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

Elenco delle figure

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       return response.json()
12    })
```

```
13     .then( appointments => {
14        /* Update UI */
15        this.setState({
16           appointments: appointments
17        })
18     })
19  }
```

Listing 1.1: Client side XHR polling example

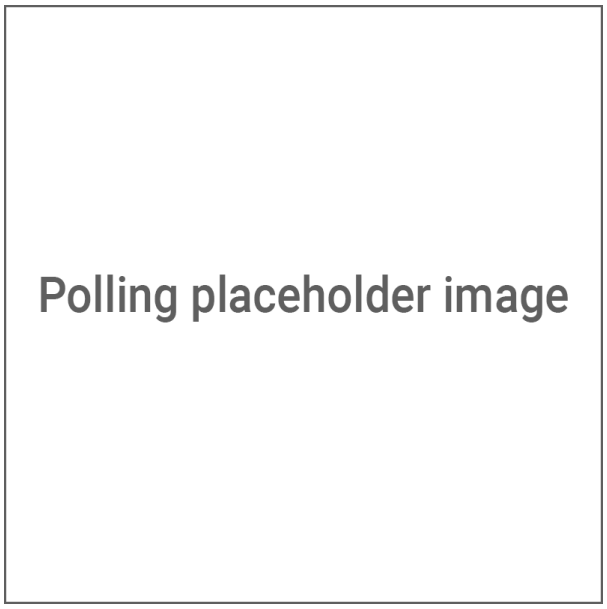

Figura 1.1: Polling

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.

```
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  }
```

Listing 1.2: Client side XHR long polling example

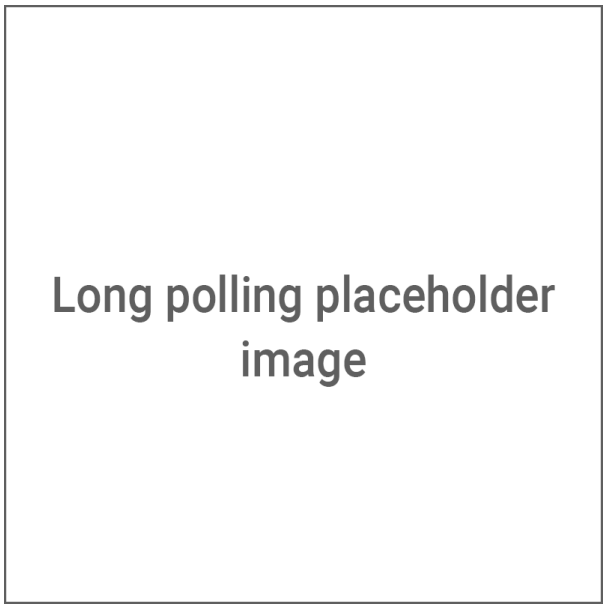

Figura 1.2: Long polling

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 sospeso 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 broswer di ricevere aggiornamenti da un server tramite una connessione HTTP. 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. 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.

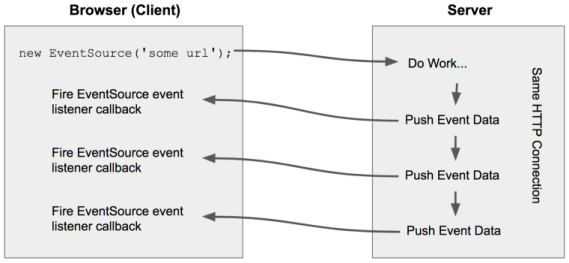

Figura 1.3: Server Sent Events Flow

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 instanti differenti, sarà necessario aprire più connessioni. Inoltre, al momento di questa stesura, nessuna versione di Internet Explorer e nessuna versione precedente alla 75 di Edge supportano i SSE.[1]

1.3.2 WebSockets

Capitolo 2

Listen JS

2.1 Struttura ad alto livello

2.1.1 Libreria client

2.1.2 Libreria server

2.2 Features

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

2.3.1 Architettura

E messaggi di scambio

2.3.2 Implementazione multiplexing

2.3.3 Implementazione disconnection detection

2.3.4 Implementazione reconnection

2.3.5 Implementazione observer pattern

2.4 Applicazione di esempio

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

Questo è un test di citazione [2]

Capitolo 4

Sviluppi futuri

Bibliografia

[1] Server Sent Events. https://caniuse.com/#search=server%20sent%20events.

[2] WebSockets - MDN. https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API.