



Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Dokumentation

**Bianca Niklass, Amanpreet Kaur, Christof Rode, Rene
Kretschmer**

Riddley-Diddley

Bianca Niklass, Amanpreet Kaur, Christof Rode, Rene Kretschmer

Riddley-Diddley

Dokumentation eingereicht im Rahmen der Prüfung in Moderne Browserkommunikation

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Martin Becke

Eingereicht am: 07. März 2017

Inhaltsverzeichnis

1	Architektur	2
1.1	Peer-to-Peer	2
1.2	Server-Client	3
1.3	UML-Diagramme	4
1.4	Schnittstellen und Nachrichtentypen	5
2	Storyboard	8
3	Technologieanalyse für serverbasierte Echtzeitkommunikation	11

Einleitung

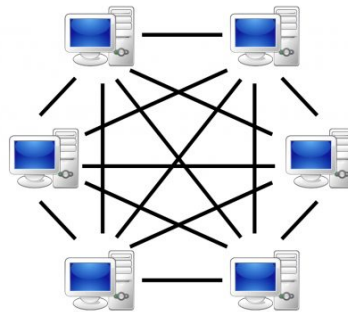
Riddle-Diddle ist ein schnellebige Mehrspieler Reaktionsspiel. Gedacht ist, dass der Server auf einem Gerät mit einem großen angeschlossenen Anzeigegerät läuft, auf welchem dann im Spiel die Anweisungen an die Spieler gut sichtbar angezeigt werden. Die Spieler können sich vor Spielbeginn per QR-Code mit ihrem Smartphone mit dem Server verbinden. Während des Spieles werden auf dem Anzeigegerät Aktionen für die Spieler angezeigt, welche sie mit dem Smartphone ausführen müssen, z.B. 'Drücke einen Button'. Das Smartphone ist als Spielsteuerung unerlässlich, da im Spiel z.B. der Gyrosensor als Eingabe ausgelesen wird. In einer späteren Iteration kann der Server auch genutzt werden um Spielstatistiken zu führen und zu speichern, um z.B. Highscores zu präsentieren. Nach beendetem Spiel wird der Server initialisiert und eine neue Runde kann gespielt werden.

1 Architektur

In diesem Kapitel beleuchten wir die unterschiedlichen Architekturen, welche für dieses Projekt möglich gewesen wären mit ihren Stärken und Schwächen.

1.1 Peer-to-Peer

Das wesentliche Merkmal bei Peer-to-Peer Netzwerken ist, dass es keinen dedizierten Server gibt, sodass auf jedem Peer eine komplette Instanz der Software laufen muss und alle Peers sich unter einander synchronisieren müssen. In der Planung dieses Projektes war, aufgrund der Realtime Implementation in z.B. WebRTC, Peer-to-Peer als eigentliche Architektur angedacht. Grund hierfür war der Gedanke, dass es nötig wäre für ein flüssiges Spielerlebnis so nahe wie möglich an Echtzeit-Kommunikation heranzukommen, was ggf. mit WebSockets nicht gewährleistet wäre.



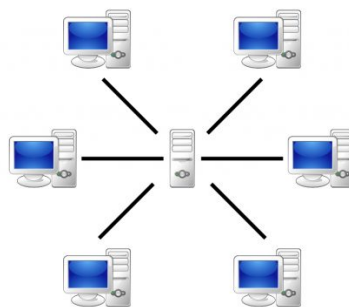
Ein simples Peer-to-Peer Netz

Entgegen der Befürchtung dass WebSockets zu viel Latenz einführen und ein flüssiges Spielen nicht möglich ist haben wir uns gegen eine Peer-to-Peer Architektur entschieden, da das Konzept unseres Spieles eine zentrale Spielführer Rolle propagiert und ein Prototyp uns gezeigt hat, dass die Antwortzeiten einer WebSocket Kommunikation im LAN absolut ausreichend sind. Da wir davon ausgehen, dass unser Spiel in einem WLAN gespielt wird, sollte eine ausreichend schnelle Kommunikation gewährleistet sein. Außerdem spricht gegen

eine Peer-to-Peer Architektur zum einen, dass alle Peers den Code des Spielführer ausgeliefert bekämen und der hauptsächliche Vorteil einer Peer-to-Peer Architektur, die Ausfallsicherheit, irrelevant wird.

