ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA
SCUOLA DI INGEGNERIA E ARCHITETTURA

CORSO DI LAUREA IN INGEGNERIA ELETTRONICA, INFORMATICA E TELECOMUNICAZIONI

MECCANISMI PER LO SVILUPPO DI WEB APP DISTRIBUITE: WEBSOCKET COME CASO DI STUDIO

Elaborata nel corso di: Sistemi Distribuiti

Tesi di Laurea di:
GIACOMO DRADI

Relatore:
Prof. ALESSANDRO RICCI

ANNO ACCADEMICO 2012-2013
SESSIONE III

PAROLE CHIAVE

Web App
Distribuite
Comunicazione
HTML5
WebSocket

Ai miei genitori e mia sorella, che mi hanno sostenuto
durante questo percorso di studi con amore,
entusiasmo, ottimismo.

Indice

Introduzione **ix**

1 Il Web come piattaforma **1**
1.1 I fondamenti del web . . . 2
1.1.1 Protocollo HTTP . . . 2
1.1.2 Representational State Transfer . . . 6
1.1.3 Web Browser . . . 9
1.2 Web statico e dinamico . . . 11
1.3 Web App moderne . . . 13
1.3.1 HTML5 . . . 14
1.3.2 Javascript . . . 15

2 Progettazione di Web App **19**
2.1 Architetture . . . 20
2.1.1 3-Layered architecture . . . 20
2.1.2 Client architecture . . . 24
2.1.3 Service Oriented Front-End Architecture . . . 29
2.2 Pattern di progettazione . . . 32
2.2.1 Model-View-Controller . . . 33

3 Meccanismi per la comunicazione **37**
3.1 Comunicazione pre-HTML5 . . . 38
3.1.1 Ajax e Long Polling . . . 39
3.1.2 Adobe Flash Socket . . . 46
3.2 HTML5: WebSocket . . . 48
3.2.1 Protocollo . . . 49
3.2.2 Opening Handshake . . . 51

3.2.3 Data framing 53
3.2.4 Invio e ricezione 57
3.2.5 Sicurezza 57
3.2.6 Client API 58
3.2.7 Server API 60

4 Caso di studio 63
4.1 PaintBoard, una lavagna condivisa 63
4.1.1 Introduzione 63
4.1.2 Obiettivi dell'applicazione 65
4.1.3 Tecnologie utilizzate 65
4.2 Architettura client-side 66
4.2.1 Struttura 66
4.2.2 Interazione con WebSocket 68
4.2.3 Interazione con Ajax Long Polling 71
4.3 Architettura server-side 74
4.3.1 Struttura 74
4.3.2 Interazione con WebSocket 75
4.3.3 Interazione con Ajax Long Polling 78
4.4 Discussione 81

5 Conclusioni 83

Introduzione

Al giorno d'oggi il World Wild Web non è più un semplice strumento per la condivisione di informazioni. Le tecnologie nate nel corso dell'ultimo decennio hanno permesso lo sviluppo di vere e proprie applicazioni Web (Web App) complesse, indipendenti e multi-utente in grado di fornire le stesse funzionalità delle normali applicazioni desktop. In questa tesi verranno trattate le caratteristiche di una Web App moderna, analizzandone l'evoluzione e il processo produttivo. Particolare attenzione sarà posta alle Web App distribuite e ai meccanismi di comunicazione client-server che queste nuove tecnologie hanno reso disponibili. I WebSocket, la tecnologia di riferimento di HTML5, saranno riportati come caso di studio e relazionati allo standard precedente ad HTML5, cioè Ajax e Long Polling.

La tesi è strutturata in cinque capitoli:

1. Il primo capitolo descrive alcuni componenti fondamentali del Web, come HTTP e ReST, e il processo evolutivo lo ha trasformato da document-oriented, come era alla sua nascita, ad application-oriented, come è allo stato attuale dell'arte. Espone inoltre le caratteristiche principali di una applicazione Web moderna.

2. Il secondo capitolo tratta della fase di progettazione di una Web App moderna, analizzandone le architetture possibili, i pattern applicabili e come si inseriscono nel contesto delle architetture già presenti come SOA e ReST.

3. Il terzo capitolo si concentra sull'aspetto distribuito di una Web App, illustrando i meccanismi di comunicazione già presenti, come Ajax o Flash Socket, e la novità portata da HTML5, i WebSocket. Particolare

attenzione sarà posta sulla facilità con cui i WebSocket riescono ad implementare specifiche comunicazioni come le Server Push.

4. Il quarto capitolo riporta un caso di studio a dimostrazione di quanto espresso nei capitoli precedenti. Ho progettato e sviluppato una Web App in due versioni: una con WebSocket e una con Ajax e Long Polling, discutendo le differenze e i vantaggi di una tecnologia rispetto all'altra in questo caso specifico.

5. Il quinto capitolo contiene le conclusioni tratte dall'elaborazione della tesi.

Capitolo 1

Il Web come piattaforma

Il World Wild Web, così definito dal suo creatore Tim Berners-Lee, nacque dalla necessità di condividere informazioni in modo semplice e costante. Sviluppato presso il CERN di Ginevra, il web, come usualmente viene chiamato, permise ai fisici e agli scienziati di divulgare istantaneamente la grande quantità di dati e informazioni prodotta ogni giorno. La comunicazione sul web viene regolata dal protocollo HTTP, in cui una macchina (il client) richiede una risorsa a un altra macchina (il server) specificandone l'indirizzo (URL). Attraverso l'uso di ipertesti (hypertext) è possibile collegare tra loro le risorse garantendone l'accesso. L'idea iniziale del web prevedeva che a ogni indirizzo fosse associato un documento, statico e predefinito, liberamente accessibile da tutti gli utenti. Negli anni successivi questa idea si è evoluta: a un indirizzo poteva essere associato un programma vero e proprio, in esecuzione sul server, in grado di produrre un output e restituirlo al client che lo aveva invocato.
Col passare del tempo, il web ha assunto una nuova caratteristica: l'utente non è più un utilizzatore passivo, ma un partecipante nella creazione delle risorse stesse: esse non sono più statiche, ma sono generate dinamicamente grazie al contributo degli utenti stessi. Ciò ha dato inizio ad una nuova generazione del web, il Web 2.0, definito così da Tim O'Reilly alla fine del 2004. Al giorno d'oggi il web è diventato una vera e propria piattaforma per lo sviluppo di applicazioni complesse, composte anche da più componenti in server diversi, in grado di collaborare tra loro. L'interazione con gli utenti è ancora un aspetto fondamentale del web, e nuovi strumenti stanno nascendo per garantire meccanismi di comunicazione e interazione migliori.

1.1 I fondamenti del web

Nonostante la continua evoluzione che ha avuto dalla sua nascita fino ad oggi, il web ha mantenuto costanti i suoi principi fondamentali. In questa sezione verranno illustrate le caratteristiche più importanti di tre di essi:

- il protocollo HTTP, che definisce come deve avvenire la comunicazione sul web.
- il Representational State Transfer (ReST), lo stile architetturale di riferimento per i sistemi di ipertesto distribuiti.
- il web browser, l'applicazione utilizzata dall'utente per inviare o ricevere informazioni sul web tramite il protocollo HTTP.

1.1.1 Protocollo HTTP

HTTP (Hyper Text Transfer Protocol) è un protocollo applicativo per la trasmissione di messaggi attraverso il web tra due entità, chiamate client e server. HTTP definisce i tipi di messaggi scambiati, la loro sintassi e la semantica del loro contenuto. Si basa sul protocollo TCP per la trasmissione di dati attraverso la rete. Le risorse trasferite comprendono sopratutto documenti HTML, ma possono essere anche immagini, risultato di query, script o altro. Ogni risorsa è identificata da un URI (Uniform Resource Identifer) che ne indica la locazione.

La comunicazione si basa sul paradigma request/response: il client invia un messaggio di richiesta al server (request), che restituisce un messaggio di risposta (response). HTTP rende possibili due tipi di connessione tra client e server: permanente o non permanente. Nel primo caso, dopo ogni request/response la connessione viene chiusa e nuove richieste da parte dello stesso client corrispondono ad aperture di nuove connessioni. La grande quantità di risorse scambiate ha portato, nella versione 1.1, alla possibilità di mantenere la stessa connessione con lo stesso client per più request/response, in modo da evitare il preambolo di apertura della connessione per ogni risorsa.

Non è presente il concetto di sessione: nel server non viene memorizzata alcuna informazione riguardo allo stato dei client connessi. Per questo

motivo il protocollo HTTP è definito stateless. Ciò rende più semplice lo sviluppo dei web server, in quanto non deve essere mantenuto uno stato per la comunicazione con ciascun client, ma rende più difficile la progettazione di applicazioni web complesse. Per superare questa limitazione sono stati ideati alcuni meccanismi, il più famoso dei quali prevede l'utilizzo dei cookie, che consentono di mantenere una sessione tra client e server. Le moderne applicazioni web possono usufruire di strumenti ancora più potenti per la memorizzazione di informazioni, come ad esempio WebStorage di HTML5.

Request message. Un messaggio di richiesta HTTP è composto da una request line, una serie di header e il body, questi ultimi separati da una linea vuota. La request line contiene il metodo richiesto, l'URI e la versione del protocollo, ed indica il metodo da applicare alla risorsa specificata dall'URI. I metodi presenti in HTML 1.1 sono i seguenti:

1. **GET.** Specifica che il client richiede al server la risorsa identificata dall'URI. Se tale risorsa esiste, il server la invierà attraverso la response.

2. **POST.** Questo metodo fornisce al client uno strumento per inviare informazioni al server. Il client manda una request di tipo POST con all'interno i dati che desidera inviare, specificando come URI la risorsa che è in grado di processare tali dati. La risposta del server sarà il risultato dell'operazione.

3. **PUT.** L'operazione di PUT è molto simile al POST, poichè consente al client di inviare dati al server. Tuttavia, il campo URI di una richiesta PUT identifica la risorsa nella quale il server deve inserire le informazioni appena ricevute.

4. **DELETE.** Il metodo DELETE indica che il client desidera eliminare la risorsa identificata dall'URI specificato.

5. **HEAD.** Questo metodo consente al client di ricevere informazioni sulla risorsa indicata dall'URI. A differenza del GET, il server non risponde inviando la risorsa stessa, ma solo le informazioni richieste, come ad esempio se la risorsa specificata esiste o no.

6. **TRACE.** Attraverso questo metodo il client può ottenere informazioni sul percorso di rete che un messaggio segue per arrivare al server.

Questi dati possono essere utilizzati per effettuare test o diagnosi. Nessuna risorsa deve essere inclusa nella richiesta.

7. **OPTIONS.** Permette di conoscere i metodi che il server consente di eseguire. Se l'URI identifica una risorsa specifica, la response conterrà i metodi eseguibili su quella risorsa; se l'URI è un asterisco (*), il metodo sarà relativo al server in generale.

8. **CONNECT.** Riservato ai server intermedi, consente di aprire un tunnel tra il client e il server di destinazione. Tale tunnel rimane aperto finchè la connessione è attiva.

Gli header sono formati da coppie chiave: valore e rappresentano una serie di informazioni supplementari facoltative riguardanti la richiesta, la risorsa o il client stesso. I più comuni sono:

1. Host. Nome del server a cui si riferisce l'URI. In HTTP 1.1 è obbligatorio poiché permette l'utilizzo di host virtuali basati su nomi.

2. User-agent. Identifica il browser con informazioni come nome, produttore, versione, ecc.

Response message. Ricevuto un messaggio di richiesta, il server genera un messaggio di risposta con la seguente struttura: una status line, una serie di header, una linea vuota e il body. La status line contiene la versione del protocollo, uno status code composto da tre cifre e un messaggio associato allo status code. Attraverso lo status code il client può sapere se l'operazione richiesta è andata a buon fine o se si sono verificati degli errori. Gli header più comuni di un messaggio di response sono:

1. Server. Equivalente all'User-Agent della request, fornisce informazioni sul server (tipo, versione).

2. Content-type. Indica il tipo (Media type) di contenuto restituito nel body. Tali tipi sono definiti MIME (Multimedia Internet Message Extensions) e sono registrati presso lo IANA (Internet Assigned Number Authority). Alcuni esempi sono text/html per i documenti HTML, text/plain per testo non formattato, text/xml per documenti XML e image/jpg per immagini JPEG.

Tabella 1.1: Status code in HTML/1.1

Codice	Descrizione
1xx	Informational. Messaggi di informazione.
2xx	Success. L'operazione ha avuto successo.
3xx	Redirection. L'operazione non è stata eseguita ma vengono fornite istruzioni aggiuntive.
4xx	Client error. La richiesta non è soddisfatta perchè errata.
5xx	Server error. La richiesta è corretta ma non può essere soddisfatta per problemi interni al server.

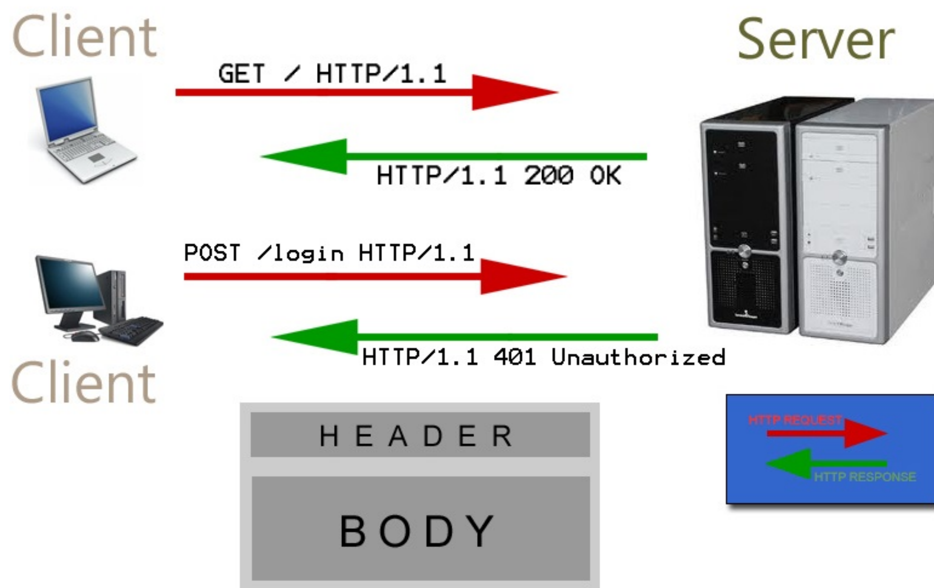

Figura 1.1: Esempio di request/response in HTTP/1.1 [10]

Gli header sono uno strumento utilizzato da client e server per comunicare informazioni riguardanti il body o per realizzare alcuni elementi di logica applicativa come cache o autenticazione. Attraverso gli header è infatti possibile allegare al messaggio vero e proprio dei metadati. Esistono quattro tipi di header: general headers, request headers, response headers e entity headers. I general headers si applicano ad entrambi i messaggi di request e response, ma non descrivono dettagli del body, che viene invece descritto dagli entity headers. Qualora il body non sia presente, gli entity headers si riferiscono alla risorsa target. Di particolare interesse per questa trattazione risulta essere il general header "Connection": esso abilita il mittente a specificare opzioni aggiuntive desiderate per quella particolare connessione [29]. Il protocollo che specifica il comportamento dei WebSocket, che sono stati introdotti con HTML5 per permettere una comunicazione full-duplex tra client e server, utilizzano infatti una "Connection: upgrade" per la gestione dell'handshake, come mostrato nell'esempio seguente. I WebSocket verranno trattati con particolare attenzione nel capitolo 3.

```
GET /mychat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==
Sec-WebSocket-Protocol: chat
Sec-WebSocket-Version: 13
Origin: http://example.com
```

1.1.2 Representational State Transfer

Il Representational State Transfer (ReST) rappresenta una delle architetture di riferimento per i sistemi ipertestuali distribuiti, come il World Wild Web. ReST si concentra sul ruolo dei componenti e le loro interazioni senza fornire dettagli sulla effettiva implementazione. In ReST, la comunicazione avviene attraverso il trasferimento della rappresentazione di una risorsa in un formato scelto tra una serie di standard, deciso a seconda della natura della risorsa stessa e dei bisogni dell'applicazione che la richiede. La rappresentazione può essere nello stesso formato della risorsa stessa, oppure diverso, ma ciò rimane nascosto dall'interfaccia uniforme dei componenti.

L'astrazione fondamentale di ReST è la risorsa. Qualsiasi entità sufficientemente rilevante può essere una risorsa, come documenti HTML, immagini, uno stato dell'applicazione, o una collezione di altre risorse. Una risorsa può essere statica, nel caso in cui una volta creata non si modifichi nel tempo, o dinamica. In questo caso deve mantenere lo stesso valore semantico, seppur con un contenuto diverso. Ogni risorsa deve essere associata ad da almeno un identificatore, e un identificatore può essere associato a solo una risorsa. Tuttavia, identificatori diversi possono riferirsi alla stessa risorsa. Le risorse consentono ad un utilizzatore di referenziare il concetto, invece che la singola rappresentazione.

