

ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Resource-centric push model over WebSockets

Relatore:
Chiar.mo Prof.
Fabio Vitali

Presentata da:
Filippo Vigani

Sessione I
Anno Accademico 2018-2019

Indice

Lista delle figure	iii
Introduzione	iv
1 Background	1
1.1 Sviluppo app moderne real-time	1
1.2 Client pull	2
1.2.1 Polling e long polling	2
1.2.2 Problematiche	5
1.3 Server push	6
1.3.1 Server Sent Events	6
1.3.2 WebSockets	7
2 Listen JS	10
2.1 Struttura ad alto livello	10
2.1.1 Libreria client	11
2.1.2 Libreria server	12
2.2 Features	14
2.2.1 Disconnection detection	14
2.2.2 Reconnection	14
2.2.3 Multiplexing	14
2.2.4 Observer pattern across the stack	14
2.2.5 Semplificazione utilizzo WebSocket	14
2.3 Struttura a basso livello	14
2.3.1 Architettura	14
2.3.2 Implementazione multiplexing	14
2.3.3 Implementazione disconnection detection	14
2.3.4 Implementazione reconnection	14
2.3.5 Implementazione observer pattern	14
2.4 Applicazione di esempio	14

3	Valutazione	15
3.1	Confronto tra XHR polling e websockets	15
3.1.1	Responsiveness	15
3.1.2	Traffico dati	15
3.1.3	User experience	15
3.2	Valore aggiunto di Listen JS	15
3.2.1	Utilizzo risorse e limitazioni browser	15
3.2.2	Immediatezza integrazione da parte di sviluppatori	15
3.2.3	Architettura	15
	Conclusioni	16
4	Sviluppi futuri	17
	Bibliografia	18

Elenco delle figure

1.1	Polling	3
1.2	Long polling	4
1.3	Server Sent Events Flow	7
1.4	WebSockets Flow	9

Introduzione

Quando si sviluppa un'applicazione web, si deve considerare che meccanismo di data delivery utilizzare. Spesso si vuole sviluppare un'applicazione che lavori con dati in tempo reale: può essere una dashboard con l'andamento di un mercato azionario, o una console di un servizio backend su cui lavorano più utenti, o ancora un semplice calendario condiviso per la gestione di appuntamenti.

Per molto tempo l'unica modalità per reperire i dati da un web browser è stata tramite client pull, ovvero il client si occupa di richiedere una risorsa, e richiedere se la risorsa stessa sia stata modificata.

Con l'implementazione dei websocket nella maggior parte dei browser, la situazione cambia. Si apre la possibilità di ricevere dati tramite server push, senza dover periodicamente richiedere una risorsa, ma lasciando al server l'onere di notificare i client dell'avvenuta modifica della risorsa.

Tuttavia i websocket rimangono un'implementazione di basso livello, e lavorarci su applicazioni di alto livello, dove l'architettura e la separation of concerns è un punto focale, risulta complesso.

La soluzione proposta permette in modo semplice ed intuitivo di rimanere in ascolto di una risorsa come se fosse un endpoint REST, e ogni qualvolta questa risorsa venga aggiornata, essere notificati del nuovo contenuto, senza doversi preoccupare di una gestione efficiente delle risorse.

Capitolo 1

Background

Nello sviluppo di applicazioni web moderne spesso si ha la necessità di aggiornare parti dell'interfaccia in modo che rispecchino delle risorse non presenti localmente in tempo reale. Si pensi per esempio ad un'applicazione che deve visualizzare i dati dello stock market, o ad un social network, o ad un calendario per la gestione degli appuntamenti, o ancora ad una semplice dashboard di gestione aziendale.

In tutti questi casi, poter vedere le modifiche effettuate da terzi sulle stesse risorse in real-time è essenziale o migliora di gran lunga la user experience. Le applicazioni real-time stanno gradualmente dominando l'internet in quanto forniscono un perfetto equilibrio di informazioni, funzionalità, contenuto e interattività che portano ad aumentare lo user engagement.