1.2 Server-Client

Die Architektur der Wahl ist eine Server-Client Architektur. Hierbei stellte sich nur noch die Frage, in wie weit die Clients intelligent an der Ausführung des Spieles beteiligt sind oder einfach nur dumme Anzeige- bzw. Eingabegeräte sind.



Ein simples Server-Client Netz

Thick-Clients

In diesem Kontext meint Thick-Client, dass der Client Teile der Geschäftslogik beherbergt und somit Teile auf dem Server und Teile auf dem Client ausgeführt werden.

Ein Problem, welches mit dieser Architektur auftreten kann ist, dass es sehr schwierig ist die Ausführung des Client-Teil des Spieles zu koordinieren. Wird in einer Spielrunde vom Server aus an alle Clients die nächste auszuführende Aktion mit der Zeit, in der die Aktion ausgeführt werden soll, kann schwierig garantiert werden, dass an allen Clients diese zur selben Zeit aufhört. Schickt man eine Enduhrzeit mit, ist man darauf angewiesen, dass alle Uhren der Clients gleich gestellt sind. Schickt man eine Dauer in Millisekunden mit, in welcher die Aktion ausgeführt werden soll, ist man darauf angewiesen, dass alle Clients die Nachricht zur selben Zeit erhalten (Übertragungsdauer). Dies kann man einfach umgehen, indem man die Clients nur zur Anzeige und Eingabe nutzt.