Tabella 1.2: Data elements in ReST

Data element	**Esempio nel web**
Risorsa	target concettuale di un collegamento ipertestuale.
Rappresentazione	documento HTML, immagine JPG
Identificatore di risorsa	URI (URL, URN)
Metadati di rappresentazione	Media-type, last-modified time
Metadati di risorsa	source-link, alternates
Dati di controllo	if-modified-since, cache-control

Una rappresentazione è lo stato corrente della risorsa a cui si riferisce, espresso in un formato specifico, chiamato media-type, a cui sono aggiunti i metadati di rappresentazione, definiti come coppie chiave-valore, per descriverne il contenuto. I metadati di rappresentazione si differenziano da quelli di risorsa perché sono specifici della rappresentazione a cui si riferiscono.

Le caratteristiche fondamentali dell'architettura ReST sono:

1. **Client-server.** Coerentemente col principio di separation of concerns, un server propone uno o più servizi e aspetta richieste di questi servizi da parte di un client, comunicando attraverso componenti chiamati connettori.

2. **Stateless.** Ogni richiesta del client deve contenere tutte le informazioni necessarie per essere compresa dal server, senza riferimenti a richieste precedenti. Nel server non viene memorizzato alcun concetto di sessione o contesto, che deve invece essere mantenuto nel client. Questo principio garantisce server più semplici e scalabili, ma aumenta la quantità di overhead inviato a ogni request e non consente al server di verificare il corretto comportamento dell'applicazione. Esistono due tipi di stato:
   - Application state. Lo stato dell'applicazione client, univoco per ogni utente. Non deve essere memorizzata nel server.
   - Resource state. Lo stato di una risorsa, è uguale per tutti i client ed è mantenuto dal server. Viene restituito al client sotto forma di rappresentazione.
3. **Cacheable.** Le informazioni ricevute dal server come response possono essere identificate come cacheable e il client può memorizzarle. In caso di successivo bisogno, il client può riutilizzarle senza eseguire una nuova request. Ciò migliora l'efficienza e le performance percepite dall'utente, ma diminuisce l'affidabilità (reliability) nel caso in cui una informazione sul server sia cambiata rispetto alla sua memorizzazione in cache.
4. **Uniform interface.** Client e server comunicano attraverso interfacce ben definite, immutabili rispetto alle singole implementazioni (portabilità). Server e client possono essere sviluppati e modificati indipendentemente, migliorando la scalabilità. Il concetto base di questo principio e l'uniformità: non è importante quali standard siano scelti per l'accesso alle risorse ma è importante che chiunque voglia accedervi lo faccia allo stesso modo.
5. **Layered system.** I componenti sono organizzati in livelli indipendenti che svolgono una precisa funzionalità. Ogni livello può comunicare solo coi livelli adiacenti. Migliora scalabilità e indipendenza, ma diminuisce le performance percepite dall'utente a causa dell'overhead che ogni livello può aggiungere.
6. **Code on demand.** Opzionalmente, un client può richiedere codice dal server ed eseguirlo localmente, solitamente sotto forma di script.

Lo scopo principale dell'architettura ReST è quindi garantire scalabilità, sicurezza, indipendenza, e semplicità di evoluzione di un sistema distribuito attraverso separation of concerns, interfacce uniformi tra componenti e scambio messaggi autosufficienti alla propria comprensione. Obiettivo è ancora migliorare l'efficienza grazie all'utilizzo delle cache.

1.1.3 Web Browser

I Web Browser sono applicazioni desktop in grado di richiedere e visualizzare contenuti dal web. Essi sono la tipologia di web client più diffusa, ma non l'unica: proxy o applicazioni client custom-built (vedere sezione 1.2) replicano infatti molte delle funzionalità tipiche di un browser. Le informazioni da mostrare sono solitamente espresse come pagine HTML, ma possono essere anche PDF, contenuti multimediali (audio, video) o script. Come i contenuti provenienti dal web debbano essere effettivamente visualizzati dai browser è stato specificato dal World Wild Web Consortium [1], tuttavia tali specifiche non sono state completamente adottate allo stesso modo da tutti i produttori, causando problemi di inconsistenza e incompatibilità. Grazie all'evoluzione continua dei browser, queste differenze stanno però scomparendo.

Un browser è composto da diversi componenti, ognuno specifico per il proprio compito, che collaborano per ottenere l'output desiderato. Essi sono:

1. User interface. Comprende la barra degli indirizzi e tutti quegli elementi grafici al di fuori dell'area dove viene visualizzata la pagina richiesta.

2. Browser engine. Interfaccia per la manipolazione e l'interrogazione del rendering engine.

3. Rendering engine. Uno dei componenti principali, è responsabile della visualizzazione della risorsa richiesta. Nel caso di un documento HTML, deve eseguire il parsing di HTML e CSS e mostrarne il risultato.

4. Networking. Effettua le comunicazioni sulla rete, ad esempio inoltrando request HTTP e accettando response HTTP. Si occupa anche della

creazione della connessione e della gestione dei proxy eventualmente specificati dall'utente.

5. Javascript Interpreter. Effettua il parsing del codice Javascript e lo esegue. L'ottimizzazione e le performance di questo componente sono aspetti principali data l'importanza che ha assunto Javascript nelle moderne applicazioni web.

6. UI backend. Disegna widget grafici come combo box e finestre, utilizzando API specifiche del sistema operativo.

7. Data storage. Ha il compito di memorizzare sull'hard disk dati persistenti, come ad esempio i cookie. HTML5 ha reso possibile utilizzare database (leggeri) interni al browser, il cui mantenimento è affidato a questo componente.

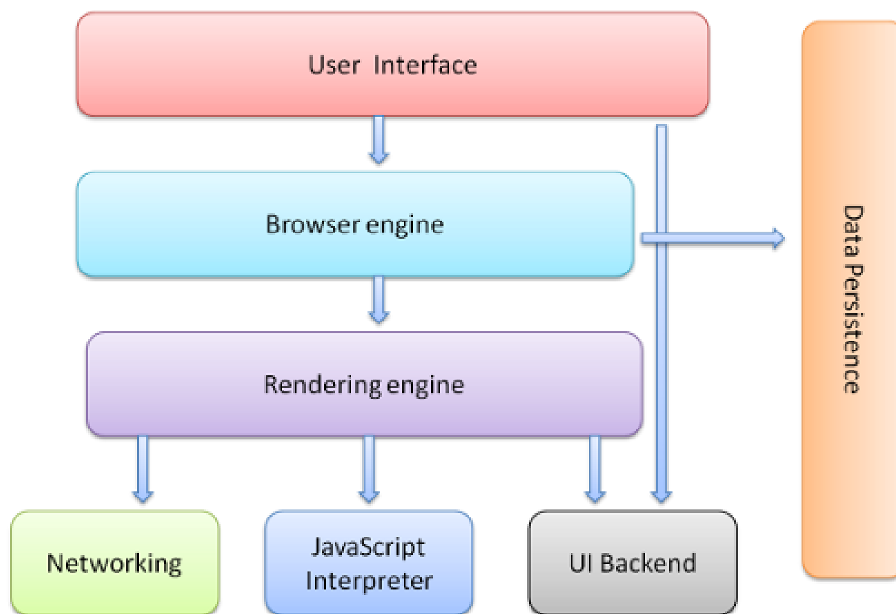

Figura 1.2: Componenti principali di un Web Browser [6].

Il rendering engine è uno dei componenti principali. Il suo ruolo è quello di prelevare il documento dal network layer, renderizzarlo e disegnarlo. Il

funzionamento dell'engine è single-threaded e resta in attesa di eventi in loop infinito. Un evento potrebbe essere il calcolo della nuova posizione di un elemento o modificarne il contenuto. Il browser, in caso di necessità, potrebbe effettuare più connessioni in parallelo per lo stesso documento per migliorarne l'efficienza del caricamento. Google Chrome, diversamente dagli altri browser, utilizza per ogni tab una istanza di rendering engine diversa.

Quando l'uente digita un URL, il netowkr layer richiede il documento associato a quell'indirizzo. Se la pagina contiene risorse esterne, come fogli di stile, Javascript o immagini, il browser procederà a richiederli dall'alto verso il basso con nuove request. Tutti gli elementi che il rendering engine ha ottenuto dal network layer vengono memorizzate in una struttura gerarchica ad albero chiamata DOM (Document Object Model). Gli script eventualmente rilevati possono essere eseguiti immediatamente oppure, se segnalati come deferred, essere messi in attesa che il DOM sia completamente generato. Il secondo caso si applica a quegli script che agiscono su un elemento specifico della pagina, che deve quindi essere presente nel DOM al momento dell'esecuzione. Parallelamente alla popolazione del DOM, il rendering engine crea un altro albero, il Render tree, contenente la rappresentazione visuale del DOM. A ogni elemento del Render tree è associato un elemento del DOM. Infine gli elementi del Render tree vengono disegnati sullo schermo.

1.2 Web statico e dinamico

Con il passare degli anni il World Wild Web si è evoluto da un semplice sistema per la condivisione di documenti a un vero e proprio environment per lo sviluppo di applicazioni general-purpose. Questa trasformazione è avvenuta come successione di diverse fasi evolutive: la prima, iniziale, in cui il web rifletteva le sue origini document-oriented, una seconda, in cui le potenzialità del web come application enviornment emersero, e l'ultima, quella attuale, in cui il web si afferma sempre di più come l'application environment di riferimento per il software moderno.

Nei primi anni di vita del web, ogni pagina rappresentava un documento ed era possibile navigare tra essi attraverso l'uso di ipertesti. A ogni click corrispondeva la richiesta e quindi la ricezione di una nuova pagina dal

server. Alcune pagina potevano contenere form, attraverso i quali potevano essere inviate informazioni al server. Nessuna comunicazione asincrona tra client e server era necessaria. Con l'avvento di Javascript, CSS e il Document Object Model (DOM), le pagine iniziarono a supportare grafica più avanzata e animazioni. Alla fine degli anni 90 le potenzialità del web come strumento commerciale vennero percepite dalle aziende; questo porto all'integrazione di plugin come Flash, RealPlayer e Quicktime per la riproduzione di contenuti multimediali. Il web stava andando in una nuova direzione diversa da quella originariamente prevista dai suoi ideatori.

Nei primi anni 2000 l'introduzione della comunicazione asincrona sul web (conosciuta come Ajax, Asynchronous JavaScript and XML) rese possibile l'idea del web come application platform. L'idea di Ajax consisteva nel permettere al client di aggiornare solo un elemento della pagina alla volta, richiedendolo in modo asincrono al server, senza dover richiedere tutta la pagina. In questo modo si rese possibile separare l'interfaccia utente e i suoi aggiornamenti dalle web request. Sebbene fosse un meccanismo e non uno stile architetturale vero e proprio, esso si diffuse ampiamente e con esso aumentò la complessità delle applicazioni web. Ciò rivelò inequivocabilmente l'inadeguatezza dei browser: la mancanza di API per l'accesso alle risorse sottostanti (come device o hardware) rese molto difficile la realizzazione di web application complesse portabili tra i vari browser.

In questo contesto nacquero le cosiddette RIA (Rich Internet Application). L'idea delle RIA era quello di riutilizzare tecnologie desktop già esistenti rendendole disponibili sul web sotto forma di plugin del browser o attraverso un runtime custom. Esempi di framework RIA furono Adobe AIR, Google Web Toolkit, Microsoft Silverlight e JavaFX. Nonostante un boom iniziale, la diffusione di queste tecnologie diminuì rapidamente proprio a causa della necessità di plugin aggiuntivi specifici. Una nuova tecnologie che assunse particolare interesse fu WebWidgets. Un WebWidget è una applicazione web precompilata che può essere installata ed eseguita su un dispositivo mobile come una normale applicazione binaria, a differenza delle normali applicazioni web che vengono eseguite attraverso il browser.

Oggigiorno sono due le tipologie di applicazioni prevalentemente diffuse: da una parte, la venuta di HTML5 e le nuove tecnologie come WebSocket

e WebGL permettono di sviluppare applicazioni web complesse e dinamiche, sfruttando il web come piattaforma e il browser come client universale; dall'altra, la maggior parte dei servizi web mette a disposizione una applicazione client custom-built specifica nativa (es: Facebook, Twitter, Skype). Sebbene un applicazione client custom specifica per il singolo servizio possa sembrare una soluzione migliore, credo che le tecnologie ora a disposizione per lo sviluppo di applicazioni web moderne garantiscano gli stessi risultati. Inoltre, le applicazioni web possono essere rese disponibili e aggiornate istantaneamente in tutto il mondo, senza bisogno di nessuna installazione. Questo aspetto le rende sicuramente più potenti ed è su di esse che il futuro sembra essere rivolto.

1.3 Web App moderne

Grazie all'avvento di HTML5 e delle tecnologie ad esso associate, le potenzialità delle applicazioni web sono aumentate esponenzialmente. Le caratteristiche che una Web App moderna possiede sono:

1. Indipendenza e autosufficienza: nonostante possa prendere dati e informazioni da web-services o applicazioni esterne, l'utente è in grado di svolgere i propri task senza dover navigare in altri siti o risorse.

2. Multi-utenza: una stessa istanza di applicazione può essere usata da più utenti diversi che cooperano e collaborano tra loro; la Web App deve perciò essere in grado di aggiornare i propri contenuti dinamicamente mediante comunicazione asincrona e/o condivisa.

3. Stand-alone: non devono essere richiesti plugin esterni, widget o altro; tutto ciò di cui ha bisogno l'applicazione per funzionare è il web browser.

4. Presenza di Rich User Interface (RUI): l'interfaccia utente deve essere esteticamente gradevole e consentire l'interazione in maniera chiara e coerente. Possono essere presenti elementi grafici e multimediali.

5. Offline: l'applicazione può essere eseguita offline, aggiornandosi quando si è connessi, attraverso l'utilizzo di cache, WebStorage o IndexedDB per il mantenimento di codice, piccole quantità di dati o interi database.

Tra questi, un aspetto particolarmente importante è la multiutenza: le applicazioni web moderne superano il paradigma client-server request/response tipico del web classico, consentendo alla stessa istanza di applicazione di interagire con più client contemporaneamente. Allo stesso modo, l'applicazione potrebbe non essere allocata interamente su un unico server, ma essere distribuita su più server diversi. Le architetture e i meccanismi con cui questa distribuzione può essere pensata e realizzata sarà oggetto principale di questo trattato.

Le tecnologie che hanno aperto la strada allo sviluppo e alla diffusione delle applicazioni web sono state HTML5 e Javascript. Sebbene la prima versione di Javascript risalga alla fine degli anni 90, l'importanza che esso ha assunto nel corso degli anni lo ha reso una tecnologia praticamente indispensabile per le Web App moderne.

1.3.1 HTML5

HTML5 è un linguaggio di markup per la strutturazione delle pagine web. La stesura delle specifiche è a carico del W3C (World Wide Web Consortium) e del WHATWG (Web Hypertext Application Technology Working Group), gruppo composto da rappresentanti delle maggiori aziende mondiali del settore come Apple, Mozilla ed Opera. Nonostante sia il diretto successore di HTML4 e mantenga la sua filosofia general-purpose, molte delle sue nuove caratteristiche sono rivolte a rendere il web un ambiente più confortevole per applicazioni desktop-like. Queste caratteristiche includono:

1. Supporto per l'offline: HTML5 fornisce un sistema di gestione delle cache attraverso il quale è possibile continuare ad eseguire la applicazione anche senza essere connessi.

2. Storage locale: per il mantenimento di grandi quantità di dati HTML5 supporta un database basato su key-value memorizzato localmente.

3. Canvas API: API per disegnare canvas 2D con cui realizzare grafica interattiva.

4. Supporto audio e video integrato attraverso tag specifici senza ricorrere a plugin esterni.

5. Caricamento di script asincrono.
6. Supporto per il drag and drop.

L'obiettivo di HTML5 è quindi sostanzialmente semplificare il lavoro degli sviluppatori (e quindi delle aziende coinvolte nella creazione di Web applications) che negli ultimi anni hanno dovuto elaborare soluzioni, a volte anche molto complesse, per superare le limitazioni di HTML, un linguaggio non nativamente progettato per lo sviluppo di articolate interfacce grafiche e sofisticate interazioni con l'utente. Una interessante caratteristica di HTML5 è che tutte le strutture di HTML4 (tag) rimangono validi e funzionanti. Questo perché buona parte del web è ancora rappresentato da siti statici o document-oriented. In questi termini, preservare il legacy è compito svolto.