1.1 Sviluppo app moderne real-time

Con applicazioni real-time intendiamo applicazioni che permettano di ricevere e visualizzare degli aggiornamenti che risiedono su un server nel minor tempo possibile. Da una definizione così banale, spuntano in realtà una serie di questioni importanti per quanto concerne sia le tecnologie per implementarle, che le tecniche utilizzate che la gestione delle risorse.

Gli sviluppatori web, fino a qualche anno fa, per ottenere dei risultati simili hanno dovuto sfruttare diverse tecniche che aggirassero le limitazioni dei browser. Infatti l'unico modo di ricevere dati per un browser era quello di inviare una richiesta al server, e ricevere una risposta. Ciò significa che il client non aveva modo di essere notificato in caso la risorsa richiesta venisse modificata.

1.2 Client pull

Questo stile di comunicazione ove la richiesta è originata dal client e risposta dal server è chiamato *client pull*. Il client pull forma la base per la comunicazione tra un browser e un server tramite il protocollo HTTP, e su di esso si sono costruiti una serie di stili architetturali per fornire interoperabilità tra web services in maniera prestabilita. Per esempio, uno degli stili più utilizzati è REST (REpresentational State Transfer), che stabilisce che i web services devono permettere ad altri sistemi di richiedere l'accesso o la manipolazione di rappresentazioni testuali di risorse web usando un insieme predefinito di operazioni stateless.

Un protocollo stateless prevede che il server non mantenga nessuna informazione riguardante la connessione attiva del client tra una richiesta e l'altra. Non mantenendo alcun tipo di informazione sulla sessione, ne consegue che per verificare se una risorsa è stata aggiornata rispetto a quella salvata da un browser in precedenza, è necessario richiedere la stessa risorsa e compararla con la precedente.

1.2.1 Polling e long polling

Una delle tecniche largamente utilizzate per verificare se una risorsa, web o meno, sia stata modificata è appunto richiedere la stessa risorsa a intervalli regolari, e confrontarla con la precedente. Questa tecnica nella letteratura è chiamata polling. Nell'ambito dei web services e di HTTP, si possono distinguere due tipi di polling: Polling semplice e long polling

Nel **polling semplice** la risorsa viene richiesta periodicamente e il server risponde immediatamente con la risorsa richiesta. Ad un livello più basso, viene aperta una connessione, inviato un messaggio dal client al server contenente la richiesta, inviato un messaggio di risposta dal server al client contenente la risorsa, e la connessione viene chiusa. Successivamente, dopo un periodo di polling prestabilito, si ripete. Un esempio di utilizzo di polling client side potrebbe essere il seguente:

```
1  const POLL_RATE = 5000
2
3  /* Start polling */
4  setTimeout(() => {
5      fetchAppointmentsAndUpdateUI()
6  }, POLL_RATE)
7
8  function fetchAppointmentsAndUpdateUI() {
9      fetch('/api/appointments')
10     .then(response => {
11         /* Parse response */
12         return response.json()
```

```
13     })
14     .then(appointments => {
15         /* Update UI */
16         this.setState({
17             appointments: appointments
18         })
19     })
20     .catch(error => {
21         /* Handle error */
22         this.setState({
23             error: error
24         })
25     })
26 }
```

