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

# Real-time web applications

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 Tecniche di sviluppo

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:

```
const POLL_RATE = 5000

/* Start polling */
setTimeout(() => {
    fetchAppointmentsAndUpdateUI()
}, POLL_RATE)

function fetchAppointmentsAndUpdateUI(){
    fetch('/api/appointments')
    .then(response => {
        /* Parse response */
        return response.json()
    })
    .then(appointments => {
        /* Update UI */
        this.setState({
            appointments: appointments
        })
    })
    .catch(error => {
        /* Handle error */
        this.setState({
            error: error
        })
    })
}
```

Listing 1.1: Client side XHR polling example

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

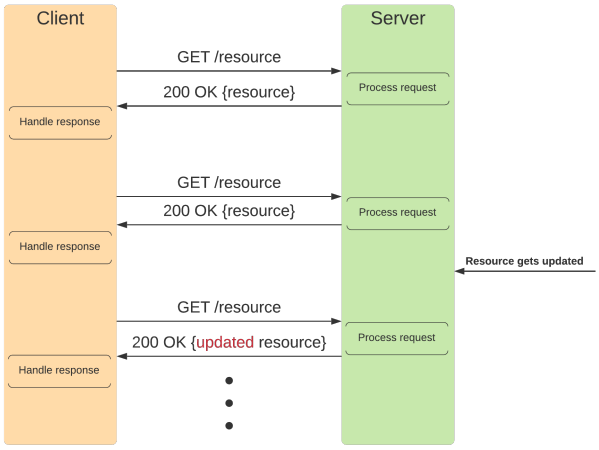


Figura 1.1: Polling

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

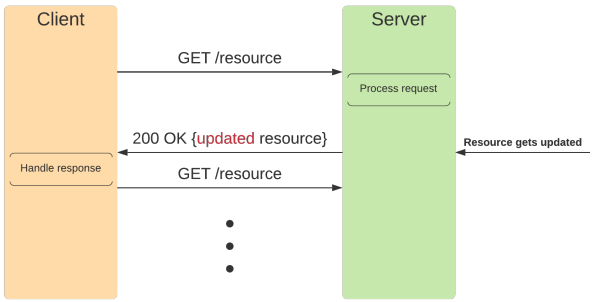


Figura 1.2: Long polling

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 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`.

[1] in un'unica direzione

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.

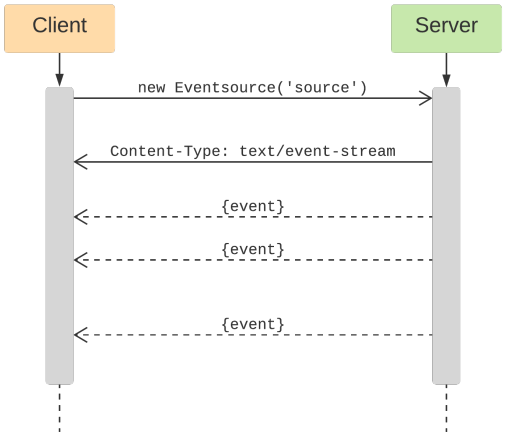


Figura 1.3: Server Sent Events Flow

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

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

source.onmessage = event => {
    /* Parse response */
    const appointments = JSON.parse(event.data)
    /* Update UI */
    this.setState({
        appointments: appointments
    })
}

connection.onerror = error => {
    /* Handle error */
    this.setState({
        error: error
    })
}
```

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 instanti 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-duplex[2] 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 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`.

[2] in entrambe le direzioni e contemporaneamente

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:

```
/* Initiate new WebSocket handshake */
const socket = new WebSocket(`ws://${location.host}/api/
    appointments`)

socket.onopen = event => {
    socket.send('Hello server!')
}

socket.onmessage = message => {
    /* Parse response */
    const appointments = JSON.parse(message.data)
    /* Update UI */
    this.setState({
        appointments: appointments
    })
}