HTML5 porta a una evoluzione del modello di markup, che non solo si amplia per accogliere nuovi elementi di chiaro valore semantico come <article>, <header>, <footer>, ma modifica anche le basi della propria sintassi e le regole per la disposizione dei contenuti sulla pagina. Le API JavaScript vengono inoltre estese per supportare tutte le funzionalità di cui una applicazione moderna potrebbe aver bisogno: WebWorker, WebSocket, IndexedDB sono alcune di esse.

1.3.2 Javascript

Javascript è un linguaggio di programmazione interpretato creato principalmente per la manipolazione di pagine web. Fu sviluppato nel 1995 da Netscape ed era molto simile a Java, ma più semplice e poteva essere facilmente incluso nelle pagine web. La sintassi ricordava ampiamente quella del C++. La sua semplicità consisteva nell'essere composto da un insieme di funzioni facilmente integrabili all'interno di una pagina web e richiamabili in seguito a specifici eventi. Ciò favorì la sua ampia diffusione in poco tempo, ma il suo potere limitato non permetteva di creare applicazioni web basate su Javascript di grandi dimensioni.

Con l'avvento di Ajax agli inizi del 2000, fu possibile utilizzare specifiche API di Javascript per inoltrare al server richieste asincrone senza bisogno di dover ricaricare l'intera pagina; i dati ricevuti dal server potevano essere inseriti nel documento dinamicamente. Data questa enorme potenzialità si

ebbe la nascita di framework come JQuery, in grado di semplificare lo sviluppo di applicazioni Javascript complesse e dinamiche enormemente garantendone inoltre la compatibilità cross-browser. Se inizialmente ai Javascript engine dei browser non erano richieste particolari prestazioni, la diffusione in larga scala di client-side Javascript portò allo sviluppo di interpreter Javascript sempre più potenti, come il V8 JavaScript di Google Chrome.

Col tempo sono stati sviluppati framework che rendono possibile l'utilizzo di Javascript anche per la componente server-side dell'applicazione web. Uno dei più famosi è Node.js, che per l'esecuzione utilizza proprio l'engine V8 di Google. Seguendo lo stile di Javascript, Node.js si basa su un modello event-driven, che favorisce l'implementazione di particolari tipi di interazioni come le comunicazioni asincrone e le server push.

Javascript non è tuttavia privo di inconvenienti. La mancanza di features come classi, moduli e namespaces fa si che le applicazioni Javascript siano un insieme di event-handlers e funzioni non strutturate, rendendo la progettazione e lo sviluppo in larga scala particolarmente complesso. Inoltre, caratteristiche come flessibilità e riutilizzo sono difficilmente applicabili. Per superare questi problemi sono stati ideati nuovi linguaggi: Typescript ad esempio è un linguaggio basato su Javascript, che viene compilato in normale Javascript, che supporta interfacce, classi, moduli, annotazioni e altri aspetti tipici della programmazione object-oriented class-based.

Con l'arrivo di HTML5, nuove tecnologie ad esso associate facenti uso di Javascript sono nate e hanno immediatamente assunto una importanza notevole: WebSocket e WebWorker ne sono un esempio. I WebSocket offrono un canale di comunicazione bidirezionale full-duplex tra client e server e sono particolarmente nello sviluppo di Web App distribuite, mentre i WebWorker rappresentano il concetto di thread, componenti in grado di eseguire codice in modo asincrono, eseguendo ad esempio calcoli complessi o operazioni di I/O, con cui si può interagire attraverso una semplice chiamata a metodi. Una analisi dettagliata dei WebSocket sarà fornita nel capitolo tre.

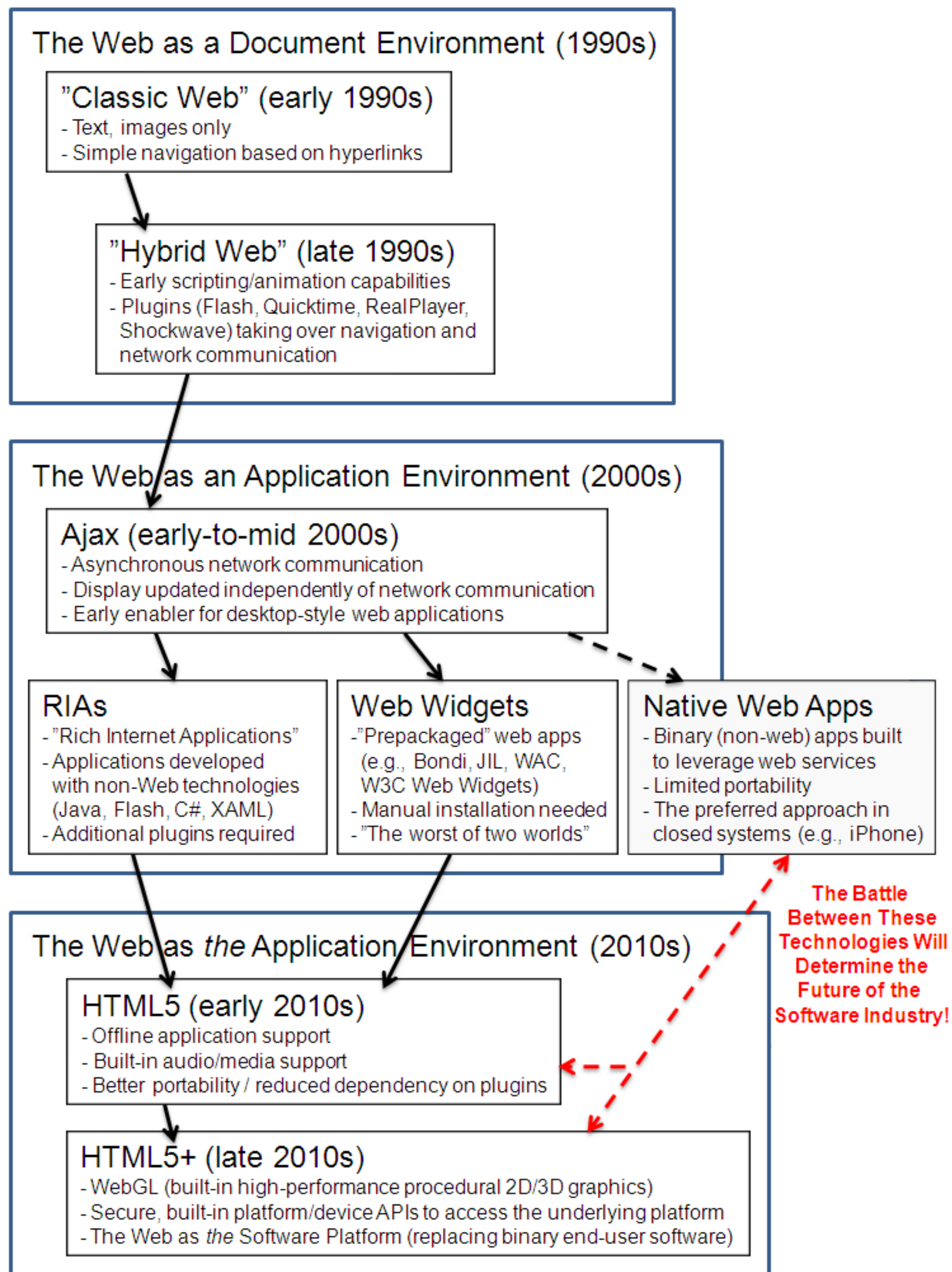

Figura 1.3: L'evoluzione del web [27].

Capitolo 2

Progettazione di Web App

Sebbene le nuove tecnologie (HTML5, WebSocket, ecc) forniscano validi strumenti per la realizzazione delle moderne applicazioni web, è necessario anche analizzare come i principi dell'ingegneria del software si applichino al contesto del Web Development. Il precedente capitolo ha illustrato brevemente le tecnologie più recenti a disposizione per gli sviluppatori, senza però specificare come le Web App debbano essere pensate e costruite. Questo aspetto, reso necessario dalla crescente complessità delle applicazioni, riguarda l'architettura, definita in termini di struttura tra i vari componenti e delle loro relazioni. L'obiettivo di questo capitolo è trovare e illustrare i principi architetturali fondamentali delle Web App moderne. Per poter sviluppare applicazioni complesse che garantiscano scalabilità e riusabilità e che siano in grado di soddisfare i requisiti funzionali e non funzionali richiesti è infatti necessario un processo di produzione ben definito. In particolare verrà analizzata la fase di progettazione, concentrandosi sulle tipologie di architettura che una moderna Web App può avere e sui design pattern che possono essere applicati per risolvere problematiche ricorrenti. Attenzione sarà anche dedicata al problema dei thin vs fat clients: ovvero le due modalità di allocazione della logica applicativa, sul client (thin) o sul server (fat). La prima, più tradizionale, ha il vantaggio di avere client semplici limitati ad agire da interfaccia utente, mentre la seconda, di crescente diffusione, prevede client complessi, a cui delegare parte della logica applicativa, che garantiscono però più efficienza dal punto di vista dell'utente e maggiore scalabilità dal punto di vista dell'ingegnerizzazione.

Alcune problematiche risultano essere ricorrenti nella progettazione sia del software in generale che delle applicazioni web. Per risolvere questi problemi è bene far ricorso ai design pattern: soluzioni progettuali, definite in termini descrittivi o tramite modelli logici che si possono applicare a categorie di problemi con caratteristiche simili. Una sezione del capitolo sarà dedicata ai design pattern specifici delle Web App.

2.1 Architetture

Nell'ambito dei sistemi software, con il termine architettura di un sistema si intende l'organizzazione fondamentale del sistema stesso, espressa in termini dei suoi componenti, delle relazioni tra loro e con l'ambiente esterno, e i principi che ne governano la progettazione e l'evoluzione [16]. Nei sistemi distribuiti, l'architettura si esprime solitamente in relazione ai componenti che lo formano, alle informazioni che si trasmettono e alle modalità con cui sono tra essi integrati per realizzare il sistema finale. Le web application, essendo sistemi distribuiti, seguono questo principio.

2.1.1 3-Layered architecture

L'approccio standard dell'ingegneria del software per la gestione della complessità di un sistema consiste nell'applicazione del principio di separazione delle competenze (separation of concerns). Tale principio si traduce solitamente nella divisione del sistema in diversi livelli logici. In questo tipo di architettura, definito a livelli (layered-architecture), ogni layer racchiude uno o più componenti che realizzano una specifica funzionalità. Un livello può richiamare componenti del livello sottostante e può essere richiamato da componenti del livello superiore. Il controllo del flusso è perciò in direzione top-down/bottom-up [28]. In quanti livelli separare la propria applicazione e i compiti di ciascun livello sono soggetti a variazioni, ma lo standard nelle applicazioni web prevede l'utilizzo di 3 layer (3-layered architecture). Questi livelli sono:

1. Presentation layer.
2. Business Logic layer.
3. Data Source layer.

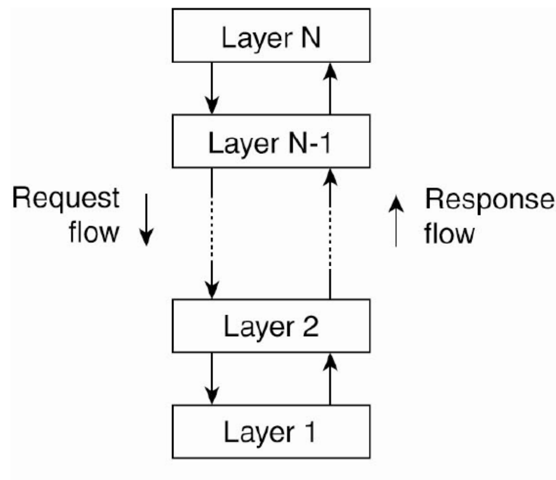

Figura 2.1: Architettura a layer [28].

Ognuno di questi layer deve essere indipendente dall'effettiva implementazione dei layer adiacenti; in questo modo, è possibile apportare modifiche a un singolo livello senza ripercussioni sul resto del sistema. Inoltre, poiché racchiude in modo atomico una funzionalità, un livello può essere riutilizzato in altri sistemi.

Presentation Layer. Il livello di presentazione è il punto di accesso dell'utente all'applicazione. A ogni URL dell'applicazione è infatti associato un handler contenuto in questo layer. Sono possibili diversi soluzioni per la strutturazione del presentation layer: la più comune nelle Web App moderne prevede l'utilizzo del pattern architetturale Model-View-Controller. In questo pattern, i controller gestiscono le richieste dell'utente e le delegano ai componenti opportuni del business layer sottostante. Il risultato dell'operazione viene restituito sotto forma di entità del modello, che implementano il dominio dell'applicazione. Queste entità saranno poi mostrate all'utente secondo una particolare vista che dipende dall'operazione stessa: potrebbe essere un documento HTML intero o semplici dati formattati con JSON o XML. Questi ultimi potrebbero essere manipolati con Javascript prima di essere visualizzati definitivamente. Il pattern ModelViewController sarà discusso in dettaglio nelle sezioni successive.

Business Logic Layer. Questo livello, anche chiamato Domain Logic Layer, è responsabile dell'esecuzione delle operazioni e dei servizi fondamentali dell'applicazione. Come il livello di presentazione, opera con entità del modello del dominio. Questo livello può essere progettato applicando diverse strategie; solitamente si utilizza un modello ad oggetti per la descrizione del dominio, specificando struttura, comportamento e interazione dei componenti che ne fanno parte. Progettare componenti che svolgano operazioni atomicamente è consigliato per facilitare la riusabilità del codice. Il prodotto del Business logic layer è sia quell'insieme di operazioni e algoritmi che definiscono l'applicazione come tale, sia il workflow, cioè la serie di passi successivi per ottenere, a partire da un input, l'output prefissato. Inoltre, dati e informazioni vengono processati e passati ai due layer adiacenti attraverso questo livello.

Data Source Layer. Il data source layer è responsabile della comunicazione con altri sistemi come database, web services e filesystem. La soluzione storicamente adottata dagli sviluppatori di Web App per il mantenimento dei dati è l'utilizzo di database relazionali come MySQL. In questo caso, il data source layer è responsabile del marshaling da oggetti del dominio, utilizzato dalla logica applicativa, a rappresentazioni compatibili col database adottato, e viceversa. Deve inoltre gestire la connessione, l'esecuzione delle query e la disconnessione dal database. Nuove tecnologie come WebServices, SOA e ROA hanno permesso alle applicazioni di non dover necessariamente conservare dati persistenti internamente: scopo di questo layer è anche la gestione e l'esecuzione delle chiamate a questi servizi.

La separazione in livelli logici non corrisponde necessariamente a una divisione fisica. Ai tre livelli logici (layer) possono corrispondere infatti diverse combinazioni di livelli fisici (tier). In una sistema client-server, il client potrebbe essere un semplice display, potrebbe contenere l'intero presentation layer, o potrebbe includere il business layer totalmente o in parte. Nel web precedente alle moderne Web App, tutti i layer erano allocati fisicamente nel server: alla richiesta di un documento da parte del client il server prelevava le informazioni richieste dal database; trasmesse al business layer, venivano manipolate fino a produrre il risultato ottenuto; infine veniva creato l'intero documento HTML da restituire al client. Quest'ultimo quindi era limitato

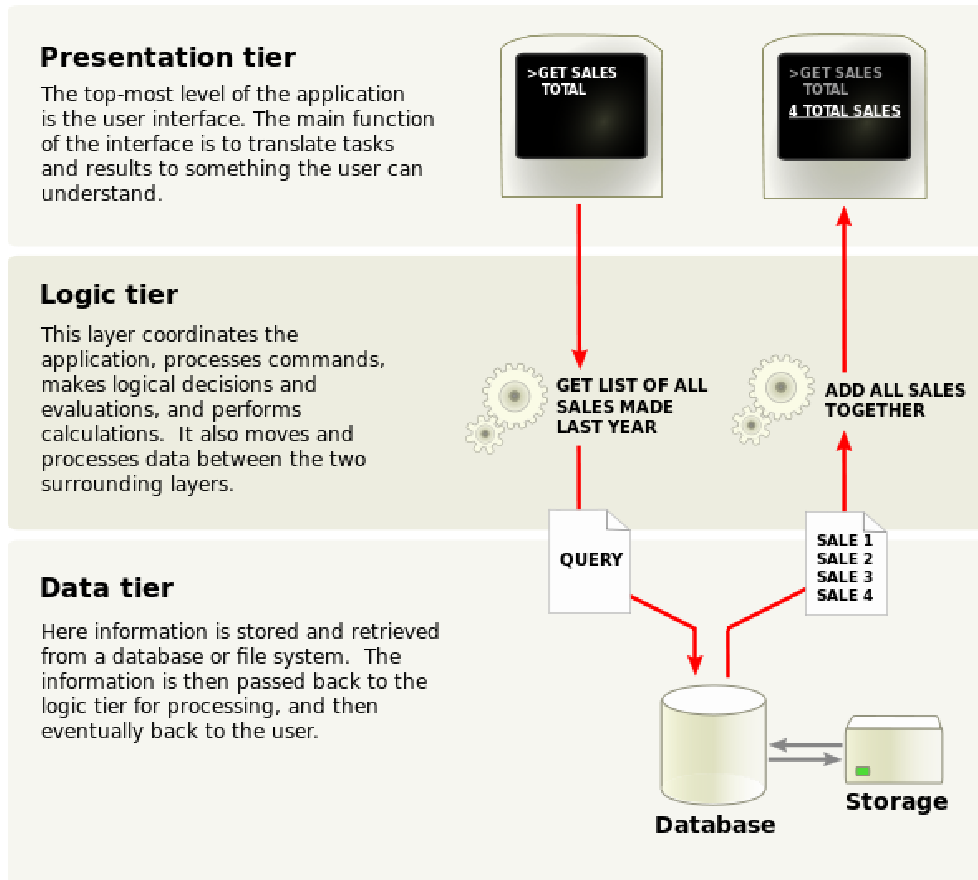

Figura 2.2: Esempio di architettura a 3 livelli [34].

a semplice strumento per l'invio di input e ricezione di output. Classica realizzazione di questo tipo di architettura erano i sistemi basati costruiti su piattaforma LAMP, formato dai seguenti componenti:

1. Linux. Il sistema operativo su cui il server veniva lanciato.
2. Apache. Il server Web, in grado di accettare request HTTP alla porta 80 ed inviare response HTTP.
3. MySQL. Il database relazionale utilizzato per la memorizzazione dei dati persistenti.