Listing 1.1: Client side XHR polling example



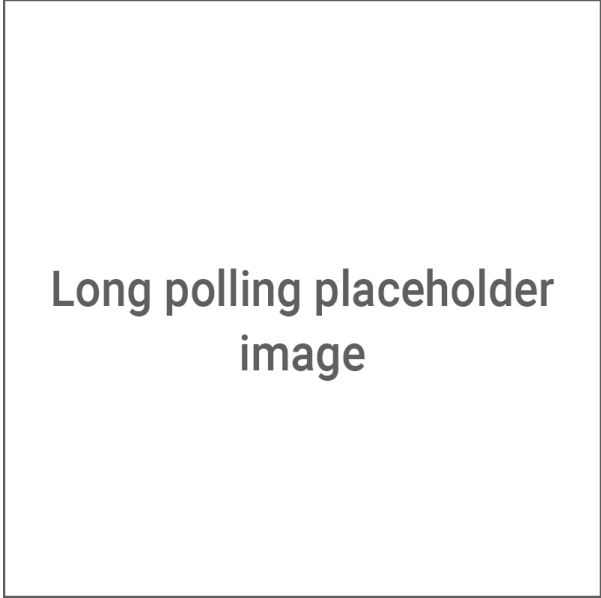
Polling placeholder image

Figura 1.1: Polling

Nel caso del **long polling**, conosciuto anche come *hanging GET* o *COMET*, invece, il client invia una richiesta ad una risorsa specifica. Il server, invece di rispondere immediatamente, tiene aperta la connessione fintanto che la risorsa non viene aggiornata o finché una soglia di timeout viene superata. Quando sono presenti nuovi dati, il server risponde alla richiesta chiudendo la connessione di conseguenza. Dopodiché il client richiede nuovamente la stessa risorsa e rimane in attesa. Questo flusso può essere implementato come l'esempio seguente:

```
1  /* Start long polling */
2  fetchAppointmentsAndUpdateUI()
3
4  function fetchAppointmentsAndUpdateUI() {
5    fetch('/api/appointments')
6    .then( response => {
7      return response.json()
8    })
9    .then( appointments => {
10     /* Update UI */
11     this.setState({
12       appointments: appointments
13     })
14     /* Restart long polling */
15     fetchAppointmentsAndUpdateUI()
16   })
17   .catch(error => {
18     /* Handle error */
19     this.setState({
20       error: error
21     })
22   })
23 }
```

Listing 1.2: Client side XHR long polling example



Long polling placeholder
image

Figura 1.2: Long polling

A differenza del polling semplice, dove l'implementazione riguarda solo la parte client, e il server espone un semplice endpoint REST, nel long polling il server deve supportare questo tipo di interazione con il client. Infatti, oltre ad esporre un semplice endpoint REST, dovrà esporre un endpoint che si occupa di mantenere il client in attesa, ed inviare la risposta solo una volta che viene aggiornata la risorsa. Inoltre, dovrà gestire lo stato di connessioni multiple, ed implementare strategie per preservare lo stato delle sessioni quando si utilizzano più server e load balancers.

1.2.2 Problematiche

Entrambe le tecniche di polling sono dei workaround dovute alle storiche limitazioni dei browser e di HTTP, e dunque presentano delle problematiche.

Consumo di banda e traffico dati

Poiché il polling richiede ad intervalli regolari la stessa risorsa, spreca traffico dati per passare sia la richiesta della risorsa stessa, che per ricevere il payload effettivo. In particolare per ogni richiesta HTTP, deve essere stabilita una nuova connessione, si deve fare il parsing degli header HTTP, si deve presumibilmente reperire i dati da un database, e infine inviare i dati al client.

Ritardo

Per limitare questo consumo, si cerca di trovare un equilibrio riducendo la frequenza di poll. Ma così facendo, risulta che per ricevere un aggiornamento di una risorsa, il client potrebbe aspettare fino alla durata dell'intervallo di tempo tra una richiesta e la successiva.

Incoerenza dei dati

Nel caso del long polling, la risorsa viene inviata presumibilmente appena subisce delle modifiche. Tuttavia nel lasso di tempo che passa tra quando il client riceve una risposta, ed effettua la nuova richiesta da tenere aperta, la risorsa stessa potrebbe essere modificata. In tal caso, il server non ha nessuna connessione in sospenso con il client, ed esso perderebbe l'aggiornamento.

Performance e scalabilità

Con l'aumentare del numero di client connessi, il numero di richieste rapportate al tempo invece di incrementare linearmente si moltiplica, in quanto ogni client per la stessa risorsa non effettuerà una singola richiesta, ma richieste multiple. Ciò rende i sistemi che implementano supportano polling difficili da scalare e poco performanti.