socket.onerror = error => {
    /* Handle error */
    this.setState({
        error: error
    })
}
```

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.

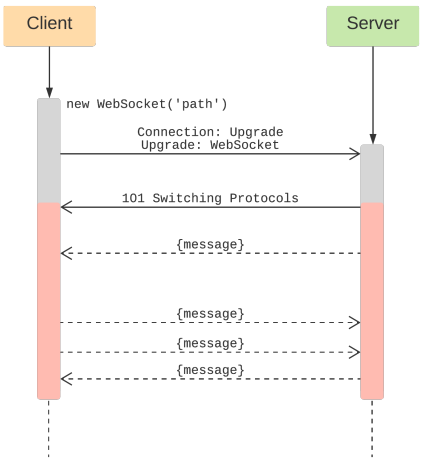


Figura 1.4: WebSockets Flow

# Capitolo 2

# ListenJS

Il mio contributo nell'ambito dello sviluppo di web application real-time si concretizza in ListenJS, 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.

ListenJS 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 2.3 per migliorare la gestione delle risorse, l'affidabilità e la semplicità d'utilizzo.

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.

Il client può mettersi in ascolto di una o più risorse in qualsiasi istante nel tempo, e ricevere aggiornamenti in tempo reale delle risorse osservate, come in figura 2.1.

## 2.1 Libreria client

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

### 2.1.1 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:

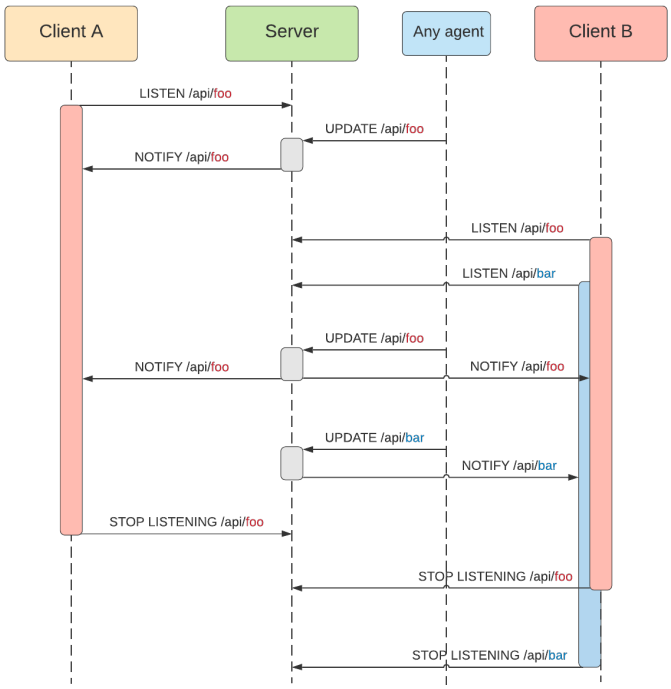


Figura 2.1: ListenJS client-server interaction

```
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.

### 2.1.2 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: ListenJS 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 `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.

## 2.2 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.

### 2.2.1 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.

### 2.2.2 Utilizzo

In fase di inizializzazione del server è necessario fare il binding di un server HTTP di NodeJS già esistente con ListenJS, per poter inizializzare con-

nessioni tramite WebSocket. Ciò si può fare chiamando la funzione `setup` come segue:

```
const http = require('http')

const server = http.createServer((request, response) => {
   /* TODO: Handle response */
})

/* Setup ListenJS */
listen.setup({server: server})

server.listen(8000)
```

Listing 2.2: ListenJS 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:

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

const app = fastify({/* Fastify parameters */})

listen.setup({server: app.server})
```

Listing 2.3: ListenJS 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:

```
app.post('/appointments', function (request, reply) {
   const appointment = request.body
   const appointments = controller.postAppointment(
      appointment)

```