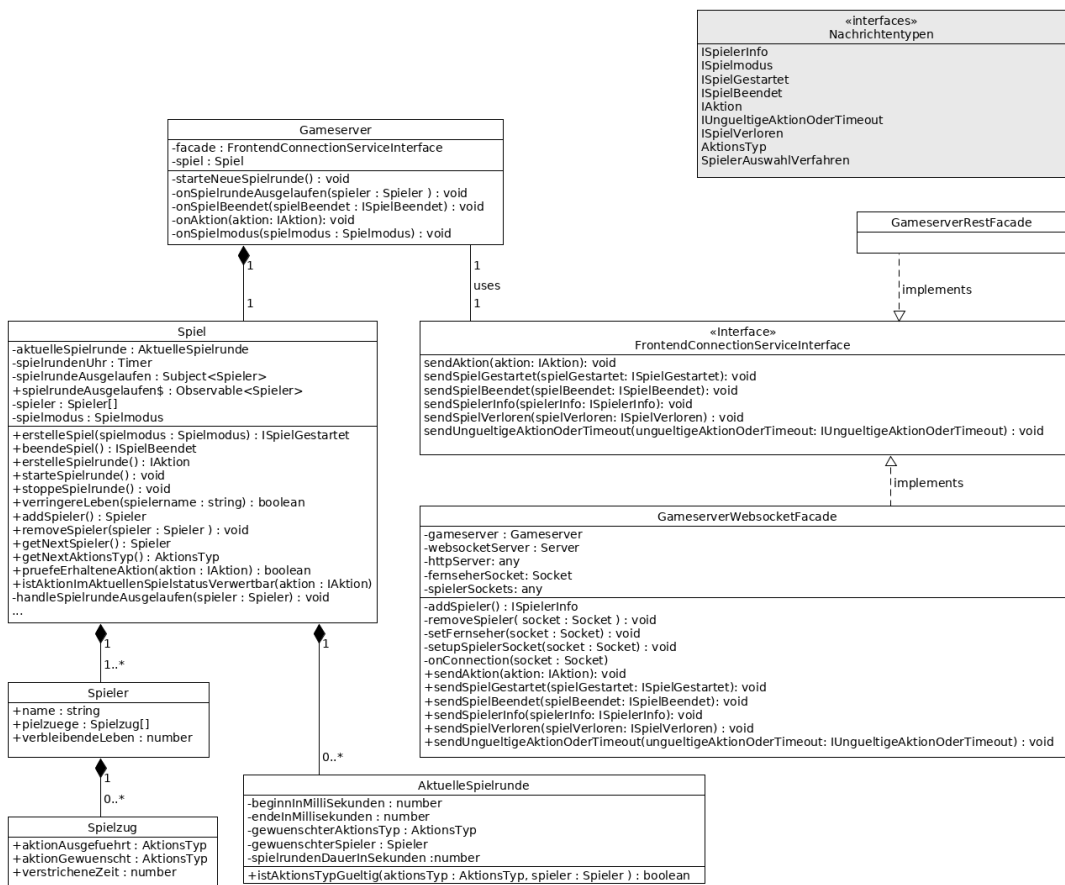
Thin-Clients

In diesem Kontext meint *Thin-Client*, dass der Client keine Teile der Geschäftslogik beherbergt und somit alle Teile auf dem Server ausgeführt werden und der Client letztendlich nur zur Anzeige dient.

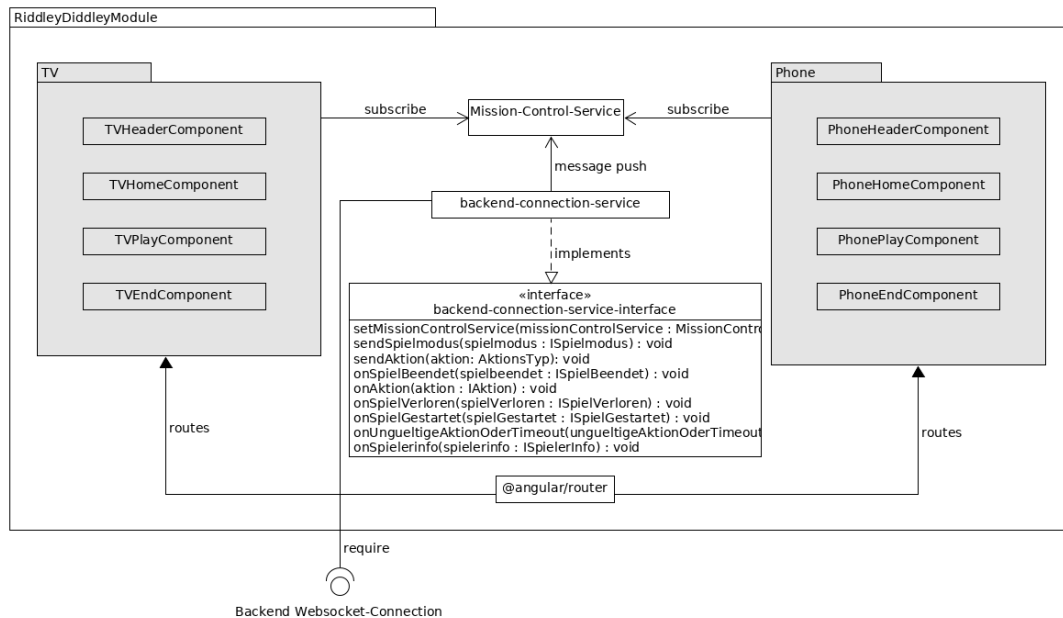
Ein guter Grund für die Wahl einer Server-Client Architektur mit Thin Clients ist die Tatsache, dass für ein Reaktionsspiel die Synchronizität der Zeit auf allen Geräten notwendig ist. Gibt man die Zeit nur vom Server vor und verteilt an alle Clients nur Daten zur Anzeige, ist keine Synchronisation notwendig. Die einzige Diskrepanz in der Zeit entsteht dann nur noch durch die Übertragung der Daten über das WLAN, was aber in allen Fällen anfallen würde.