1.3 Server push

A differenza del client pull, il server push è uno stile di comunicazione che prevede che sia il server a inviare ad un client dei dati, senza che una richiesta venga inviata in precedenza da parte del client. I servizi che supportano server push, spesso si basano su delle preferenze espresse dal client in precedenza. Questo modello è chiamato *publish/subscribe*. Un client esprime di voler ricevere gli aggiornamenti ad una risorsa o “channel”, e il server si occupa di inviare la risorsa a tale channel ogni volta che viene modificata.

Prima di HTML5, non era possibile implementare un modello *publish/subscribe* che funzionasse sulle applicazioni web, se non tramite astrazioni basate su *polling*. Con l'arrivo di HTML living standard e diverse tecnologie web, è ora possibile implementare sistemi *connection-based* per lo scambio di dati in *real-time*.

1.3.1 Server Sent Events

I Server Sent Events (SSE) sono una tecnologia push che permette ad un browser di ricevere aggiornamenti da un server tramite una connessione HTTP tramite una comunicazione *simplex*¹. L'API che offre questa funzionalità, chiamata *EventSource API*, è standardizzata dal W3C come parte di HTML5.

Il flusso di una comunicazione basata su *EventSource* è il seguente[1], sintetizzato in figura 1.3.

1. Il client tramite la chiamata API `new EventSource('/api/myEndpoint')` apre una nuova connessione HTTP.
2. Il client registra le callback agli eventi `onmessage`, `onopen` e `onerror`.
3. Il server risponde alla prima richiesta specificando nell'header `Content-Type` il tipo `text/event-stream`.
4. Il client, se supporta la *EventSource API*, mantiene la connessione aperta in attesa di nuovi messaggi.
5. Il server può dunque inviare quando desidera un nuovo messaggio, finché la connessione non viene chiusa da una delle due parti.

Un esempio di utilizzo di quest'API è il seguente:

```
1 /* Open new EventSource connection */
2 const source = new EventSource('/api/appointments')
3
```

¹in un'unica direzione

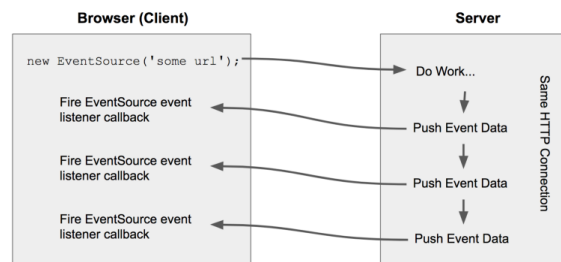


Figura 1.3: Server Sent Events Flow

```

4 source.onmessage = event => {
5   /* Parse response */
6   const appointments = JSON.parse(event.data)
7   /* Update UI */
8   this.setState({
9     appointments: appointments
10  })
11 }
12
13 connection.onerror = error => {
14   /* Handle error */
15   this.setState({
16     error: error
17   })
18 }

```

Listing 1.3: Client side EventSource example

La limitazione di questa tecnologia è che una volta aperta una connessione, il client non potrà sfruttare la stessa per inviare ulteriori messaggi. Ciò significa che per rimanere in ascolto di diverse risorse in istanti differenti, sarà necessario aprire più connessioni. Se invece il server supporta HTTP/2, SSE potrà sfruttare il multiplexing offerto da HTTP/2 automaticamente. Inoltre, al momento di questa stesura, nessuna versione di Internet Explorer e nessuna versione precedente alla 75 di Edge supportano i SSE[2].