```
5     listen.notify(request.urlData.path, appointments)
6
7     reply.send(200)
8 })
```

Listing 2.4: ListenJS server notify example

La libreria si occuperà di inviare un messaggio contentente 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.

## 2.3 Features

Oltre a semplificare la comunicazione dell'aggiornamento delle varie risorse in ascolto dai client, ListenJS implementa una serie di funzionalità per semplificare la vita allo sviluppatore, e permettere di adempire al Single Responsibility Principle[8] nell'applicazione real-time che utilizza questo sistema di comunicazione.

### 2.3.1 Disconnection detection

L'evento `onclose` fornito dai WebSocket viene lanciato solo nel caso in cui il socket venga chiuso esplicitamente o dal client o dal server tramite l'handshake di chiusura specificato nell'RFC 6455. Tuttavia, in caso di problemi di network, la chiusura del socket non verrà notificata, né dal client né dal server. In aggiunta, i messaggi scambiati in caso di pessime condizioni di network verranno lasciate in sospese.

Per ovviare a questo problema ListenJS implementa un sistema di heartbeat (aka ping-pong), che fa si che il client invii periodicamente dei messaggi di ping, aspettandosi un messaggio di pong dal server entro un certo timeout. Se non dovesse riceverlo, allora considererà il socket come chiuso, provando ad iniziare l'handshake di chiusura tramite la chiamata alla WebSocket API. Di contro, il server si aspetterà di ricevere un messaggio di ping ogni intervallo prestabilito. Se non dovesse succedere, allora si aspetterà di non avere più connettività con il client, chiudendo il socket.

### 2.3.2 Reconnection

I WebSocket non permettono di riaprire un socket precedentemente chiuso. Ciò comporta che in caso di disconnessione dovuta a delle condizioni di network non ottimali, non sia possibile riutilizzare lo stesso socket per mantenere la sessione precedente attiva.

ListenJS fornisce un sistema di riconnessione aprendo un nuovo WebSocket, ma mantenendo le informazioni di uno specifico client come se la sessione precedente fosse ripristinata. Ciò significa che il client non dovrà preoccuparsi di rimettersi in ascolto delle risorse precedentemente specificate, ma sarà tutto gestito dalla libreria.

### 2.3.3 Multiplexing

Un'implementazione naïve per rimanere in ascolto di alcune risorse potrebbe essere quella di aprire un WebSocket per ogni risorsa che si vuole osservare, specificando l'URL della risorsa nel pacchetto di handshake. Per esempio, se volessimo rimanere in ascolto di due risorse `foo` e `bar`, potremmo aprire due WebSocket chiamando `new WebSocket('ws://myDomain.com/api/foo')` e `new WebSocket('ws://myDomain.com/api/bar')`. Tuttavia così facendo si aprirebbero due connessioni separate, che non è ideale per due motivazioni:

- I browser limitano il numero di connessioni HTTP contemporanee su uno stesso dominio[9]. La maggior parte dei browser moderni consentono un massimo di sei connessioni per dominio.
- Per aprire un WebSocket c'è un ritardo dovuto all'handshake specificato dal protocollo, che potrebbe essere evitato utilizzando lo stesso socket.

Per evitare questi problemi, ListenJS sfrutta intelligentemente le connessioni attive. Dunque, se non esiste un socket aperto per uno specifico dominio, allora ne aprirà uno, altrimenti riutilizzerà lo stesso socket per scambiare ulteriori messaggi. Per fare un parallelismo con l'esempio precedente, se con la libreria client si decidesse di osservare prima `foo` e poi `bar` sullo

stesso dominio, prima verrebbe aperto un socket per il dominio, mentre successivamente verrebbe riutilizzato il socket già aperto per comunicare l'intenzione di mettersi in ascolto su `bar`. In aggiunta, se nel ciclo di vita dell'applicazione non ci fosse più nessuna risorsa osservata su uno stesso dominio, ListenJS si occuperà di chiudere il socket per liberare le risorse utilizzate.

# Capitolo 3

# ListenJS: Dettagli implementativi

Abbiamo visto ad alto livello cosa offre ListenJS e in generale come queste funzionalità sono utili nello sviluppo di applicazioni real-time. Possiamo ora andare più a fondo nel funzionamento specifico, in modo da avere le basi per estendere e potenzialmente migliorare questi comportamenti.

## 3.1 Message channel

Il funzionamento del sistema si basa sullo scambio dei messaggi che hanno una struttura ben specifica, e conosciuta sia dal client che dal server. Ogni messaggio può essere un *Control Message* o un *Payload Message*, e sono dei messaggi testuali serializzati in JSON come segue:

```
{
   "action": "<ACTION>",
   "clientId": "<UIID>"
}
```

Listing 3.1: ListenJS Message Contract

Più altri valori specifici di una certa azione. In particolare, `<ACTION>` è una stringa che specifica il tipo di messaggio di controllo (o di payload), e può assumere i seguenti valori:

- Da client a server:

  - `HANDSHAKE`: Specifica un messaggio di controllo per inizializzare un nuova nuova connessione con un client. Facoltativamente può essere specificato un `clientId` come campo extra per ripristinare una sessione di uno specifico client. Se il server non riceve nessun messaggio di handshake entro un timeout specificato (default: 10000 ms) da quando un socket viene aperto, chiude automaticamente la connessione.
  - `SUBSCRIBE`: Specifica che il client desidera mettersi in ascolto dell'endpoint specificato con il campo `path`, tramite un observer gestito dal client con `observerId`.
  - `UNSUBSCRIBE`: Specifica che il client non desidera più ricevere notifiche di aggiornamenti dell'endpoint specificato con il campo `path`.
  - `HEARTBEAT`: Invia un messaggio di ping al server per verificare che sia ancora raggiungibile.

- Da server a client:

  - `HANDSHAKE_ACK`: Risponde ad un messaggio di handshake, chiudendo la procedura di riconoscimento del client. Se un `clientId` è specificato come campo, allora verrà ripristinato quel client, recuperando le informazioni sulle risorse che sta ascoltando. Altrimenti, genererà un `clientId` univoco come UUID (e.g. `B16B00B5-81F6-A420-81F6-008C3285ADB9`).
  - `SUBSCRIBE_ACK`: Conferma al client che la sottoscrizione alla risorsa in ascolto da `observerId` è avvenuta con successo.
  - `UNSUBSCRIBE_ACK`: Conferma al client l'avvenuta rimozione delle sottoscrizioni alla risorsa specificata dal suo `path`.
  - `HEARTBEAT_ACK`: Invia un messaggio di pong al client in risposta ad un ping.
  - `UPDATE`: Invia un messaggio di payload al client, specificandone il contenuto nel campo `body` e la risorsa di riferimento nel campo `path`.

## 3.2 Observer pattern across the stack

La Gang of Four definisce l'observer pattern[10] come un behavioral pattern che prevede che un certo numero di observer siano notificati quando un subject subisce un cambiamento di stato. Questo tipo di pattern è largamente utilizzato nel design di interfacce grafiche, dove si vuole ridurre al minimo il coupling tra le classi per non renderle meno riutilizzabili. Questo però non significa che possa essere utilizzato in ambiti al di fuori dello sviluppo di UI.

Nel nostro caso, l'observer pattern viene inteso come publish/subscribe, e funziona attraverso i differenti layer di comunicazione. Ciò significa che viene in primo luogo applicato ad un livello di astrazione più alto, dove gli observer sono i vari client, e il subject è il server.
Inoltre, lo stesso pattern viene applicato nella gestione del client, dove si hanno diversi observer che stanno in ascolto di una specifica risorsa. Quando un messaggio di payload viene ricevuto da un *manager* del client, questo viene inoltrato agli observer corretti che sono in ascolto su una risorsa.
In ListenJS il manager è un'istanza che gestisce la connessione ad uno specifico dominio. Ogni manager tiene in memoria una lista di observer. Quando si effettua la chiamata `listen(path)`, viene creato un nuovo observer per il path specificato, ed aggiunto al manager.
Un observer è descritto dalle seguenti proprietà:

- `id`: Identifica l'observer univocamente
- `path`: Indica il path che l'observer desidera osservare
- `status`: Indica lo stato dell'observer, che può essere:
  - `WAITING_HANDSHAKE`: L'observer è stato appena inizializzato e sta attendendo che un handshake sia completato per potersi sottoscrivere
  - `SUBSCRIBING`: Il manager ha inviato la richiesta di osservare l'endpoint, ma sta attendendo di ricevere una conferma