1.3 UML-Diagramme

Backend



Frontend



1.4 Schnittstellen und Nachrichtentypen

Für die Kommunikation zwischen dem Gameserver, dem Fernseher-Client und den Smartphone-Clients nutzen wir die Websocket Schnittstelle. Die Nachrichten die auf dieser Schnittstelle ausgetauscht werden, sind wie folgt definiert:

- **SpielerInfo**

string : username

Beschreibung:

Information über den Spieler. Wird vom Gameserver nach dem Verbinden an den jeweiligen Spieler gesendet.

username : Der vom Gameserver dem Spieler zugewiesene eindeutige Spielername

- **Spielmodus**

number : zeitFuerAktion

number : auswahlVerfahrenSpieler

number : anzahlLeben

Beschreibung:

Definiert die Spielregeln, die für das Spiel gelten sollen. Der Smartphone-Client, der das Spiel startet, sendet diese Nachricht an den Gameserver, um die Regeln festzulegen.

zeitFuerAktion : Die Zeit, die jeder Spieler hat, um eine Aktion auszuführen in Sekunden von inklusive 1 bis inklusive 4

auswahlVerfahrenSpieler : Intern repräsentiert durch einen Enum. 1 bedeutet Spieler werden in Reihenfolge des beitreten zu Aktionen aufgefordert. 2 bedeutet Spieler werden zufällig zu Aktionen aufgefordert

anzahlLeben : Die Leben, die jeder Spieler hat (Bei falscher oder zu später Aktion wird ein Leben abgezogen) von inklusive 1 bis inklusive 4

- **SpielGestartet**

Spielmodus : spielmodus

string[] : beteiligteSpieler

Beschreibung:

Wenn das Spiel erstellt und gestartet wurde sendet der Gameserver diese Nachricht an alle Clients.

spielmodus : Der Spielmodus, welcher vom Ersteller des Spieles definiert wurde (*siehe Spielmodus*)

beteiligteSpieler : Eine Auflistung der Namen aller beteiligten Spieler

- **SpielBeendet**

Beschreibung:

Wird vom Gameserver an alle Clients geschickt, wenn das Spiel beendet ist.

- **Aktion**

string : spieler

number : typ

Beschreibung:

Der Gameserver schickt diese Aktion an den Fernseher-Client damit dieser die Aufforderung für alle Mitspieler anzeigen kann. Die Smartphone-Clients schicken diese Nachricht an den Gameserver, wenn sie eine Aktion ausgeführt haben.

spieler : Enthält wenn an den Fernseher-Client geschickt, den Spielernamen des Spielers der die Aktion ausführen soll. Wenn von Smartphone-Client geschickt, enthält den Namen des Spielers der die Aktion ausgeführt hat, um falsche Eingaben zu identifizieren

typ : Intern repräsentiert durch einen Enum. 1 bedeutet linker Button. 2 bedeutet rechter Button. 3 bedeutet Schütteln. 4 bedeutet unterer Button

- **UnguelteAktionOderTimeout**

string : spieler

Beschreibung:

Wird vom Gameserver an den Fernseher-Client geschickt, wenn ein Spieler eine falsche oder verspätete Eingabe gemacht hat.

spieler : Der Spielername des Spielers, welcher die falsche oder verspätete Eingabe gemacht hat

- **SpielVerloren**

string : spieler

Beschreibung:

Wird vom Gameserver an den Fernseher-Client geschickt, wenn ein Spieler keine Leben mehr hat und aus dem Spiel ausgeschieden ist.