1.3.2 WebSockets

I WebSocket sono un protocollo che permette una comunicazione full-duplex² mediante una singola connessione TCP. L’RFC 6455 afferma che WebSocket è progettato in modo da funzionare attraverso le porte HTTP 80 e 433 e di supportare proxy e intermediari HTTP, rendendolo dunque compatibile con il protocollo HTTP[3]. Per poter essere compatibile, l’handshake tramite WebSocket fa uso dell’header `upgrade` di HTTP per

²in entrambe le direzioni e contemporaneamente

cambiare protocollo da HTTP a WebSocket. Ciò è possibile perché entrambi i protocolli fanno parte dell'application layer nel modello OSI e dipendono da TCP.

Nei browser i WebSocket possono essere utilizzati mediante la *WebSocket API*, anch'essa presente nell'HTML Living Standard di WHATWG. Il flusso di comunicazione basato su WebSocket può essere espresso come segue[4] ed illustrato in figura 1.4:

1. Il client tramite la chiamata API `new WebSocket('/api/myEndpoint')` apre una nuova connessione HTTP, specificando nella richiesta gli header `Upgrade: WebSocket` e `Connection: Upgrade`.
2. Il client registra le callback agli eventi `onmessage`, `onopen`, `onclose` e `onerror`.
3. Il server risponde alla richiesta con codice 101 `Switching Protocols` e gestisce il socket aperto tramite protocollo WebSocket.
4. Il client e il server possono finalmente scambiarsi messaggi in entrambe le direzioni.

L'implementazione è molto simile a quella dei Server Sent Events:

```
1  /* Initiate new WebSocket handshake */
2  const socket = new WebSocket(`ws://${location.host}/api/appointments`)
3
4  connection.onmessage = message => {
5    /* Parse response */
6    const appointments = JSON.parse(message.data)
7    /* Update UI */
8    this.setState({
9      appointments: appointments
10   })
11 }
12
13 connection.onerror = error => {
14   /* Handle error */
15   this.setState({
16     error: error
17   })
18 }
```

Listing 1.4: Client side WebSocket example

I WebSocket sono supportati da tutti i browser moderni. Sfortunatamente al momento della stesura non è possibile comunicare con i WebSocket tramite HTTP/2, in quanto non dispone del meccanismo di Upgrade di HTTP/1.1. Tuttavia i browser si stanno adoperando per supportare l'apertura di una comunicazione attraverso WebSocket su HTTP/2 seguendo il recente RCF 8441[5]. Ciò porterà a poter sfruttare il multiplexing offerto da HTTP/2 automaticamente anche con i WebSocket.

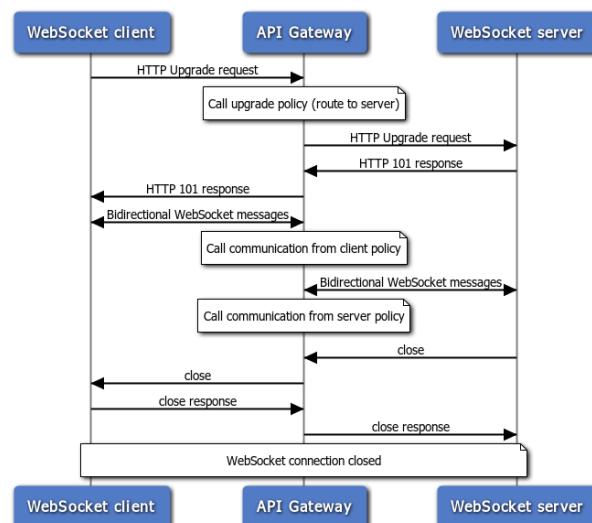


Figura 1.4: WebSockets Flow

Capitolo 2

Listen JS

Il mio contributo nell'ambito dello sviluppo di web application real-time si concretizza in Listen JS, un insieme di librerie che permettono di osservare in tempo reale gli aggiornamenti ad una o più risorse. Il progetto comprende anche un'applicazione di esempio che permette a più utenti di collaborare sull'inserimento, la modifica e la cancellazione di appuntamenti su un calendario tramite una web app.