4. PHP, Python, o Perl. I linguaggi di scripting normalmente utilizzati.

Il database MySQL rappresentava il data layer; le pagine PHP contenevano la logica applicativa, accedevano al data layer eseguendo query al database e producevano in output l'intera pagina HTML (presentation layer), che veniva inviata al client come response. Il trend nelle moderne Web App spinge invece verso client sempre più complessi, che contengono il presentation layer e molta (se non tutta) la logica applicativa. In questo modo, sebbene il tempo necessario al loro sviluppo sia maggiore, l'efficienza percepita dall'utente e la scalabilità del server risultano maggiori.

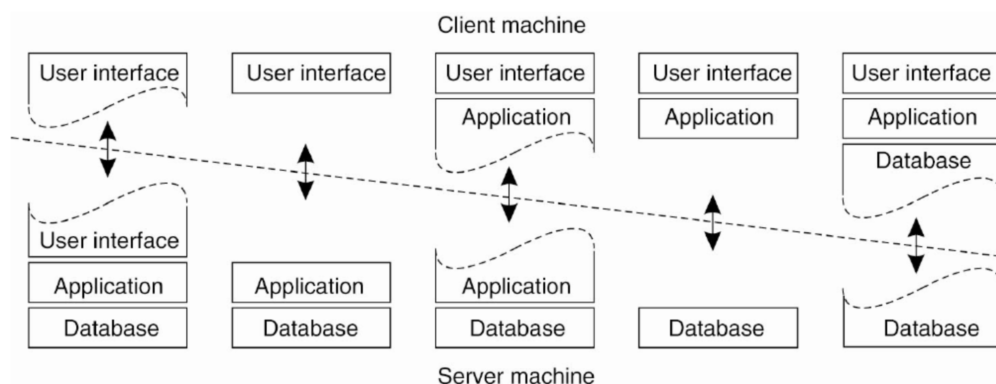

Figura 2.3: Diverse organizzazioni dei layer tra client e server [28].

2.1.2 Client architecture

Come suddividere fisicamente i layer presentati nella sezione precedente rappresenta una scelta progettuale e architetturale importante. Le applicazione web del passato, document-oriented, erano caratterizzate da una struttura di questo tipo: tutti i layer erano implementati nel server. Il client, attraverso delle request, richiedevano una particolare risorsa che gli veniva restituita come documento HTML pronto per la visualizzazione. I dati, la logica applicativa (se presente) e la formattazione della pagina era interamente a carico del server (figura 2.1.2). Una architettura di questo tipo risulta inapplicabile alle applicazioni web moderne.

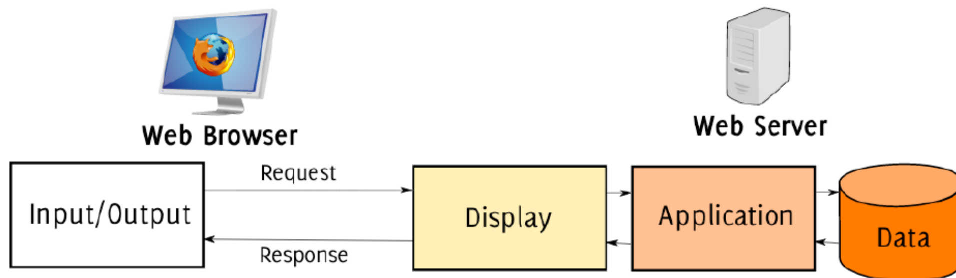

Figura 2.4: Organizzazione dei layer nel web document-oriented [23].

L'evoluzione dei browser e l'utilizzo di Javascript permette di delegare al client parte (o tutta) la logica applicativa. In un architettura di questo tipo il client, sotto forma di codice Javascript, è in grado di svolgere parte della computazione necessaria al funzionamento dell'applicazione. L'URL non rappresenta più l'identificatore di un documento, ma diventa il punto di accesso della applicazione. Il browser, accedendo a un URL, scarica il codice Javascript necessario (code shipping) che sarà in esecuzione localmente. Ciò consente a ogni client di memorizzare lo stato corrente della sua istanza di applicazione. Inoltre, attraverso tecnologie come Ajax, è possibile effettuare richieste asincrone al server, senza dover ricaricare la pagina, per inviare o ricevere nuove informazioni, e aggiornare il contenuto visualizzato dinamicamente. Architetture di questo tipo vengono definite Rich Client.

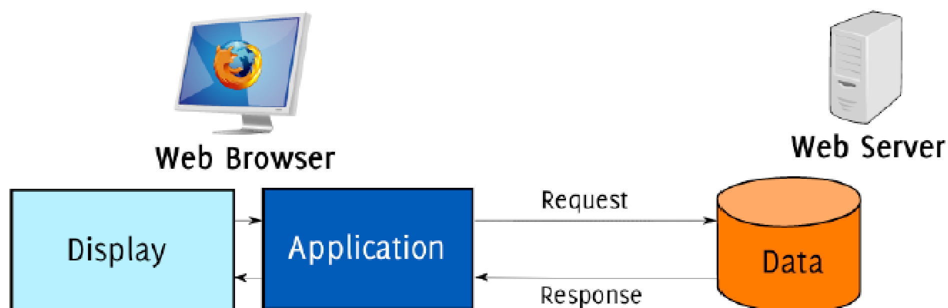

Figura 2.5: Architettura di una applicazione web Rich Client [23].

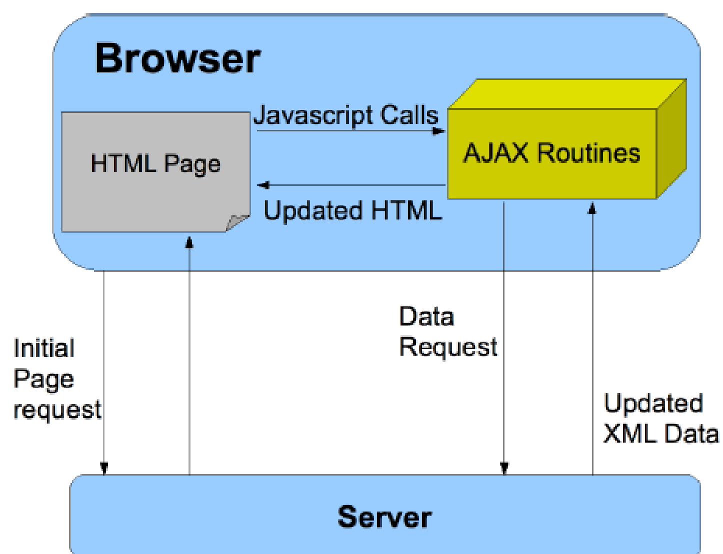

Figura 2.6: Interazione client-server in una Web App con Ajax [22].

Web Application e Web Services. Una tecnologia che ben si integra con questo stile architetturale è quella dei Web Service. Un Web Service è un componente computazionale autonomo che incapsula una determinata funzionalità ed è in grado di fornirla attraverso la rete. La comunicazione con questo tipo di entità si basa su scambio di messaggi in un formato indipendente dalla piattaforma su cui si esegue, secondo due possibili architetture: ReSTful (aderente ai principi di ReST) o SOA (Service Oriented Architecture). Le applicazioni web in cui la business logic è allocata sul client possono interagire con questi servizi per utilizzarne le funzionalità all'interno della propria logica applicativa. In applicazioni di questo tipo, la memorizzazione di dati persistenti può essere effettuata dal client attraverso un sistema di cache o tramite tecnologie più recenti come Web Storage.

I vantaggi di architetture Rich Client comprendono:

1. Riduzione del carico di lavoro del server. Poiché la logica applicativa e l'MVC sono allocati sul client, la computazione richiesta dal server è minima. Inoltre, sulla rete vengono inviate unicamente le risorse

necessarie, ottenendo un risparmio di banda.

2. Ampio utilizzo delle cache. Il client può memorizzare tutte le informazioni di cui ha bisogno sfruttando le risorse del sistema su cui è in esecuzione.

3. Rich UI. Grazie a framework come JQuery e WebGL, le interfacce utente possono raggiungere un livello di complessità e interattività al pari delle applicazioni desktop.

Sono presenti però anche alcuni svantaggi:

1. Contenuto non indicizzabile. Poiché ogni pagina viene riempita dinamicamente e non contiene di per sé informazioni, l'indicizzazione del contenuto da parte dei crawler come GoogleBot è difficilmente attuabile.

2. Necessità di un browser moderno. Le più recenti tecnologie non sono ancora supportate da tutti i browser e alcuni utenti potrebbero non avere l'ultima versione disponibile del proprio browser.

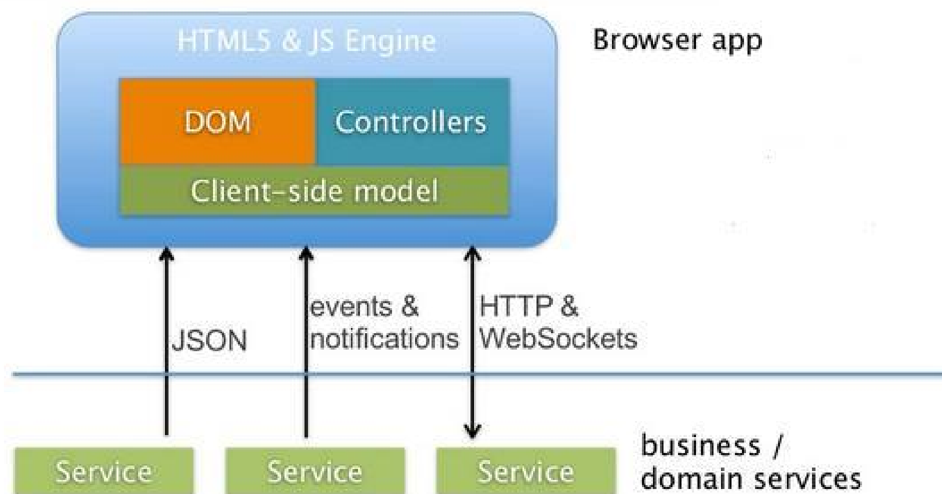

Figura 2.7: Architettura di una Web App con Web Services [3].

Usualmente, il client veniva comunemente definito front-end, mentre il server back-end. Per front-end veniva inteso il componente software che gestiva l'interazione con l'utente, mentre per back-end quell'insieme di componenti (logica applicativa e dati) che formavano l'applicazione. Nelle Web App moderne, questa associazione non è più cosi netta.

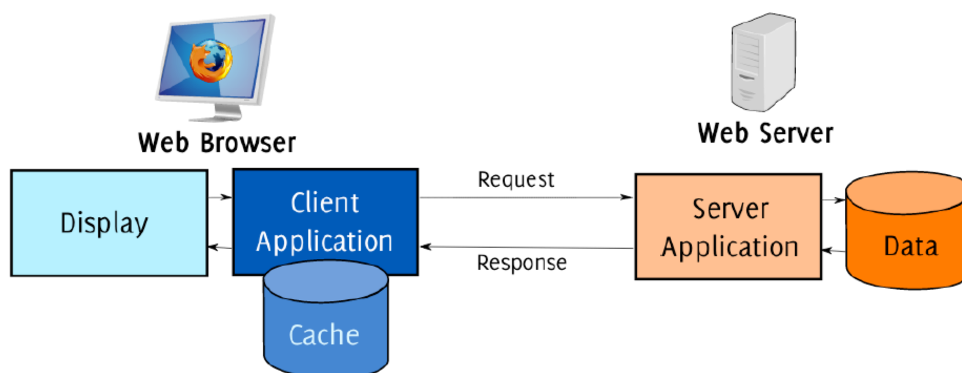

Figura 2.8: Architettura generale di una moderna Web App [23].

Client Application e Javascript. La necessità di sviluppare la logica applicativa nel client, attraverso l'utilizzo di Javascript, ha messo in luce le limitazioni di questo linguaggio. Le motivazioni che hanno portato alla nascita di Javascript alla fine del secolo scorso non comprendevano infatti lo sviluppo di applicazioni complesse e strutturate. Il linguaggio non prevedeva interfacce, ed essendo debolmente orientato agli oggetti concetti come l'ereditarietà non erano pienamente supportati. Non erano inoltre presenti framework o plugin che semplificassero il lavoro agli sviluppatori. Col tempo parte di questi problemi sono stati risolti; gli strumenti oggi a disposizione per uno sviluppatore web sono:

1. Librerie che semplificano la manipolazione di HTML, la gestione degli eventi, aspetti grafici e chiamate Ajax, e che garantiscono compatibilità cross-browser, come JQuery.

2. Framework per la strutturazione del codice secondo pattern architetturali come MVC utilizzando un approccio più dichiarativo che imperativo (es: AngularJS).

3. Linguaggi che si compilano in Javascript, molto simili ai linguaggi tradizionali come Java o Ruby, che sfruttano un maggior potere semantico e sintattico rispetto a Javascript. Questi linguaggi non richiedono l'installazione di plugin esterni poiché la loro compilazione genera normale codice Javascript (es: Dart, Typescript).

Single page Architecture. Un grosso vantaggio che deriva dallo spostamento della logica applicativa sul client è la riduzione del numero di request effettuate al server. In una applicazione tradizionale, ogni interazione dell'utente risulta in una nuova request al server e nel caricamento della nuova pagina intera. Nelle moderne Web App ciò non accade: le richieste sono gestite da handler Javascript, che manipolano l'albero DOM della pagina già visualizzata per inserire i nuovi contenuti. Questa assenza di refresh porta a una migliore user experience e a un minor numero di request al server. Applicazioni di questo tipo vengono definite a Single page Architecture [25].

2.1.3 Service Oriented Front-End Architecture

SOA è uno stile architetturale ideato con lo scopo di raggiungere l'interoperabilità tra applicazioni diverse situate sulla stessa macchina o attraverso la rete, utilizzando la logica di servizio riusabile. In SOA, applicazioni complesse sono realizzate attraverso l'interazione di entità più semplici chiamate servizi, utilizzabili anche da più sistemi contemporaneamente. Ogni servizio fornisce agli utilizzatori una specifica funzionalità ed è in grado di utilizzare quelle offerte da altri servizi riuscendo ad eseguire, in questo modo, operazioni anche molto complesse.

In questo contesto si inserisce Service Oriented Front End Architecture (SOFEA), uno stile architetturale che definisce come progettare quei componenti al di sopra del layer dei servizi. La diffusione dei servizi di tipo SOA degli ultimi anni ha reso questa architetturale particolarmente interessante. I principi fondamentali su cui si basa SOFEA sono:

1. Il download dell'applicazione, lo scambio di dati e la presentazione devono essere disaccoppiati. Nessun componente del client deve essere generato o invocato dal server.

