abschliessen.

Ich konnte aus Zeitmangel keine representativen Belastungstests durchführen. Auch ausführliche und dokumentierte Tests kamen viel zu kurz. Jedoch waren die Erfahrungen während der Entwicklung und den abschliessenden Anwendungstests mit der Demo-Applikation so überzeugend, das dies wohl kein Problem darstellen dürfte.

6.1 Ergebnisse

Ein paar für mich eindrückliche Zahlen möchte ich hier noch festhalten:

Umgebung: 100Mbit LAN, Server: Via EPIA-M 500Mhz, mini-fanless.

- Mit dem Werkzeug Apache-Benchmark(ab), habe ich Nachrichten an einen echten Client gesendet, dieser empfing 750 Nachrichten pro Sekunde, bei 30 simulierten gleichzeitig ausgehenden Verbindungen. Diese Zahlen hängen natürlich stark von der Umgebung des PushServers und des Clients ab, zeigen sie jedoch die Robustheit und Belastbarkeit sehr schön.
- Die Latenz zwischen dem versenden einer Nachricht an einen anderen Client in der Demo-Applikation liegt zwischen 20 und 50ms. Sie variiert stark, je nach Last des Netzes.
- Die in der Aufgabenstellung formulierte Anforderung, das der Server 250 Verbindungen gleichzeitig aufrecht erhalten sollte, konnte ich nicht hinreichend testen.

7 Quellenverzeichnis

1. Hypertext Transfer Protocol – HTTP/1.0 / Maii 1996 / RFC
   http://www.ietf.org/rfc/rfc1945.txt
2. HTTP State Management Mechanism / Februar 1997 / RFC
   http://www.ietf.org/rfc/rfc2109.txt
3. Hypertext Transfer Protocol -- HTTP/1.1 / Juni 1999 / RFC
   http://www.ietf.org/rfc/rfc2616.txt
4. The Same Origin Policy / August 2001 / Mozilla, Jesse Ruderman
   http://www.mozilla.org/projects/security/components/same-origin.html
5. Dynamic Resource Pools: apr_reslist / April 2006 / Apache Tutor
   http://www.apachetutor.org/dev/reslist
6. HTTP Authentication: Basic and Digest Access Authentication / Juni 1999 / RFC
   http://www.ietf.org/rfc/rfc2617.txt
7. Xerces-C++ / The Apache XML-Project
   http://xml.apache.org/xerces-c/
8. C++ - Lernen und Professionell anwenden /2002, mitp Verlag/ Peter Prinz, Ulla Kirch-Prinz
9. C++, UML und Design Patterns / 2005, Addison-Wesley Verlag / Helmut Herold, Michael Klar, Susanne Klar
10. Die C++ Standard-bibliothek / 2001, Springer Verlag / Stefan Kuhlins, Martin Schader

8 Anhang

8.1 Aufgabenstellung HSZ-T

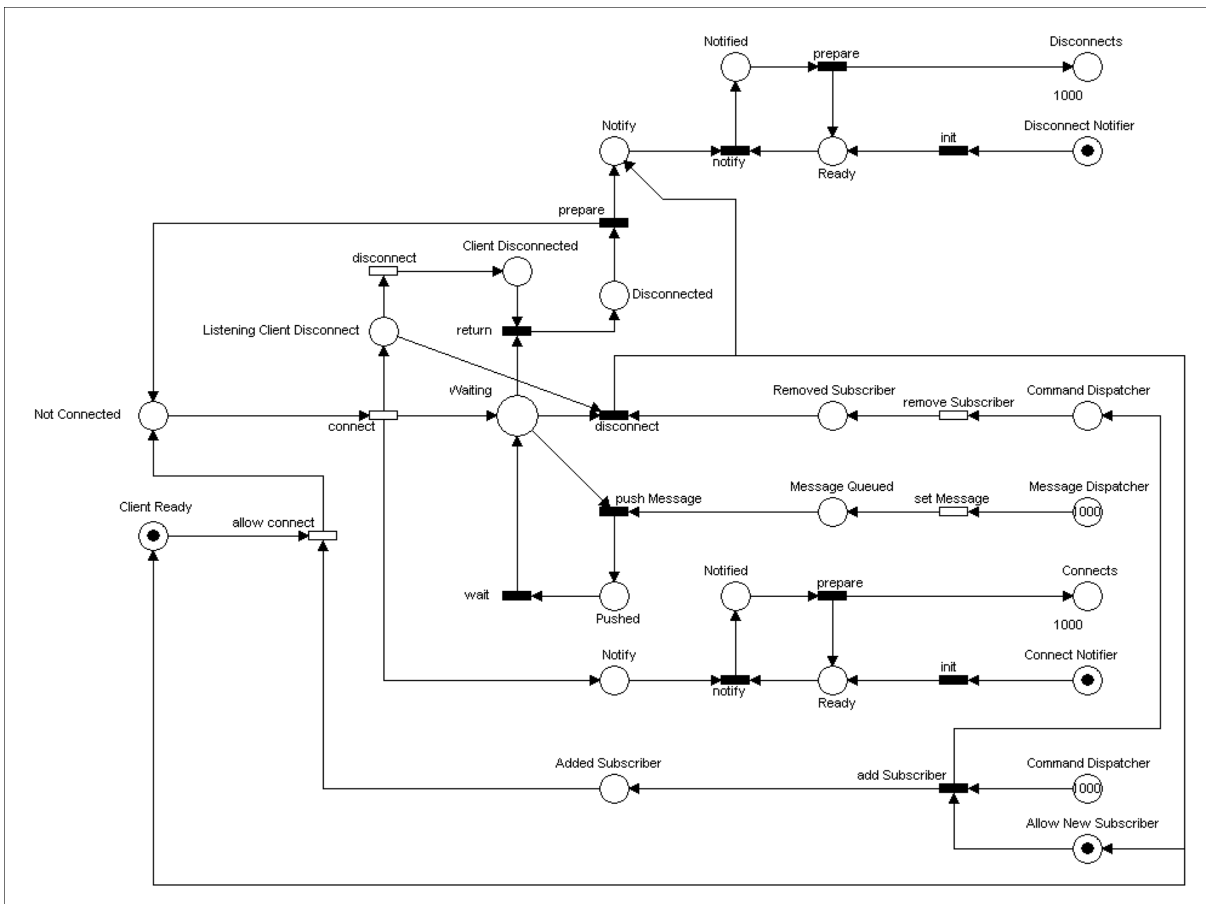
Dies ist die formale Aufgabenstellung für die Semesterarbeit.