Le librerie messe a disposizione sono rispettivamente `listenjs` e `listenjs-server`, la prima è una libreria client, utilizzabile su qualsiasi browser che supporti i WebSocket, la seconda è una libreria server utilizzabile su server sviluppati in NodeJS. Entrambe le librerie sono volutamente framework-agnostic, cioè possono essere facilmente integrate in qualsiasi framework già esistente, sia front-end (e.g. Angular, React, VueJS...) che back-end (e.g. Express, Fastify, Koa...), e non sono opinionate rispetto ad alcun paradigma di programmazione. Utilizzando costrutti già presenti in Javascript, e non avendo dipendenze ad altre librerie, possono essere integrate in una codebase già esistente apportando pochissime modifiche.

Listen JS è volta a diminuire l'inerzia dell'implementazione di un'app real-time basata su WebSocket facilitandone l'utilizzo e riducendo il codice boilerplate. Inoltre fornisce una serie di funzionalità descritte nella sezione 2.2 per migliorare la gestione delle risorse, l'affidabilità e la semplicità d'utilizzo.

2.1 Struttura ad alto livello

Le librerie utilizzano i WebSocket come layer di trasporto e astraggono la comunicazione scambiandosi dei messaggi predefiniti, ma che contengono payload definiti dall'utente. La scelta di utilizzare i WebSocket come trasporto invece dei Server Sent Events è dovuta ai seguenti motivi:

1. I WebSocket sono supportati da tutti i browser moderni, a differenza dei Server Sent Events che (al momento della stesura) non sono supportati da Internet Explorer e alcune versioni di Edge.
2. Con i Server Sent Events non è possibile fare multiplexing utilizzando HTTP/1.1, in quanto non è possibile inviare messaggi successivi da client a server utilizzando la stessa connessione.
3. Poiché con i Server Sent Events non si possono inviare messaggi da client a server sulla stessa connessione, risulta difficile implementare meccanismi di ping personalizzati. È dunque complicato lato client verificare quando un server non è più raggiungibile.

2.1.1 Libreria client

La libreria client permette con una linea di codice di rimanere in ascolto di qualsiasi risorsa negli endpoint che si desidera.

Installazione

La libreria client di ListenJS può essere aggiunta ad un progetto che sfrutti un module bundler (come per esempio WebPack) tramite il package manager NPM, eseguendo il comando:

```
npm install @filippovigani/listenjs
```

per poi essere importata o tramite CommonJS o gli import dei moduli di ES6:

```
const listen = require('@filippovigani/listenjs').listen
```

oppure

```
import { listen, ... } from "@filippovigani/listenjs/listen"
```

Alternativamente è possibile scaricare direttamente il codice sorgente[6] e importarlo nel progetto che si desidera.

Utilizzo

Per ogni risorsa per cui si vuole essere notificati degli aggiornamenti, basterà semplicemente chiamare la funzione per ascoltare uno specifico endpoint:

```
1  const URI = '/api/appointments'
2
3  listen(
4    URI,
5    appointments => {
6      /* Handle new data update */
7      this.setState({
8        appointments: appointments
9      })
10   }
11 )
```

Listing 2.1: Listen JS client example

L'URI passato come argomento della funzione, può sia essere relativo, e quindi utilizzare lo stesso dominio della pagina di origine, oppure specificare un indirizzo completo, per esempio `http://differentDomain.com/api/appointments`. Questo dominio per permettere al protocollo di WebSocket di fare l'handshake, dovrà avere abilitato i CORS (Cross-Origin Resource Sharing) in quanto la risorsa è su un'origine differente.

2.1.2 Libreria server

La libreria server permette di inizializzare un server che supporti i WebSocket e gestisce i client in ascolto sulle varie risorse, permettendo di notificare tutti i client in ascolto su una specifica risorsa.

Installazione

Per aggiungere la libreria server di ListenJS ad un progetto NodeJS è sufficiente eseguire il comando:

```
npm install @filippovigani/listenjs-server
```

e per poterla utilizzare basta importarla come tutti i moduli di NodeJS

```
const listen = require('@filippovigani/listenjs-server')
```

Come per la parte client, è in alternativa possibile scaricare il codice sorgente[7] e importarlo manualmente nel progetto.