2. Il flusso di presentazione deve essere a carico del solo client.

3. Tutte le comunicazioni con il server devono avvenire tramite servizi (SOA o eventualmente ReST).

4. Deve essere applicato il design pattern Model View Controller (MVC) e deve essere allocato interamente sul client.

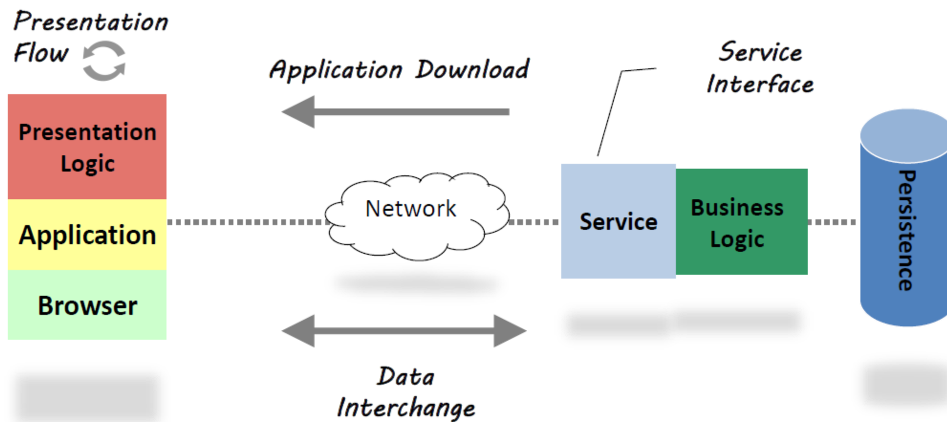

Figura 2.9: Struttura di una applicazione web secondo SOFEA.

I benefici dell'applicazione di SOFEA sono:

1. **Scalabilità.** L'unico compito del server è quello di fornire dei servizi. La presentazione è a carico del client.

2. **Minori tempi di latenza.** Dopo il download iniziale di tutto il codice applicativo, gli scambi attraverso la rete sono unicamente di dati. Minore banda consumata e maggiori performance percepite dall'utente.

3. **Integrazione con SOA.** La natura stessa dell'architettura ne favorisce l'integrazione con i Web Services SOA già esistenti.

4. **Separation of concerns.** Il client si occupa della visualizzazione, il server di fornire dei servizi, in modo indipendente l'uno rispetto all'altro.

5. **Offline.** Il client può continuare l'esecuzione anche in caso di assenza di connessione grazie a cache e storage locale.

6. **Interoperabilità.** I servizi possono essere sviluppati in qualsiasi linguaggio, a patto che l'interfaccia con cui sono offerti sia costante. Variazioni dell'implementazione non ha ripercussioni sul client.

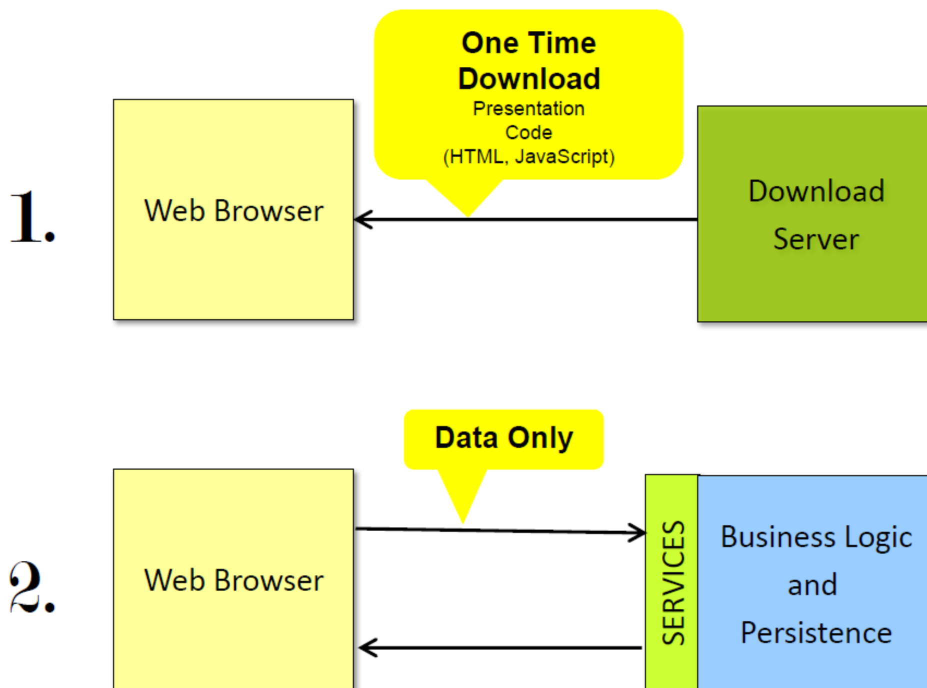

Figura 2.10: Ciclo di vita di una applicazione SOFEA.

2.2 Pattern di progettazione

I design pattern descrivono come oggetti e classi devono essere strutturati e come devono comunicare per risolvere un problema generale di progettazione in uno specifico contesto [5]. Un design pattern rappresenta una soluzione generale a un problema di progettazione ricorrente, gli attribuisce un nome, astrae e identifica gli aspetti principali della struttura utilizzata per la soluzione del problema, identifica le classi e le istanze partecipanti e la distribuzione delle responsabilità, descrive quando e come può essere applicato [32]. I pattern sono classificati secondo diverse tipologie; le principali sono:

1. **Pattern strutturali.** Esprimono l'organizzazione strutturale e comportamentale di un sistema software. Definiscono un insieme predefinito di sottosistemi, specificando le loro responsabilità e le loro relazioni. Composite e Decorator fanno parte di questa categoria.

2. **Pattern comportamentali.** Forniscono soluzioni alle più comuni tipologie di interazione tra componenti. Il più famoso è l'Observer-Observable.

3. **Pattern creazionali.** Utili per poter utilizzare un oggetto senza conoscere come è stato implementato. Forniscono dei metodi specifici, diversi dai costruttori, per la creazione di oggetti. Es: Factory e Abstract Factory.

Nell'ingegneria del software tradizionale, la definizione dei pattern e il loro utilizzo si applicava al contesto del software per desktop. L'evoluzione del web e le tecnologie attuali consentono lo sviluppo di applicazioni large-scale complesse, che richiedono una attenta fase di progettazione. In questo scenario, l'ingegneria del web ha assimilato ciò che già esisteva nell'ingegneria del software tradizionale, adattandolo al contesto delle applicazioni web. Rispetto alle classificazione iniziale dei pattern espressa in precedenza, una nuova tipologia di pattern ha avuto maggiore diffusione: i pattern architetturali. I pattern architetturali si differenziano dai pattern strutturali poiché riguardano l'intera organizzazione del sistema software, e non sono ristretti a un particolare insieme di componenti. Il pattern architetturale più utilizzato nelle applicazioni web moderne è il Model-View-Controller (MVC).

2.2.1 Model-View-Controller

Il Model-View-Controller (MVC) nacque negli anni '70 per il linguaggio SmallTalk-80. Col tempo, MVC è diventato il pattern di riferimento per la gestione dell'interfaccia utente nei linguaggi object-oriented. L'idea principale alla base di MVC è l'applicazione del principio di separation of concerns. In una applicazione web, questo pattern viene utilizzato principalmente, lato client, per l'organizzazione dei layer di presentazione e di business logic. In un sistema MVC, i componenti si suddividono in tre gruppi:

1. Il modello
2. La vista
3. Il controller

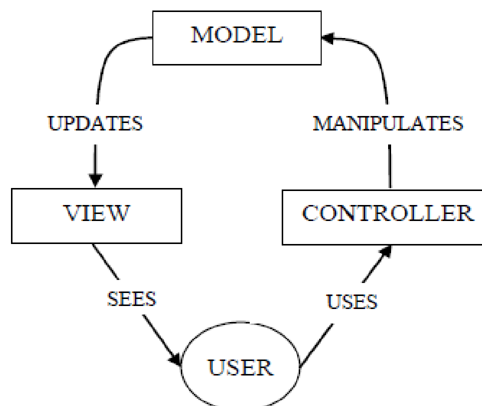

Figura 2.11: Visione semplificata del Model-View-Controller [21].

Il modello. Il modello consiste in quei componenti, statici o dinamici, che forniscono una rappresentazione della logica applicativa. Lavorando sul modello, il progettista deve far sì che esso sia indipendente dalla vista scelta e dalle tecnologie usate per catturare le azioni dell'utente. A ogni modello sono associate una o più viste e uno o più controller. Quando lo stato del modello cambia, una notifica viene inviata ai componenti associati, solitamente attraverso un meccanismo di Observer-Observable. In seguito a tali

notifiche, la vista modificherà l'interfaccia utente in output, mentre il controller aggiornerà la lista di comandi accettabili. Il modello può essere più o meno attivo a seconda del particolare scenario applicativo: alcune notifiche possono anche non essere presenti.

La vista. La vista è responsabile della presentazione grafica del modello. La vista ha un forte accoppiamento con il modello, perché deve conoscere le specifiche dei dati e delle operazioni supportate: a ogni notifica ricevuta, essa deve ottenere il nuovo stato del modello attraverso delle query, e aggiornare la propria presentazione. Viceversa, modifiche alla vista e alla sua implementazione non devono avere ripercussioni sul modello.

Il controller. Il controller deve intercettare operazioni dell'utente (come click del mouse) e mapparle in operazioni sul modello (cambiamento dello stato) o cambiamenti alla vista. Assieme alla vista, fornisce il look and feel dell'applicazione. Un errore comune nella comprensione del controller consiste nell'integrare logica applicativa all'interno del controller stesso; questo porterebbe a un accoppiamento tra modello e controller: cambiamenti dell'uno si ripercuoterebbero sull'altro. Per questo motivo, il controller deve contenere solo riferimenti alla logica effettivamente implementata nel modello.

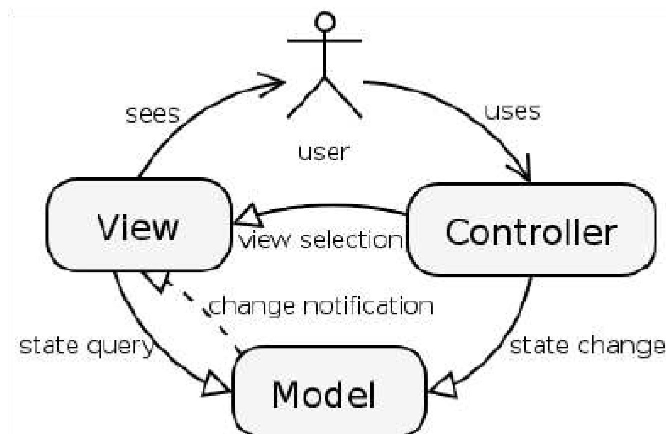

Figura 2.12: Interazioni tra componenti in Model-View-Controller [19].

Nelle moderne applicazioni web, il client ha la possibilità di mantenere una cache locale o addirittura un piccolo database (es: indexedDB). Questo, unitamente alla presenza di business logica, consente all'applicazione l'esecuzione offline. Per questo motivo, l'interazione del modello con i restanti componenti assume caratteristiche diverse. Nelle applicazioni desktop classiche, i tre componenti del MVC (modello, vista e controller) sono allocati sulla stessa macchina, e la comunicazione tra essi è perciò sempre possibile. Nelle Web App che supportano l'esecuzione offline, può succedere che aggiornamenti allo stato del modello comportino cambiamenti nel database allocato sul server. Se la connessione non è attiva, e l'applicazione è in esecuzione in modalità offline, memorizzare questi cambiamenti nel database risulta impossibile. Per questo motivo è necessario che il client mantenga una copia del modello locale e la sincronizzi con il modello effettivo del server durante l'esecuzione online. Il server, poiché possono arrivare sincronizzazioni da diversi client contemporaneamente, deve poter gestire concorrentemente ogni connessione e garantire al modello finale consistenza.

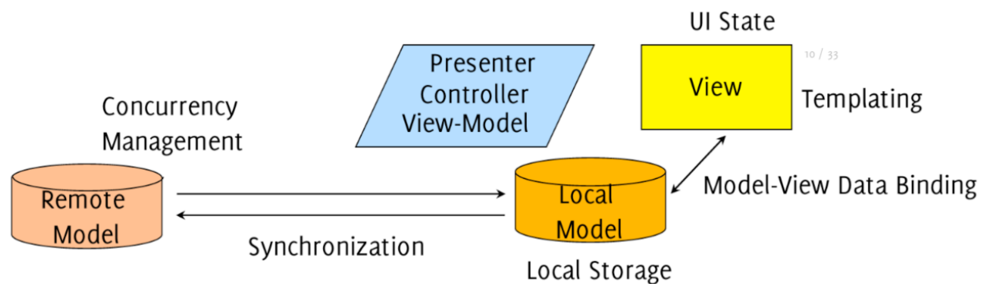

Figura 2.13: Modello locale e MVC in una Web App.

Capitolo 3

Meccanismi per la comunicazione

Uno degli aspetti fondamentali delle applicazioni web è l'interazione tra uno o più utenti e uno o più server. Dal semplice request/response di HTTP, l'evoluzione del web e delle sue tecnologie ha portato alla nascita di nuove tecnologie per migliorare sempre di più la comunicazione remota. Il modello tradizionale di comunicazione, derivato dalle specifiche standard di HTTP, prevedeva una comunicazione sincrona: in seguito a una azione dell'utente (request), il server eseguiva l'operazione richiesta e restituiva il risultato (response). Dopo la richiesta iniziale, il client si poneva in uno stato di attesa fino a quando la risposta non era ricevuta, risultando in uno spreco di tempo e risorse. Il refresh della pagina peggiorava inoltre la user-experience. Allo stesso tempo, il server non manteneva nessuna informazione riguardo alla comunicazione appena avvenuta. Più richieste della stessa operazione venivano ogni volta re-processate e rigenerate per ogni client che le richiedeva. Una comunicazione di questo tipo, semplice da effettuare dal punto di vista implementativo, risulta tuttavia inefficiente e inadatta ad applicazioni di larga scala moderne. Il primo fondamentale passo è stato rendere la comunicazione da sincrona ad asincrona. Ciò è stato possibile attraverso l'uso di plugin esterni, come Flash, oppure tramite nuovi meccanismi come Ajax. Tuttavia, entrambi presentano dei problemi: nel primo caso, un utente poteva non aver intenzione di installare software esterno, rendendo così inutile il plugin; nel secondo caso, la gestione stessa della comunicazione attraverso Javascript poteva velocemente raggiungere livelli di complessità

non accettabili per grandi applicazioni. In questo capitolo verranno trattati i meccanismi di comunicazione precedenti ad HTML5 più diffusi, Ajax e Flash Socket, e la nuova tecnologia associata ad HTML5: i Web Socket. Questi ultimi rappresentano un passo avanti non indifferente per quanto riguarda il futuro delle applicazioni web.

3.1 Comunicazione pre-HTML5

I meccanismi in grado di realizzare una comunicazione asincrona sul web hanno segnato l'inizio di una nuova era. Nati nella prima metà degli anni 2000, hanno contribuito a trasformare il web da document-oriented ad application-oriented. Poiché qualunque tecnologia per la comunicazione sul web deve affidarsi, al livello più basso, al protocollo HTTP, che costituisce il fondamento della rete, particolare importanza hanno assunto quei tipi di comunicazione definiti server push. Il protocollo HTTP prevede infatti che la comunicazione cominci sempre dal client attraverso una request. Ciò deriva dallo visione originale del web, come strumento per la condivisione di documenti disponibili su richiesta. Il caso in cui il server, dopo una iniziale connessione del client, voglia per primo inviargli dei dati (push) senza aver ricevuto nessuna richiesta, risulta di difficile attuazione. Con il tempo, diverse soluzioni sono state proposte per superare questo ostacolo:

1. **HTTP Server Push.** Questo meccanismo, conosciuto anche come HTTP Streaming, è stato uno dei primi a essere realizzato. Veniva attuato dalla maggior parte dei web server attraverso specifiche CGI (Common Gateway Interface). Ad esempio, mediante i Non-Parsed Headers scripts di Apache. Il server, dopo una richiesta, memorizzava l'entità del client. Quando occorreva effettuare una push verso quel client, i dati da inviare venivano memorizzati in una coda, associando l'identificativo del client. A una successiva request di quel client, il server aggiungeva nella response anche i dati precedentemente salvati eliminandoli dalla coda.

2. **Multipart replace.** Introdotto da Netscape nel 1995, questo tipo di MIME può essere applicato a una response del server. Tutte le parti di messaggio definito come mixed-replace hanno lo stesso significato semantico; tuttavia ogni parte sovrascrive le precedenti. Il client processa ogni singola parte individualmente non appena viene ricevuta,

senza aspettare il completamento dell'intero messaggio. Attraverso questo meccanismo è possibile simulare una server push.

3. **Pushlet.** La tenica delle pushlet consiste nel mantenere la connessione tra server e client persistente senza terminare mai la response del server. Questo è possibile specificando come header del messaggio HTTP "Connection: Keep-Alive" (in HTTP 1.1 questo è stato settato di default). Il server può inviare perciò pacchetti di Javascript periodicamente, poiché il browser non considera la pagina caricata. Questo metodo presenta però degli svantaggi: il server non può avere nessuna notifica del corretto funzionamento del client, e in caso di migliaia di client connessi, il server potrebbe trovarsi in carenza di risorse a causa di tutte le connessioni aperte (scarsa scalabilità).