- `UNSUBSCRIBING`: Il manager ha inviato la richiesta di non osservare più l'endpoint
- `LISTENING`: L'observer sta correttamente ascoltando gli aggiornamenti ad una risorsa
- `ERROR`: Un errore imprevisto

I passaggi di stato da uno all'altro rispetto agli eventi che hanno luogo nel manager sono descritti in figura 3.1.

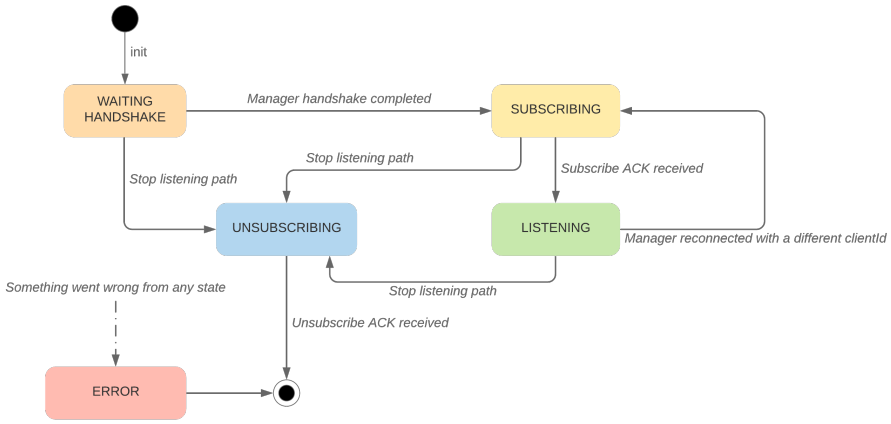


Figura 3.1: ListenJS client observer statuses

Come l'observer, anche il manager mantiene uno stato interno che indica la fase della connessione. Lo stato del manager è legato agli eventi ricevuti dal socket, ma è più preciso sulla fase di connessione:

- `IDLE`: Manager appena inizializzato
- `CONNECTING`: Il socket è in fase di connessione
- `HANDSHAKING`: Il socket è connesso ma il manager non ha ancora finito la fase di handshake
- `CONNECTED`: Il socket è connesso e l'handshake è stato eseguito correttamente
- `DISCONNECTED`: Il socket è disconnesso

- `RECONNECTING`: Il socket è in fase di connessione dopo che un altro socket è stato chiuso

- `ERROR`: Un errore imprevisto

I cambiamenti di stato in base agli eventi del socket e il numero di observer sono meglio rappresentati in figura 3.2.

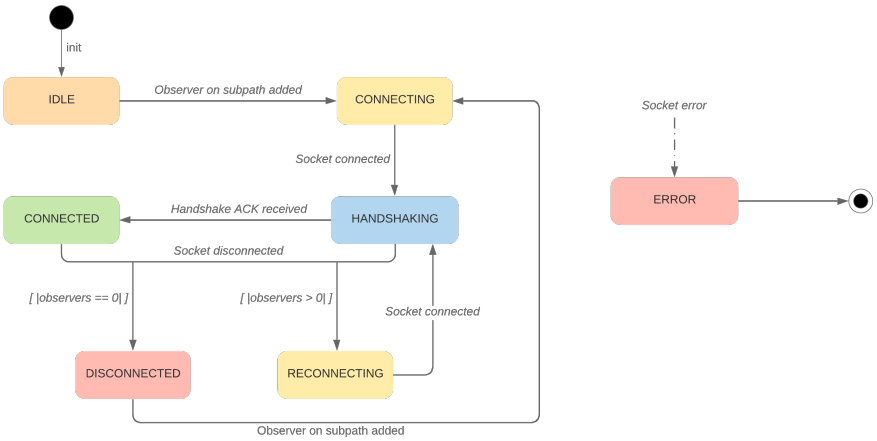


Figura 3.2: ListenJS client manager statuses

Il manager gestisce sia gli observer, che il socket. In particolare, quando un nuovo observer viene aggiunto al manager, esso utilizza il proprio stato interno per capire quando sia opportuno inviare i messaggi di controllo. Per esempio, se viene aggiunto un nuovo observer, ma lo stato del manager è `CONNECTING` o `HANDSHAKING`, allora dovrà mettersi in attesa che il proprio stato sia propriamente `CONNECTED`. Una volta `CONNECTED` (o nel caso lo fosse già in fase di aggiunta), potrà procedere ad inviare tramite il socket i messaggi di controllo per fare il `SUBSCRIBE` del `path` specificato. Inoltre, il manager si occupa di aprire un nuovo socket appena un nuovo observer viene aggiunto. Quando invece vengono rimossi tutti gli observer per lo stesso manager, allora si occuperà di chiudere il socket attuale.

## 3.3 Implementazione features

Nella sezione precedente è stato definito formalmente il canale di comunicazione tra libreria client e libreria server, e l'interazione tra i componenti principali. Per quanto riguarda i dettagli implementativi specifici dell'una o dell'altra, vale la pena analizzare singolarmente come le feature siano state implementate.

### 3.3.1 Multiplexing

Ogni richiesta che viene fatta per mettersi in ascolto di una risorsa, deve essere gestita da un manager. Un manager però, come abbiamo visto, gestisce solo un socket alla volta. Il componente software che si occupa di gestire i manager è stato chiamato *multiplexer*.

Il multiplexer si occupa di assicurarsi che per ogni richiesta effettuata, ci sia sempre solamente un manager attivo per ogni dominio di destinazione. Il funzionamento è piuttosto intuitivo: Ogni richiesta del client di ascoltare una risorsa passa attraverso il client, che fa da proxy per quella richiesta, inoltrandola al manager corretto. Il flusso per questa gestione può essere riassunto come segue:

1. Il client invia una richiesta al multiplexer
2. Il multiplexer fa il parsing dell'URL di destinazione, estraendo il dominio del server che si vuole ascoltare
3. Il multiplexer a questo punto controlla se nella propria map di manager, indicizzati per dominio di destinazione, è presente un manager
4. Se esiste già un manager per quel dominio, inoltra la richiesta. Altrimenti, crea un nuovo manager, lo salva nella map, e inoltra la richiesta

Oltre a gestire i manager per limitare i socket ad uno per dominio, il multiplexer si occupa anche di fare routing degli eventi scatenati dai diversi manager. In tal modo chi utilizza la libreria può rimanere in ascolto di