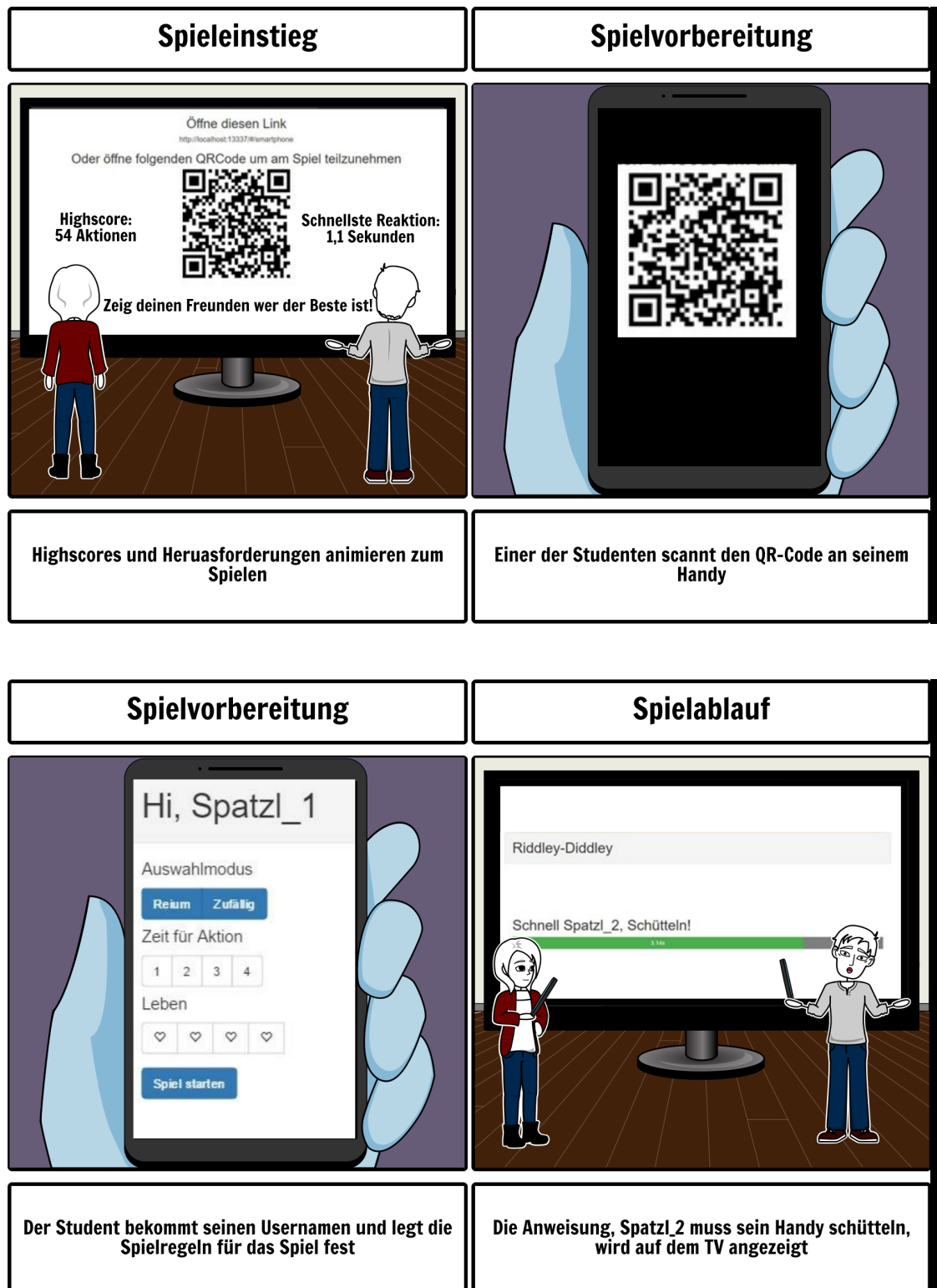
spieler : Der Spielername des Spielers, der aus dem Spiel ausgeschieden ist.

2 Storyboard

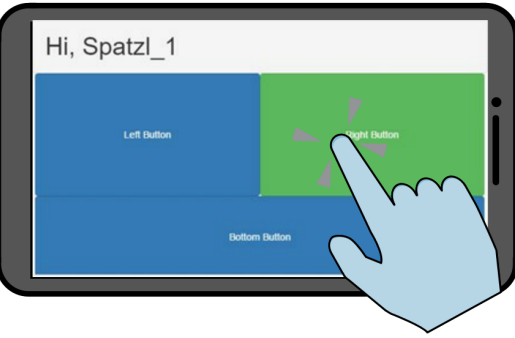

Gedachtes Einsatzszenario

Eine öffentlich zugängliche große Anzeige präsentiert dauerhaft den Startscreen, sodass man sich einfach via Smartphone mit dem Spiel verbinden kann.