4. **Long Polling.** Il long polling non è una strategia di push vera e propria, ma può essere usata per emularlo quando non è possibile effettuare un push vero e proprio. In un long polling, il client richiede al server informazioni come in un normale polling, ma qualora esse non siano disponibili sul momento, invece che restituire una risposta vuota, il server attende la loro disponibilità. Quando questo avviene, il server risponde alla richiesta inviandole. Questo metodo elimina l'intervallo di tempo che trascorre in un normale polling tra l'istante in cui le informazioni diventano disponibili sul server e l'istante in cui vengono richieste dal client. Inoltre, la frequenza con cui effettuare il polling diminuisce.

Nella sezione successiva verrà approfondito il meccanismo di comunicazione asincrono Ajax con riferimento al long polling come metodo per il push. Successivamente verrà analizzata un'altra soluzione, i Flash Socket, che si basa sull'utilizzo di plugin esterni.

3.1.1 Ajax e Long Polling

AJAX (Asynchronous JavaScript And XML) è lo strumento che per primo ha permesso l'evoluzione del web nella direzione delle applicazioni moderne Rich User. Ajax non è in realtà una tecnologie a sé: si basa sull'utilizzo di Javascript e XML per permettere la comunicazione asincrona tra client e server senza la necessità di software di terze parti (plugin). Il componente fondamentali di Ajax è l'XMLHttpRequest. Questo oggetto consente di

effettuare al server richieste HTTP di una risorsa indipendentemente dallo stato del browser, e anche di inviare informazioni attraverso i metodi GET o POST. Essendo richieste asincrone, il client può effettuare altre operazioni durante il tempo di attesa della risposta. Possono inoltre essere inviate più richieste in serie, senza dover attendere la risposta di ognuna. La risposta del server, una volta ricevuta, verrà gestita da una particolare funzione, chiamata callback. Proprio questo aspetto rende difficile l'applicazione di Ajax ad applicazioni di larga scala: il codice Javascript potrebbe velocemente diventare una serie lunghissima di callback. Qualora fosse necessaria una forma di sincronizzazione tra le varie richieste, questo diventerebbe complicato.

L'oggetto Javascript XMLHttpRequest supporta diversi metodi:

1. **open.** Open è il metodo di apertura di una chiamata Ajax. Deve sempre essere eseguito.

2. **send.** Utilizzato per effettuare fisicamente la richiesta, sia GET o POST, al server.

3. **setRequestHeader**. Setta gli header specificati della request col valore desiderato.

4. **getAllResponseHeaders e getResponseHeader .** Ottengono gli header inviati dal server nel messaggio di response.

5. **abort.** Questo metodo termina immediatamente le operazioni di invio o ricezione in esecuzione.

Di seguito verrà analizzata la sintassi e il funzionamento di questi metodi.

open. Il metodo open deve essere eseguito all'inizio di ogni richiesta, e accetta cinque parametri. La sintassi è:

```
// sintassi del metodo open
open (method, uri [,async][,user][,password])
```

Il primo parametro può essere "get" o "post" e identifica il metodo della request HTTP. Scegliendo GET, le variabili da inviare saranno appese

all'URL (es: pagina.php?variabile=valore), mentre scegliendo POST verranno inserite nel corpo della richiesta. Il metodo GET è più immediato da utilizzare, ma poiché il numero di caratteri dell'URL è limitato (solitamente a 256), non è sempre consigliabile. POST prevede un limite di 8 megabyte, che in genere costituisce un problema solo durante l'invio di file. Il secondo parametro è l'URL della pagina, che può essere assoluto o relativo. Se si vuole utilizzare il metodo GET, è necessario inserire nell'URL anche le variabili da trasmettere. Il terzo parametro, dato che Ajax viene usato per chiamate asincrone, deve essere impostato come "true. Gli ultimi due parametri, opzionali, devono essere inseriti quando è necessaria l'autenticazione.

```
// esempio di apertura di una chiamata asincrona con GET
ajax.open("get", "path/resource.php?variable=value", true);

// esempio di apertura di una chiamata asincrona con POST
ajax.open("post", "path/resource.php", true);
```

send. Il metodo send inoltra realmente la richiesta. La sintassi è:

```
// sintassi del metodo open
send(data)
```

In una chiamata GET, non è necessario specificare alcun dato (null), mentre per una chiamata POST sono necessari alcuni accorgimenti. Occorre infatti assegnare, attraverso gli header, il content-type della richiesta come "application/x-www-form-urlencoded". Ciò è possibile utilizzando il metodo setRequestHeader. I dati da inviare devono essere specificati come parametro di send nel formato chiave=valore, separati da una "&".

Dalla versione di HTTP 1.1, la connessione è impostata di default come keep-alive. Sia per GET che per POST, per chiudere la connessione al termine della request/response occorre settare l'header "connection" come "close" prima del send. Se un valore contiene"&" o altri caratteri particolari, potrebbe essere interpretato come separatore invece che come effettivo carattere. Per superare questo problema è necessario utilizzare la funzione escape(), che restituisce un output valido per la stringa passata come

parametro.

```
// esempio di una chiamata POST
ajax.open("post", "path/resource.php", true);

ajax.setRequestHeader("content-type",
   "application/x-www-form-urlencoded");

ajax.setRequestHeader("connection", "close");

var params = "variable1=" + escape("1&2");
params += "&";
params += "variable2=" + escape("l'apostrofo");

ajax.send(params);
```

Dopo l'invio della request, una callback deve essere eseguita alla ricezione della response. Per fare questo si utilizzano i parametri dell'oggetto XMLHttpRequest, che sono:

1. **onreadystatechange.** L'unico parametro non di sola lettura, serve a specificare la funzione di callback da eseguire.

2. **readyState.** Un intero che rappresenta lo stato corrente della richiesta.

3. **responseText e responseXML.** Contengono i dati ricevuti nella response. Il primo in formato di stringa, il secondo in formato XML. Se la response non rappresenta un file XML, responseXML è null.

4. **status.** Contiene lo status code della response ricevuta.

5. **statusText.** Contiene il messaggio associato allo status code.

readystate. Istante per istante, lo stato della richiesta può essere controllato leggendo il valore di questo parametro. Può avere cinque diversi valori:

- 0 uninitialized: l'oggetto XMLHttpRequest è stato creato ma non non è ancora stato eseguito il metodo di apertura.

- 1 open: è stato eseguito il metodo open ma non send.

- 2 sent: sent è stato eseguito e la richiesta è partita.
- 3 receiving: la risposta è stata ricevuta ed è in lettura.
- 4 loaded: la risposta è pronta e l'operazione è stata completata.

Raggiunto lo stato 4, esso non cambierà più fino alla ricezione di successivi comandi. Durante lo stato di receiving è possibile leggere alcune informazioni dagli header come la lunghezza ("content-length") e il tipo del testo ricevuto attraverso i metodi response.

onreadystatechange. Questo parametro consente di assegnare all'oggetto XMLHttpRequest una funzione. Ogni volta che lo stato dell'oggetto cambia, questa funzione di callback, se specificata, verrà eseguita. Il momento più sicuro per l'assegnazione della funzione è dopo l'open e prima del send.

```
// Esempio di callback
ajax.onreadystatechange = function() {
   if(ajax.readyState == 4) {
      if(ajax.status == 200) {
         alert("Successo");
         elemento.innerHTML = ajax.responseText;
      } else {
         alert("Errore: " + ajax.status);
      }
   }
};
```

All'interno di questa funzione, come mostrato nell'esempio, può essere utile leggere il valore dello stato e, nel caso di stato 4, cioè loaded, verificare l'esito dell'operazione. Se l'operazione ha avuto successo si possono usare i metodi responseText e responseXML per ottenere il testo della risposta.

Timeout. Quando il server a cui è diretta la request è sovraccarico o in caso di eccessive latenze della rete, la richiesta può andare persa. Poiché il server non ha inviato nessuna risposta, non è possibile gestire questa situazione attraverso la lettura dello status code. Inoltre, in assenza di notifiche all'utente, questo sarà portato ad eseguire più volte l'operazione, peggiorando la situazione. Una delle soluzioni per superare questo problema è

controllare il tempo trascorso attraverso un timeout. Allo scadere, l'utente sarà notificato del fallimento dell'operazione. L'esempio seguente mostra una possibile implementazione.

```
var timeout = 5000; // in millisecondi
var checkTimeout; // funzione di timeout
var startTime = new Date().getTime();

ajax.onreadystatechange = function() {

   // Se l'operazione è stata completata
   if(ajax.readyState === 4) {
      // Questa funzione non serve più
      checkTimeout = function(){};
      if(ajax.status == 200)
            alert("Successo");
            elemento.innerHTML = ajax.responseText;
      else
         alert("Errore: " + ajax.status);

   } else {

      checkTimeout = function() {

         // Se l'operazione è andata in timeout
         if((new Date().getTime() - startTime) > timeout) {

            // Rimuovere la callback per evitare che venga eseguita
                all'abort
            ajax.onreadystatechange = function(){return;};
            ajax.abort();
            alert("Timeout");

         // Se non è ancora in timeout, si rieffettua il controllo
             dopo un intervallo di tempo ritenuto opportuno
         } else {
            setTimeout(checkTimeout, 100);
         }
   };
```

```
    checkTimeout();
};

ajax.send(null);
```

L'esempio mostra l'utilizzo di una funzione, richiamata in modo asincrono, con lo scopo di verificare lo stato del timeout. Al primo onreadystatechange, quello causato dall'invio della richiesta, viene eseguita la callback, che non trovando una risposta, esegue la funzione checkTimeout. Questa funzione controlla se il timeout è trascorso. In caso affermativo, interrompe la richiesta e notifica l'utente. In caso negativo, stabilisce che dopo un certo intervallo di tempo, da scegliere accuratamente, deve essere richiamata se stessa per effettuare un nuovo controllo. Se nel frattempo la risposta del server arriva, il riferimento alla funzione checkTimeout viene eliminato interrompendone perciò il ciclo di attesa.

Long polling. Le moderne applicazioni richiedono però che la comunicazione non segua solamente il paradigma request/response, ma che sia possibile anche effettuare push di informazioni dal server. Dal punto di vista del client, una chiamata Ajax di long polling non è diversa da una chiamata Ajax normale. La differenza è che il server, se non ha i dati richiesti a disposizione, non restituisce una risposta vuota ma attende che essi siano disponibili e poi li invia. In questo modo sarà sempre presente un canale di comunicazione attivo, che può essere utilizzato istantaneamente dal server. Un approccio di questo tipo non è tuttavia realizzabile in tutti i server web. Molti server infatti hanno a disposizione un numero limitato di richieste massime accettabili, terminate le quali sono costretti a rilasciarne qualcuna.

Un framework utilizzabile per realizzare applicazioni web server in grado di effettuare long polling è Node.js. Come enunciato nel capitolo 1, attraverso Node.js è possibile utilizzare Javascript anche per l'implementazione lato server dell'applicazione. Node.js, basandosi sul modello event-driven a callback di Javascript, ha delle strutture di basso livello capaci di gestire migliaia di connessioni aperte da client diversi.

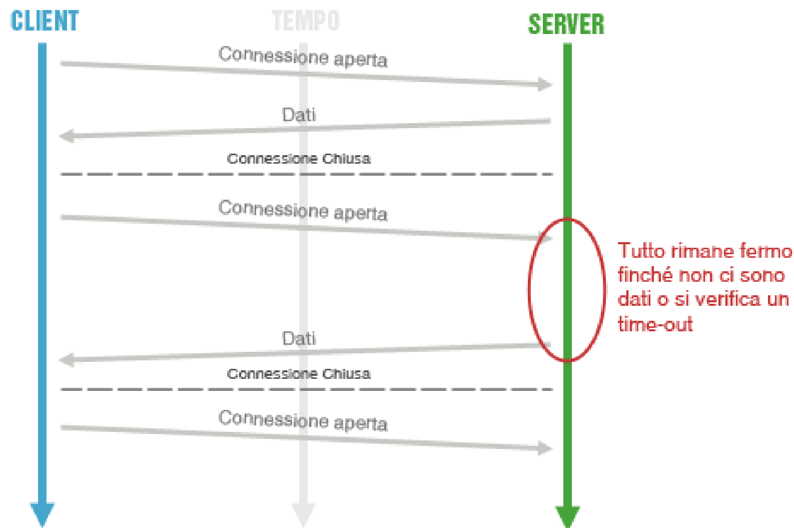

Figura 3.1: Modello di una comunicazione con long polling [14].

3.1.2 Adobe Flash Socket

Flash è un software sviluppato da Adobe che permette la creazione di animazioni e grafica vettoriale per il web. Col tempo Flash si è evoluto diventato un vero e proprio strumento per la creazione di Rich User Interface. Un particolare componente di Flash, i Flash Socket, possono essere usati come meccanismo di comunicazione tra client e server. I socket sono canali di rete attraverso cui due processi in esecuzione su computer diversi possono comunicare. I Flash Socket utilizzati dal Flash Player del browser agiscono come client-socket TCP. Ciò significa che possono connettersi ad un socket server, ma non possono essi stessi funzionare come server. Per utilizzare i Flash Socket, sono disponibili API per ActionScript, il linguaggio di scripting di Flash. Attraverso l'ambiente di sviluppo Adobe Integrated Runtime (AIR), è possibile estendere la funzionalità di base dei Flash Socket, per creare ad esempio un server socket.

Le seguenti API possono essere utilizzate per effettuare comunicazioni TCP:

1. Socket. Permette al client di connettersi ad un server. Non è possibile

accettare connessioni in ingresso.

2. SecureSocket (AIR). Simile ai socket, permette la connessione a server sicuri e comunicazione criptata.

3. ServerSocket (AIR). Abilita una applicazione ad agire come server.

4. XMLSocket. Permette al client di connettersi ad un server comunicando attraverso messaggi XML.

Di seguito verranno descritte le caratteristiche principali della classe Socket.

Socket. La classe Socket consente di creare connessioni TCP e leggere e scrivere dati in formato binario. Come per un normale socket, per la connessione è richiesto l'indirizzo IP del server e la porta. Degli handler sono associati come listener a particolari eventi, come la chiusura del socket o la presenza di errori. Una volta effettuata la connessione, sono disponibili specifici metodi per l'invio e la ricezione di dati.

```
// Creazione di un Flash Socket
socket = new Socket();

// Assegnazione degli event listener
socket.addEventListener(Event.CONNECT, connectHandler);
socket.addEventListener(Event.CLOSE, closeHandler);
socket.addEventListener(ErrorEvent.ERROR, errorHandler);
socket.addEventListener(IOErrorEvent.IO_ERROR, ioErrorHandler);
socket.addEventListener(ProgressEvent.SOCKET_DATA, dataHandler);

// Connessione ad un socket server
serverURL = "127.0.0.1";
portNumber = 5412;
socket.connect(serverURL, portNumber);

// Scrittura di dati sul socket
socket.writeBytes(ba);
socket.flush();

// Ricezione di dati dal socket
```

```
// byteAvailable è una proprietà della classe Socket, di sola
   lettura, che rappresenta il numero di byte di dati disponibili
   a essere letti.
str = readBytes(bytesAvailable);
```

Per una gestione agile delle server push, è possibile specificare che l'handler del evento ProgressEvent.SOCKET_DATA effettui la lettura del socket. In questo modo, se il server per primo invia dati al client, essi verranno letti istantaneamente.

```
// Esempio di dataHandler che effettua la lettura
private function dataHandler(event:ProgressEvent):void {
     var str:String = readBytes(bytesAvailable);
}
```

I socket sono uno strumento di comunicazione molto potente. Con essi è possibile effettuare comunicazione sincrona, asincrona e server push in modo semplice. Inoltre, differentemente dalle chiamate Ajax, i dati scambiati non devono seguire le specifiche di un messaggio HTTP (request/response GET o POST), ma possono essere semplici dati binari. Il principale problema di questa tecnologia tuttavia è la necessità dell'installazione del software Adobe per l'esecuzione.

3.2 HTML5: WebSocket

Tra le novità portate da HTML5, i WebSocket rappresentano quella di maggior importanza dal punto di vista dell'interazione tra client e server. WebSocket è una tecnologia per effettuare comunicazioni bidirezionali in tempo reale. Essi prevedono un canale di comunicazione sempre attivo, a bassa latenza, tra client e server, utilizzabile da entrambi sia in scrittura che in lettura. Tale canale è costituito da una connessione TCP persistente, garantito da un handshaking client-key iniziale ed un modello di sicurezza origin-based. Per la protezione dei dati trasmessi contro lo sniffing sono applicate apposite maschere. Ecco le caratteristiche principali dei WebSocket:

1. **Bidirezionali.** Quando il canale di comunicazione è attivo, sia il client che il server sono connessi ed entrambi possono inviare e ricevere messaggi.

2. **Full-duplex.** Dati inviati contemporaneamente dai due attori (client e server) non generano collisioni e vengono ricevuti correttamente.