eventi di connessione e disconnessione in un unico punto, nascondendo così i dettagli implementativi dello smistamento dei socket.

### 3.3.2 Disconnection detection

L'implementazione browser dei WebSocket non prevede nessun meccanismo di rilevazione di disconnessione, a meno che il socket non venga chiuso esplicitamente da una delle due parti con un handshake di chiusura completo. Per ovviare a questo problema, è stato sviluppato un meccanismo di keep alive implementato un livello sopra ai WebSocket.

Ogni manager in fase di handshake di ListenJS, da non confondere con l'handshake dei WebSocket, può inviare due parametri di configurazione: `pingInterval` e `pingTimeout`, espressi in millisecondi. Il server che riceve una richiesta di handshake può onorare i due parametri in arrivo, se presenti, e rispondere con gli stessi parametri, o rispondere con dei parametri diversi. Il client, una volta finito l'handshake, fa fede ai parametri ricevuti in risposta nell'ACK di handshake. Questo sistema è stato pensato in caso sia necessario limitare il carico di messaggi in arrivo sul server. Una volta terminato l'handshake, entrambe le parti hanno un contract sul quale fare affidamento per stabilire se una o l'altra parte è ancora connessa. Da un lato, il client invierà periodicamente un pacchetto di ping al server, con intervallo `pingInterval`, aspettandosi dal server una risposta (pong) entro `pingTimeout` millisecondi, o considerando il socket disconnesso dopo quel periodo di timeout. Dall'altro lato, il server si aspetterà di ricevere ogni `pingInterval` millisecondi, per rispondere con un pong. Se da un pong all'altro dovesse passare più tempo di `pingInterval + pingTimeout`, considererà il socket come disconnesso. Il pingTimeout è aggiunto per consentire un tempo massimo di ricezione dell'intervallo di ping.

### 3.3.3 Reconnection

Una volta che un socket viene chiuso, o si rileva una disconnessione con il sistema descritto in precedenza, il manager si occupa di riaprire un nuovo socket. Se dovesse fallire nel riaprie un socket, allora attenderà un certo

periodo di tempo, e ritenterà. La frequenza con cui un manager prova a riaprire un socket è scandita da un algoritmo chiamato *Truncated Exponential Backoff*.

$$E(x) = min(d * f^x * (1 + j * r), t)$$

La funzione calcola il prossimo delay da aspettare prima di effettuare un nuovo tentativo, sulla base del numero di tentativi $x$ e di una base esponenziale $f$. $d$ è un delay minimo, $t$ è una soglia massima, e $r$ è un fattore random generato ad ogni esecuzione che nel range $[-1, 1]$, e $j$ è una durata di jitter, per non ripetere diverse richieste nello stesso istante continuativamente.
ListenJS usa $f = 2$, $j = 0.5$, mentre i valore di minimo e di massimo possono essere configurati.

# Capitolo 4

# Valutazione

## 4.1 Applicazione di esempio

CORS

## 4.2 Confronto tra XHR polling e websockets

### 4.2.1 Responsiveness

### 4.2.2 Traffico dati

### 4.2.3 User experience

Tradeoff

## 4.3 Valore aggiunto di ListenJS

### 4.3.1 Utilizzo risorse e limitazioni browser

Limite socket aperti contemporaneamente

### 4.3.2 Immediatezza integrazione da parte di sviluppatori

### 4.3.3 Architettura

Separation of Concerns

# Conclusioni

# Sviluppi futuri

SSL delta messaging message queue

# 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.

[8] Robert C. Martin. SRP: The Single Responsibility Principle. In *Clean Architecture*, chapter 7. Prentice Hall, 2017.

[9] R. Fielding and J. Gettys and J. Mogul and H. Frystyk and L. Masinter and P. Leach and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, RFC Editor, June 1999.

[10] E. Gamma and R. Helm and R. Johnson and J. Vlissides. Behavioral Patterns: Observer. In *Design Patterns*, chapter 5. Addison-Wesley, 1994.