8.2 C++ Source

Der komplette Source-Code und dessen Dokumentation.

8.3 Petri-Netz Simulation

Mittels diesem Petri-Netz wird die parallele Verarbeitung von Requests simuliert. Besonders die Interaktion mit der "Wait"-Stelle musste sorgfältig modelliert werden, ist sie auch das Herzstück im PushServer, die PushConnection.

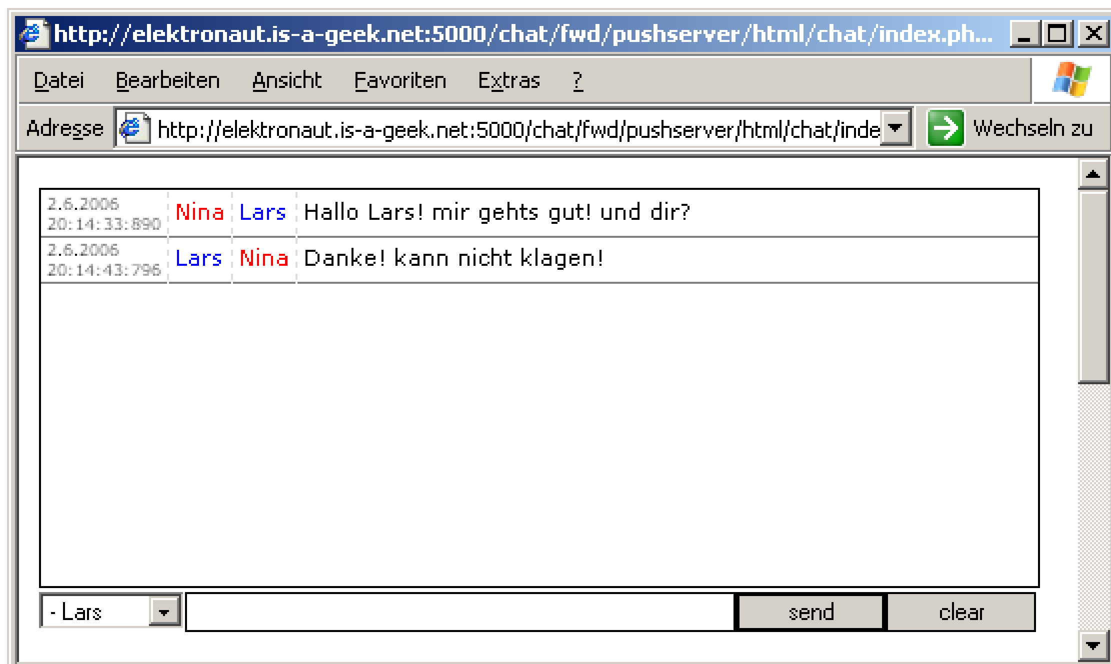


8.4 PHP-Wrapper Source

Dies ist der Source-Code für die PHP Wrapper-Klasse, die für die Kommunikation zwischen Server-Applikation und PushServer verwendet werden kann.

8.5 Demo Server-Application

Eine einfache Server-Applikation, die alle Funktionen des PushServers verwendet.



8.6 Testresultate

8.6.1 Message-Pushing to one Subscriber

```
Server Software:
Server Hostname:        minnie
Server Port:            5000

Document Path:
/chat/cmd/?command=pushSubscriberMessage&subscriberId=lars&message=test+
Document Length:        45 bytes

Concurrency Level:      30
Time taken for tests:   1.339133 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      110160 bytes
HTML transferred:       45900 bytes
Requests per second:    746.75 [#/sec] (mean)
Time per request:       40.174 [ms] (mean)
Time per request:       1.339 [ms] (mean, across all concurrent requests)
Transfer rate:          79.90 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        9   18   5.1     18      30
Processing:    11   20   5.1     21      33
Waiting:        0   13   7.8     14      30
Total:         38   38   1.2     38      42
```