3. **Basati su TCP.** Il protocollo usato a livello di rete per la comunicazione è il TCP, che garantisce un meccanismo affidabile (controllo degli errori, re-invio di pacchetti persi, ecc) per il trasporto di byte da una sorgente a una destinazione.

4. **Client-key handshake.** All'apertura di una connessione, il client invia al server una chiave segreta di 16 byte codificata con base64. Il server aggiunge a questa un'altra stringa, detta magic string, specificata nel protocollo ("258EAFA5-E914-47DA-95CA-C5AB0DC85B11"), codifica con SHA1 e invia il risultato al client. Cosi facendo, il client può verificare che l'identità del server che ha risposto corrisponda a quella desiderata.

5. **Sicurezza origin-based.** Alla richiesta di una nuova connessione, il server può identificare l'origine della richiesta come non autorizzata o non attendibile e rifiutarla.

6. **Maschera dei dati.** Nella trama iniziale di ogni messaggio, il client invia una maschera di 4 byte per l'offuscamento. Effettuando uno XOR bit a bit tra i dati trasmessi e la chiave è possibile ottenere il messaggio originale. Ciò è utile per evitare lo sniffing, cioè l'intercettazione di informazioni da parte di terze parti.

L'utilizzo dei WebSocket lato client è possibile attraverso l'uso di specifiche API Javascript che consentono di ottenere informazioni sullo stato della connessione (aperta, chiusa, in apertura o in chiusura), di interagire con essa (inviare dati o chiudere la comunicazione) o di gestire particolari eventi come la ricezione di errori. Lato server, esistono implementazioni dei WebSocket per la maggior parte dei linguaggi più utilizzati (Node.js, Java, C#, Python, Ruby).

3.2.1 Protocollo

Il protocollo dei WebSocket è stato redatto dal WHATWG e standardizzato dall'IETF nel Dicembre 2011. Suo obiettivo è quello di permettere una comunicazione HTTP bidirezionale nel contesto delle infrastrutture HTTP

già presenti; per questo motivo utilizza la porta 80 e supporta proxy e intermediari HTTP, anche se ciò implica una maggiore complessità. Il protocollo non esclude tuttavia future implementazioni diverse dall'HTTP (differenti handshake e porta). Esso è composto da due parti: l'apertura (handshake) e il trasferimento dei dati. Quando client e server hanno inviato il proprio handshake e l'operazione ha avuto successo, è possibile iniziare il trasferimento dei dati. Durante la comunicazione vengono trasferite unita logiche chiamate messaggi. Sulla rete, ogni messaggio è composto da uno o più frame, non necessariamente uguali ai frame del livello di trasporto. A ogni frame è associato un tipo, che può essere testuale, binario o di controllo. La versione attuale del protocollo definisce sei tipi di frame utilizzabili, e ne lascia dieci tipi per future implementazioni. I frame che fanno parte dello stesso messaggio sono caratterizzati dallo stesso tipo.

L'idea alle spalle dei WebSocket è che il numero di frame dovrebbe essere molto ridotto, ma sufficiente a rendere il protocollo frame-based invece che stream-based e distinguere tra trasmissione di dati (frame binari) o testo (frame UTF8). Qualsiasi tipo di metadato deve essere gestito a livello applicativo al di sopra dei WebSocket, come succede per HTTP su TCP. Concettualmente, WebSocket è quindi semplicemente un layer al di sopra di TCP in grado di:

1. aggiungere un modello di sicurezza origin-based per i browser.
2. aggiungere un meccanismo di addressing e protocol naming per supportare più servizi sulla stessa porta e più host sullo stesso indirizzo IP.
3. aggiungere un meccanismo di frame basato su TCP molto simile a quello usato dai pacchetti IP ma senza limiti di lunghezza.
4. aggiungere un handshake di chiusura in grado di funzionare in presenza di proxy o altri intermediari.

Sostanzialmente, WebSocket fornisce un meccanismo molto simile a puro TCP, considerati i vincoli imposti dal web. Esso rappresenta perciò un protocollo indipendente TCP-based, il cui handshake iniziale è interpretato dai server HTTP come un upgrade request. La porta utilizzata è la 80 per i WebSocket standard o la 443 per le comunicazioni Transport Layer Security (TLS).

3.2.2 Opening Handshake

La prima fase della comunicazione è composta dall'handshake di apertura. L'handshake è stato pensato per essere compatibile con server e intermediari HTTP-based, in modo tale che la stessa singola porta possa essere utilizzata per l'interazione col server sia da un client HTTP normale che da un client WebSocket. Per questo motivo, l'handshake da parte del client è una richiesta di upgrade HTTP:

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

L'URI richiesto tramite il metodo GET indica l'endpoint della connessione WebSocket. Ciò permette di fornire più WebSocket endpoint da parte dello stesso server. Il client specifica il server a cui connettersi attraverso l'header host; in questo modo il server può verificare l'effettiva identità richiesta. Un URI può essere definito in due modi:

- ws-URI = ws: // host [: port] path [? query]
- wss-URI = wss: // host [: port] path [? query]

Il primo per WebSocket standard, il secondo per WebSocket sicuri. La porta è opzionale, il valore di default è 80 per i primi e 433 per i secondi. L'identificatore di una risorsa si ottiene concatenando path e query, separati da un "?". Se il path è vuoto, viene inserito un "/".

Il primo passo per effettuare una comunicazione è l'apertura della connessione da parte del client e l'invio dell'handshake di apertura, specificando un WebSocket URI corretto. Non possono esistere due connessioni in apertura tra uno specifico client e uno specifico server; se un client ha necessità di aprire più connessioni, deve serializzarle e aspettare che la connessione precedente sia stata completata prima di iniziare la successiva. Qualora

il client non possa conoscere se il destinatario è lo stesso (per esempio se agisce dietro un proxy che effettua query DNS), è consigliato limitare il numero massimo di connessioni in apertura ha un valore ragionevolmente basso. Questa limitazione garantisce un primo meccanismo di difesa del server contro attacchi di tipo Denial-of-Service (DOS). Il primo passo per effettuare la comunicazione è perciò la creazione di un socket TCP tra il client e l'host attraverso la porta specificata. Nel caso in cui il client sia in esecuzione dietro a un proxy, la creazione di tale socket deve essere compiuta dal proxy. Se l'operazione è stata completata con successo, il client deve inviare l'handshake di apertura, che deve soddisfare questi requisiti:

1. deve essere una request HTTP 1.1 con metodo GET. Ad esempio, se l'URI del WebSocket è "ws://example.com/chat", la request line dovrebbe essere "GET /chat HTTP/1.1". La risorsa deve essere espressa secondo le specifiche esposte precedentemente.

2. la request deve contenere l'header host, eventualmente aggiungendo in coda ":" e il numero della porta (opzionale), gli header "Upgrade: websocket", "Connection: Upgrade" e "Sec-WebSocket-Version: 13".

3. la request deve inoltre contenere l'header "Sec-WebSocket-Key" il cui valore si ottiene scegliendo 16 byte casuali e codificandoli come base64. Ogni nuova connessione deve avere la propria WebSocket key.

4. se il client che effettua la richiesta è un web browser, deve essere inserito anche l'header origin, che specifica l'identità del client stesso, indicando come valore l'host da cui è stato scaricato il codice che vuole effettuare la connessione.

5. la richiesta può specificare uno o più sottoprocolli di WebSocket da utilizzare, in ordine di preferenza, attraverso l'header "Sec-WebSocket-Protocol". Questo header è opzionale.

6. ogni altro header definito dal protocollo HTTP può essere utilizzato, come ad esempio cookie o authorization.

Dopo che la request è stata inoltrata, il client deve attendere una risposta dal server. Essa deve soddisfare i seguenti requisiti:

1. la response deve avere come status code 101; in caso contrario deve essere considerata come normale risposta HTTP, ad esempio se viene ricevuto un codice 401 il server comunica al client che è necessaria l'autenticazione.

2. tra gli header deve essere contenuto "Upgrade: websocket", "Connection: Upgrade".

3. deve contenere come valore dell'header "Sec-WebSocket-Accept" la concatenazione del valore di Sec-WebSocket-Key precedentemente inviato dal client con la stringa 258EAFA5-E914-47DA-95CA-C5AB0DC 85B11, codificati come SHA-1.

4. non deve contenere nell'header "Sec-WebSocket-Protocol" un sottoprotocollo diverso da quelli eventualmente specificati dalla request del client.

Se questi requisiti sono soddisfatti, la connessione viene definita stabilita, in caso contrario rifiutata. A questo punto, è possibile iniziare la comunicazione vera e propria.

3.2.3 Data framing

Nel protocollo WebSocket, i dati sono trasmessi come sequenze di frame. Per motivi di sicurezza e compatibilità con eventuali intermediari (proxy, ecc), ogni frame inviato dal client deve essere mascherato, anche nelle connessioni non TLS. Viceversa, i frame inviati dal server non devono essere mascherati. Se il server riceve un frame non mascherato deve chiudere immediatamente la connessione; stessa cosa deve fare il client se riceve un frame mascherato. Prima della chiusura effettiva, può essere inviato un frame di chiusura specificando lo status code 1002 (protocol error).

Quando l'handshake è stato completato, e un determinato endpoint non ha ancora inviato un frame di chiusura, tale endpoint può inviare frame di dati. Ogni frame contiene i seguenti bit:

1. **FIN.** 1 bit che indica se il frame è l'ultimo di un messaggio. Poiché un messaggio può essere composto da un solo frame, può essere sia il primo che l'ultimo.

2. **RSV1, RSV2, RSV3.** 3 bit che devono valere 0 a meno che non sia stato negoziata un'estensione del protocollo che renda valori diversi da 0 significativi. Se non c'è stata alcuna negoziazione e questi bit non sono tutti 0, la connessione deve terminare.

3. **Opcode.** 4 bit che definiscono il tipo dei dati contenuti nel frame. I valori possibili sono:

   - 0x0 frame di continuazione
   - 0x1 frame di testo
   - 0x2 frame binario
   - 0x3-7 riservati per uso futuro
   - 0x8 frame di chiusura della connessione
   - 0x9 frame di ping
   - 0xA frame di pong
   - 0xB-F riservati per uso futuro

4. **Mask.** 1 bit che definisce se il frame è mascherato o no. I frame inviati dal client devono avere questo bit settato. Se questo bit è settato, deve essere presente la masking-key.

5. **Payload length.** Indica la lunghezza del payload in byte. Se è compreso tra 0 e 125, quella è la lunghezza. Se è 126, la lunghezza è rappresentata dai due byte successivi (unsigned integer). Se è 127, a rappresentare la lunghezza sono invece gli otto byte successivi. La lunghezza del payload è data dalla somma della lunghezza dell'extension data e dell'application data.

6. **Masking key.** 0 o 4 byte che esprimono la maschera usata per il camuffamento dei dati. Deve essere presente nei frame dove il bit mask è 1 (ovvero nei frame inviati dal client).

7. **Payload Data.** Bit che contengono i dati che si vogliono comunicare (payload). Se durante l'handshake è stata negoziata un'estensione, e la sua lunghezza, è composto da extension data e application data. La lunghezza effettiva dell'application data è la differenza tra la lunghezza del payload e la lunghezza dell'extension data.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-------+-+-------------+-------------------------------+
|F|R|R|R| opcode|M| Payload len |    Extended payload length    |
|I|S|S|S|  (4)  |A|     (7)     |             (16/64)           |
|N|V|V|V|       |S|             |   (if payload len==126/127)   |
| |1|2|3|       |K|             |                               |
+-+-+-+-+-------+-+-------------+ - - - - - - - - - - - - - - - +
|     Extended payload length continued, if payload len == 127  |
+ - - - - - - - - - - - - - - - +-------------------------------+
|                               |Masking-key, if MASK set to 1  |
+-------------------------------+-------------------------------+
| Masking-key (continued)       |          Payload Data         |
+-------------------------------- - - - - - - - - - - - - - - - +
:                     Payload Data continued ...                :
+ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
|                     Payload Data continued ...                |
+---------------------------------------------------------------+
```

Figura 3.2: Struttura di un frame WebSocket.

Da notare che la rappresentazione dei bit che costituiscono un frame WebSocket è binaria, e non ASCII.

Come mostrato, WebSocket è un protocollo che prevede la frammentazione. Ciò comporta diversi vantaggi: rende possibile l'invio di un messaggio di lunghezza sconosciuta non appena iniziato, senza dover aspettare di bufferizzarlo completamente e conoscerne la lunghezza totale. Potendo frammentarlo, infatti, un endpoint può scegliere un buffer delle dimensioni volute, e non appena il buffer è pieno inviare quella porzione di messaggio come frame. Un altro vantaggio si evidenzia nelle situazioni di multiplexing: se lo stesso canale deve essere utilizzato da comunicazioni logicamente diverse, l'invio di pochi ma pesanti messaggi risulta peggiore di frequenti ma più corti messaggi. Di seguito è riportato un esempio di un messaggio frammentato in 3 frame:

1. il primo frame ha FIN = 0, OPCODE = 1.
2. il secondo frame ha FIN = 0, OPCODE = 0;
3. il terzo frame ha FIN = 1, OPCODE = 0;

Il contenuto del messaggio iniziale può essere ottenuto concatenando i tre payload data ricevuti, eventualmente smascherandoli se il bit di maschera è

settato. I frame devono essere ricevuti nello stesso ordine in cui sono stati inviati. Il tipo di un messaggio viene quindi specificato dal valore dell'opcode del primo frame soltanto.

I frame di controllo possono essere inviati in mezzo ai frammenti di un messaggio, ma non devono essere essi stessi frammentati. Nella versione attuale di WebSocket essi sono tre: close, ping e pong. Sono usati per comunicare lo stato del WebSocket e la loro lunghezza deve essere al massimo 125 byte.

Close Frame. Questo frame di controllo ha come valore del bit di opcode 0x8. Viene utilizzato per indicare la chiusura della comunicazione. Può contenere come payload il motivo della chiusura (spegnimento dell'endpoint, frame ricevuto non corretto, ecc). Dopo l'invio di un frame di chiusura, l'endpoint che l'ha inviato non deve inviare più alcun frame. Se un endpoint riceve un frame di chiusura senza averlo prima inviato a sua volta, deve inviarne uno in risposta. Se un Close Frame viene inviato dal client, deve essere mascherato. Quando entrambi gli endpoint hanno inviato e ricevuto un frame di chiusura, la comunicazione si considera conclusa e la connessione TCP sottostante deve essere terminata.

Ping e Pong Frame. Questi frame (con opcode 0x9 e 0xA) hanno lo scopo di verificare l'effettiva presenza dell'altro endpoint. Un endpoint invia all'altro un frame di ping, a cui deve rispondere un frame di pong. Possono essere inseriti dei dati di payload, che devono essere rispediti nel pong. Se un endpoint riceve più ping a cui non ha ancora risposto, può decidere di rispondere solo all'ultimo ping ricevuto.

Frame di dati. Nella versione attuale del protocollo, i frame di dati possono essere di due tipi: testuali (opcode 0x1) o binari (0x2). I bit di payload che contengono non sono significativi per il protocollo, poiché devono essere interpretati a livello applicativo. Gli opcode non assegnati, sia per i frame di dati che per i frame di controllo, possono essere usati durante la comunicazione se durante l'handshake di apertura è stato specificato l'utilizzo di una estensione del protocollo.

3.2.4 Invio e ricezione

Per inviare dati attraverso una connessione WebSocket, un determinato endpoint deve effettuare le seguenti operazioni:

1. Assicurarsi che la connessione sia nello stato open, cioè che l'handshake di apertura sia stato completato con successo.

2. Incapsulare i dati in uno o più frame secondo la struttura definita nella sezione precedente. La lunghezza di ogni frame può essere variabile, a discrezione dell'endpoint.

3. Se l'endpoint è un client, deve offuscare i dati utilizzando la maschera.

Eseguite queste operazioni, il frame o i frame generati possono essere trasmessi attraverso la connessione TCP sottostante.

Per ricevere dati, un endpoint deve essere in ascolto del socket TCP associato al WebSocket. I byte ricevuti devono essere interpretati secondo le specifiche del frame illustrate precedentemente. Se il frame ricevuto è di tipo data, deve esserne tenuta in considerazione la codifica (testuale o binaria). Un messaggio WebSocket viene considerato ricevuto quando il frame con bit FIN = 1 viene rilevato. Se un messaggio è composto da un singolo frame, il payload di quel frame è il messaggio stesso. Altrimenti, l'endpoint deve concatenare i payload di tutti i frame appartenenti allo stesso messaggio. Se l'endpoint che riceve i frame è il server, deve rimuovere l'offuscamento utilizzando la chiave di maschera.

3.2.5 Sicurezza

Questa sezione descrive alcune misure di sicurezza applicate dal protocollo WebSocket.

Non-Browser Client. L'utilizzo del protocollo previsto dovrebbe essere da parte di script all'intero di applicazioni web, in particolare Javascript. Tuttavia potrebbe essere usato direttamente anche da applicazioni client. Per questo motivo, un possibile strumento di difesa è il controllo dell'origine. Inoltre, un server che supporta WebSocket deve sempre controllare il

valore di dati ricevuti utilizzati come input, senza supporre che siano sempre validi; se input malevoli fossero usati all'interno di una query SQL senza essere controllati, il server sarebbe vulnerabile a SQL Injection.

Controllo dell'origine. I server devono accettare connessioni solamente da quei client che ritengono affidabili. Per fare questo devono controllare il valore dell'heder origin durante l'handshake di apertura. Se la richiesta proviene da un client non sicuro, la richiesta deve essere rifiutata. Questo meccanismo protegge in particolare da codice Javascript maligno in esecuzione su un client sicuro: il client può contattare il server, ma quando quest'ultimo rileva che il codice Javascript che vuole stabilire una connessione non è sicuro, può rifiutare la connessione.

Masking. Questa protezione ha lo scopo di proteggere da attacchi rivolti verso componenti del web diversi dagli endpoint della comunicazione, come ad esempio proxy. Una simulazione di questo tipo di attacco è riportata nell'RFC 6455 [30]. La contromisura proposta è di camuffare i dati di payload che effettivamente attraversano la rete mediante una maschera. Tale maschera deve essere diversa per ogni frame e generata in modo random, così da non essere predicibile da terze parti.

3.2.6 Client API

L'utilizzo del protocollo WebSocket lato client è possibile attraverso specifiche API. Esse consentono di aprire e chiudere una connessione, ricevere e inviare messaggi, e reagire in seguito a particolari eventi. Il linguaggio scelto per la trattazione è Javascript.

Apertura della connessione. Per richiedere una connessione è sufficiente creare un oggetto di tipo WebSocket speficiando come parametro l'URL del server. Opzionalmente può essere aggiunta una stringa o un array di sub-protocolli da utilizzare.