Spielablauf	Spielablauf
	
<p>Spatzl_2 schüttelt sein Handy</p>	<p>Die Anweisung, Spatzl_1 muss den unteren Button drücken, wird auf dem TV angezeigt</p>

Spielablauf	Spielende
	
<p>Spatzl_1 drückt auf ihrem Handy den rechten Button</p>	<p>Spatzl_1 hat eine falsche Eingabe gemacht und das Spiel leider verloren</p>

3 Technologieanalyse für serverbasierte Echtzeitkommunikation

Am Beispiel einer funktionalen Anwendung: REST versus WebSocket

Ausgehend von dem in der Einleitung beschriebenem Spielkonzept ist eine bidirektionale Kommunikation zwischen dem Gameserver und den angeschlossenen Clients (Spieler und Fernseher) notwendig. Beide Seiten sollten in der Lage sein die Kommunikation zu initiieren, da beispielsweise für die Signalisierung des Spielstarts der Gameserver, für das Übermitteln der Eingaben jedoch die Spieler verantwortlich sind. Für die Umsetzung dieser Anforderungen werden die beiden etablierten Web-Technologien Websockets und REST/HTTP 1.1 miteinander verglichen.

Gemeinsamkeiten und Unterschiede

Sowohl Websockets als auch HTTP sind in der Lage die als JSON definierten Textnachrichten (siehe 1.4 Nachrichtentypen) zu transportieren. Es ist möglich mittels Websockets beliebige Binärdaten zu versenden. Für das Spiel ist jedoch nur Text relevant und Implementation wie Socket.io vereinfachen den Versand von Textnachrichten. Beide Protokolle setzen auf TCP auf und nutzen den Standard Port 80 (bzw. 443 für verschlüsselte TLS Verbindungen) was den Einsatz in vielen Netzwerken ermöglicht, da dieser in der Regel von Firewalls nicht blockiert wird.

Websockets verwenden eine persistente TCP-Verbindung und ermöglichen eine Full-Duplex Kommunikation zwischen den Kommunikationspartnern. Im Gegensatz dazu kann bei REST nur der Client Anfragen an den Server stellen, der Server selbst kann den Verbindungsaufbau nicht initialisieren, was für einige Nachrichtentypen wie SpielGestartet jedoch notwendig ist.

Um diese Anforderung trotzdem umsetzen zu können, können Mechanismen wie Polling verwendet werden. Dabei stellt der Client in regelmäßigen kurzen Abständen Anfragen an der Server um festzustellen, ob der Server Informationen hat, die er an den Client übertragen

möchte. Dies kann unter Umständen sehr ineffizient sein wenn der Server nur selten Daten zum Spieler bzw. Fernseher versenden möchte. Daher wirkt sich diese Technik negativ auf die Latenz aus, da der Server die Information nicht direkt "pushen" kann, sondern diese erst angefragt werden müssen. Dieser Ansatz kann allerdings durch Long Polling deutlich verbessert werden. Dabei blockiert der Server solange, bis er Daten an den Client ausliefern kann. Dadurch entfallen die u. U. vielen ergebnislosen Anfragen. Der Client muss bei dieser Technik allerdings mit den ggf. langen Antwort-Zeiten umgehen können und darf die Verbindung nicht frühzeitig durch einen Timeout beenden. In diesem Fall ist es sinnvoll, dass der Client direkt nach dem Erhalt der Antwort erneut eine Anfrage an den Server stellt und nicht wie bei Polling zunächst einige Zeit wartet. Des weiteren könnte sich der Auf- und Abbau der TCP-Verbindungen negativ auf die Performance auswirken. Beispielsweise wird in der Default-Konfiguration des Webserver Apache 2.4 eine Verbindung nur 5s gehalten¹ und nachkommende Anfragen erfordern einen erneuten Verbindungsaufbau.

Diese Eigenschaften von REST/HTTP machen einige Nachteile für eine Echtzeit-Anwendung deutlich und zeigen die Stärken von Websockets.

Bei der Implementation beider Techniken haben sich diese Nachteile bestätigt, da für die REST-Komponente im Server zusätzlich ein Long-Polling-Mechanismus programmiert werden musste. Der Aufwand für die WebSocket-Implementation war geringer und technisch weniger komplex.

¹<https://httpd.apache.org/docs/2.2/mod/core.html#keepalivetimeout>