Utilizzo

In fase di inizializzazione del server è necessario fare il binding di un server HTTP di NodeJS già esistente ListenJS, per poter inizializzare connessioni tramite WebSocket. Ciò si può fare chiamando la funzione `setup` come segue:

```
1  const http = require('http')
2
3  const server = http.createServer((request, response) => {
4    /* TODO: Handle response */
5  })
6
7  /* Setup ListenJS*/
8  listen.setup({server: server})
9
10 server.listen(8000)
```

Listing 2.2: Listen JS server setup example

In caso fossero utilizzati framework per il routing degli endpoint, è sempre possibile reperire l'istanza del server HTTP sottostante per poter inizializzare ListenJS. Per esempio, utilizzando Fastify, ListenJS verrà inizializzato in questo modo:

```
1  const fastify = require('fastify')
2  const listen = require('@filippovigani/listenjs-server')
3
4  const app = fastify({/* Fastify parameters */})
5
6  listen.setup({server: app.server})
```

Listing 2.3: Listen JS Fastify setup example

Per qualsiasi altro framework si può trovare nella documentazione come reperire il server HTTP.

Una volta inizializzato, è possibile notificare i client che sono in ascolto di una risorsa tramite la funzione `notify`. Per esempio, in un endpoint POST, il codice per notificare tutti i client in ascolto su una risorsa è il seguente:

```
1  app.post('/appointments', function (request, reply) {
2    const appointment = request.body
3    const appointments = controller.postAppointment(appointment)
4
5    listen.notify(request.urlData.path, appointments)
6
7    reply.send(200)
8  })
```

Listing 2.4: Listen JS server notify example

La libreria si occuperà di inviare un messaggio contenente il nuovo payload a tutti e soli i client in ascolto. Se non ci fosse nessun client in ascolto su quella specifica risorsa, la chiamata non avrà alcun effetto.

2.2 Features

Oltre a permettere...

2.2.1 Disconnection detection

2.2.2 Reconnection

2.2.3 Multiplexing

2.2.4 Observer pattern across the stack

2.2.5 Semplificazione utilizzo WebSocket

2.3 Struttura a basso livello

Il funzionamento del sistema si basa sullo scambio dei messaggi che hanno la seguente struttura:

2.3.1 Architettura

E messaggi di scambio

2.3.2 Implementazione multiplexing

2.3.3 Implementazione disconnection detection

ping/keep alive

2.3.4 Implementazione reconnection

backoff

2.3.5 Implementazione observer pattern

2.4 Applicazione di esempio

CORS

Capitolo 3

Valutazione

3.1 Confronto tra XHR polling e websockets

3.1.1 Responsiveness

3.1.2 Traffico dati

3.1.3 User experience

Tradeoff

3.2 Valore aggiunto di Listen JS

3.2.1 Utilizzo risorse e limitazioni browser

Limite socket aperti contemporaneamente

3.2.2 Immediatezza integrazione da parte di sviluppatori

3.2.3 Architettura

Separation of Concerns

Conclusioni

Capitolo 4

Sviluppi futuri

SSL

Bibliografia

- [1] WHATWG. Server-Sent Events. <https://html.spec.whatwg.org/multipage/server-sent-events.html#server-sent-events>.
- [2] Server Sent Events Support. <https://caniuse.com/#search=server%20sent%20events>.
- [3] I. Fette and A. Melnikov. The WebSocket Protocol. RFC 6455, RFC Editor, December 2011.
- [4] MDN. WebSockets. https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API.
- [5] P. McManus. Bootstrapping WebSockets with HTTP/2. RFC 8441, RFC Editor, September 2018.
- [6] ListenJS client source code. <https://github.com/VeegaP/listenjs>.
- [7] ListenJS server source code. <https://github.com/VeegaP/listenjs-server>.