```
// Apertura di una connessione WebSocket
ws = new WebSocket("ws://echo.websocket.org", "echo-protocol");
```

Attraverso il metodo onopen è possibile eseguire una callback quando la connessione è stata effettivamente stabilita.

```
// Callback di connessione effettuata
ws.onopen = function(){
   alert("connessione effettuata");
}
```

Chiusura della connessione. Per chiudere una connessione l'oggetto WebSocket mette a disposizione il metodo close.

```
// Chiusura di una connessione WebSocket
ws.close();
```

Per rilevare una disconnessione iniziata dal server si può associare una callback all'evento onclose.

```
// Callback di chiusura connessione
ws.onclose = function(){
   alert("connessione chiusa");
}
```

Per catturare eventuali errori, è presente il metodo onerror, a cui associare l'apposita callback.

```
// Callback in caso di errori
ws.onerror = function(){
   alert("errore durante la connessione");
}
```

Invio di un messaggio. Per inviare un messaggio occorre utilizzare il metodo send.

```
// Invio di un messaggio
ws.send("testo del messaggio");
```

Ricezione di un messaggio. Per la ricezione dei messaggi è necessario associare una callback, tramite il metodo onmessage, che verrà richiamata a ogni messaggio ricevuto.

```
// Callback di ricezione messaggi
ws.onmessage = function(event){
    alert("messagio ricevuto: " + event.data);
}
```

3.2.7 Server API

Lato server, sono disponibili diverse implementazioni del protocollo WebSocket per i linguaggi più diffusi (Java, Python, Node.js, ecc.), ciascuna con proprie API. In questo contesto è stato scelta l'implementazione per Node.js chiamata "WS", resa disponibile sotto forma di modulo [7]. Le API di WS permettono di creare un server WebSocket, accettare connessioni, inviare e ricevere messaggi in formato testuale o binario e gestire gli errori principalmente attraverso l'utilizzo di callback.

Creazione del server. WS permette di creare un server WebSocket standalone, oppure integrarlo ad un server HTTP (utilizzando perciò la stessa porta).

```
// Import del modulo WS
var WebSocketServer = require('ws').Server

// Creazione di un server standalone
wss = new WebSocketServer({port: 80});

// Creazione di un server integrato
wss = new webSocketServer({server: httpServer});
```

Gestione connessioni. Una volta istanziato, l'oggetto WebSocketServer fornisce un metodo (on) attraverso cui effettuare l'associazione evento-callback. Le callback che gestiscono le connessioni hanno come parametro il websocket associato a quella specifica connessione. Su di esso veranno registrate le callback di ricezione e chiusura allo stesso modo del server.

L'invio di un messaggio si effettua col metodo send dell'oggetto WebSocket.

```
// Callback richiamata quando una connessione è accettata
wss.on("connection", function(ws) {

   // Callback richiamata alla ricezione di un messaggio
    ws.on("message", function(message) {
        console.log("ricevuto: %s", message);
    });

    // Invio di un messaggio con callback di controllo errori
    ws.send("message", function(error) {
      if (error != null) {
         console.log("errore durante send");
      }
    });

    // Callback richiamata alla chiusura della connessione
    ws.on("close", function() {
       console.log("connnessione chiusa");
   });
});
```

Capitolo 4

Caso di studio

Per meglio comprendere l'impatto che HTML5 e WebSocket hanno generato nel mondo delle applicazioni Web ho progettato e implementato una Web App multiutente, realizzandone due versioni distinte:

1. nella prima versione ho utilizzato come meccanismo di comunicazione Ajax e Long Polling, precedenti ad HTML5.

2. nella seconda ho utilizzato come tecnologia per la comunicazione i WebSocket di HTML5.

Nella prima sezione verrà descritta l'applicazione dal punto di vista dell'utente finale utilizzatore, mentre nelle sezioni successive verranno analizzati gli aspetti progettuali e le differenze tra le due versioni. In particolare sarà posta attenzione sulle conseguenze che la scelta di un meccanismo per la comunicazione rispetto all'altro genera sull'intero processo produttivo.

4.1 PaintBoard, una lavagna condivisa

4.1.1 Introduzione

PaintBoard, la Web App che ho realizzato, è una lavagna multiutente condivisa. Quando un utente accede all'applicazione ha a disposizione un'area in cui, utilizzando il puntatore del mouse, può disegnare linee di diverso colore e spessore. Se più utenti si collegano all'applicazione, ognuno di essi può vedere in tempo reale le linee tracciate dagli altri utenti, col colore e

lo spessore scelto. Ogni utente ha la possibilità di uscire dalla modalità condivisa e continuare il proprio disegno in maniera autonoma, senza che gli altri utenti lo visualizzino. Allo stesso modo, le linee disegnate da altri utenti non saranno visibili. Ciò è possibile tramite un apposito pulsante di disconnessione. Una volta disconnessi, si può rientrare nella modalità condivisa premendo il tasto di riconnessione; le linee disegnate da quel momento in poi saranno trasmesse e condivise, mentre quelle disegnate durante la modalità offline rimarranno locali.

L'applicazione tiene traccia degli utenti connessi mostrandoli in una lista. Allo scopo di evidenziare le interazioni tra client e server, l'interfaccia utente prevede anche delle finestre di log.

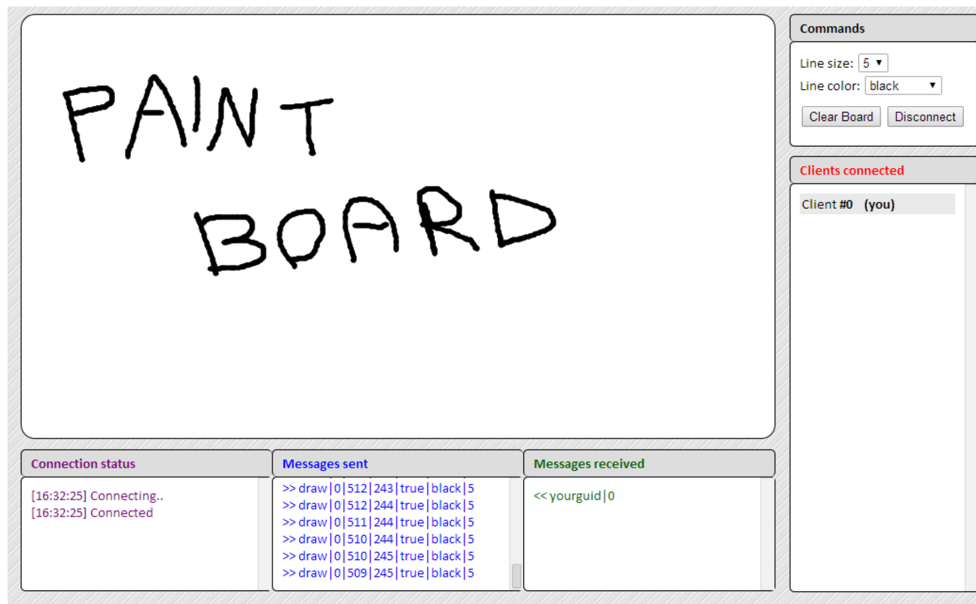

Figura 4.1: Interfaccia utente di PaintBoard.

Le due versioni dell'applicazione hanno la stessa interfaccia uente, ma sono tra loro indipendenti: gli utenti che utilizzano la Web App con WebSocket sono separati da coloro che utilizzano quella con Ajax.

4.1.2 Obiettivi dell'applicazione

Scopo principale di questa applicazione è mostrare come i WebSocket di HTML5 abbiano facilitato enormemente lo sviluppo di applicazioni che richiedono una frequente interazione tra più client e server, specialmente nel caso in cui la comunicazione sia iniziata dal server. Una domanda a cui si cercherà di dare risposta è inoltre se il meccanismo di comunicazione scelto ha ripercussioni sulla intera logica dell'applicazione, o se è una semplice scelta implementativa.

4.1.3 Tecnologie utilizzate

Le tecnologie utilizzate sono state:

- **Node.js.** Node.js è una piattaforma che consente l'utilizzo di Javascript per la realizzazione della parte server di applicazioni network-based. Sfrutta il modello event-driven di Javascript e ha a disposizione diversi moduli per implementare server HTTP e WebSocket. La gestione a eventi di Node.js è parsa particolarmente indicata per questa applicazione, in particolare per la gestione del Long Polling. Per questo motivo è stato preferito Node.js rispetto ai più classici Apache, Tomcat o Glassfish.

- **Ajax e WebSocket** per realizzare la comunicazione tra client e server nelle due diverse versioni della Web App. Javascript fornisce API standard per l'utilizzo di entrambe queste tecnologie.

- **JQuery.** Per velocizzare l'implementazione della applicazione client è stato utilizzato il framework JQuery. Attraverso specifiche API, JQuery permette di eseguire chiamate Ajax o interagire col DOM in modo rapido e cross-browser.

- **HTML5 e CSS3** per realizzare l'interfaccia utente.

Nelle successive sezioni sarà analizzata l'architettura di entrambe le parti dell'applicazione (client e server) sia nel caso WebSocket che nel caso Ajax.

4.2 Architettura client-side

La parte client dell'applicazione è stata progettata seguendo i principi dell'architettura a livelli. I livelli in cui è divisa sono:

1. **Network Layer.** Responsabile della comunicazione col server. Ne sono stati sviluppati due tipi: uno che utilizza WebSocket, e uno che utilizza Ajax.

2. **Application Layer.** Livello che gestisce la lavagna virtuale ed esegue le operazioni su di essa.

3. **Presentation Layer.** Comprende l'interfaccia grafica (view) e un controller per gestire gli eventi dell'utente e interagire con l'application layer.

Per esprimere l'architettura ho utilizzato il linguaggio di modellazione UML. Poiché le tecnologie che verranno utilizzate sono già note a priori, per la progettazione non sono state specificate interfacce, non rappresentabili in Javascript, ma direttamente oggetti. Anche l'ereditarietà è un concetto non presente in Javascript: nonostante sia stato usato in fase di progettazione, l'implementazione finale includerà un solo oggetto con le proprietà e i metodi sia dell'oggetto esteso che dell'estensore.

4.2.1 Struttura

Durante la progettazione ho modellato i componenti interni di ciascun layer, definendone le operazioni che sono in grado di eseguire.

Application Layer. Questo layer comprende la lavagna vera e propria e le funzionalità per gestire il disegno e la pulitura. L'oggetto lavagna è stato esteso per meglio rappresentare il concetto di lavagna condivisa: quest'ultima, a differenza della prima, è in grado di disegnare linee generate da più utenti e può comunicare con il Network Layer per l'invio delle linee disegnate localmente. La lavagna, essendo un oggetto Javascript, si basa sul modello event-driven.

L'altro componente è il Controller, che cattura le azioni effettuate dall'utente (disegno sulla lavagna, click sui bottoni, ecc) attraverso appositi handler.

Tali handler comunicano alla lavagna di effettuare determinate operazioni, ed eventualmente notificano alla view di aggiornare l'interfaccia grafica.

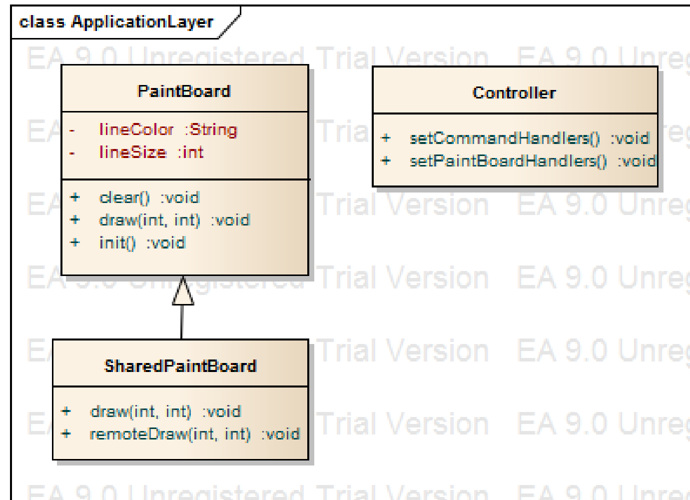

Figura 4.2: Struttura dell'Application Layer.

Presentation Layer. Esso contiene l'interfaccia grafica, che sarà implementata come singola pagina HTML5 e CSS3, più un componente che rappresenta la vista. Questo componente, chiamato View, è in grado di modificare l'interfaccia grafica visualizzata dall'utente, attraverso specifiche API. Esse sono richiamate dagli altri componenti dell'applicazione in seguito a particolari eventi. Anche questo layer contiene componenti che si basano sul modello event-driven.

Network Layer. Questo livello si occupa di interagire con il server, attraverso un apposito componente. Poiché la logica con cui questa interazione avviene cambia nel caso Ajax rispetto a WebSocket, ne sono state realizzate due versioni. Compito di questo componente è connettersi al server, comunicare con esso per ottenere una lista sempre aggiornata degli utenti connessi, inviare le linee disegnate localmente e ricevere quelle disegnate da altri client. Esso interagisce con il livello applicativo per la gestione delle linee da inviare e ricevute, e può delegare alla View l'aggiornamento dell'interfaccia grafica.

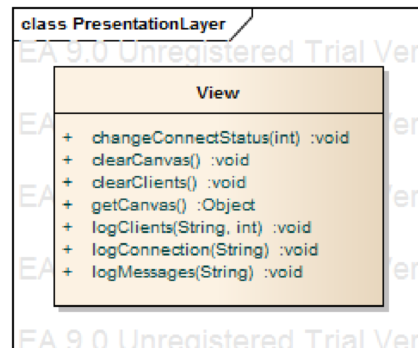

Figura 4.3: Struttura del Presentation Layer.

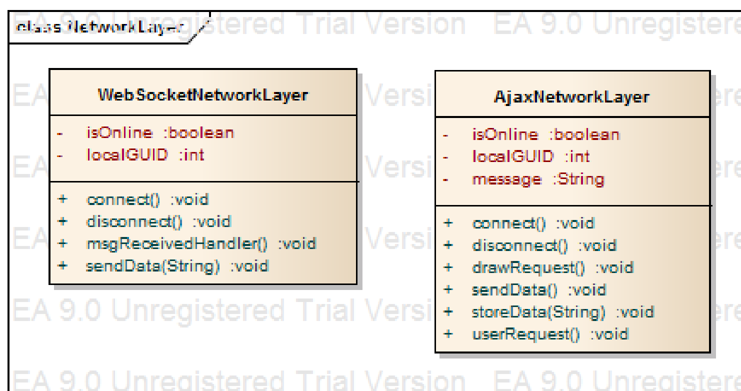

Figura 4.4: Struttura del Network Layer.

4.2.2 Interazione con WebSocket

Dopo aver enunciato la struttura dell'applicazione, verrà ora analizzata l'interazione che i componenti hanno tra loro sia internamente al client, che sopratutto tra client e server, nella versione che utilizza WebSocket.

L'applicazione viene lanciata digitando il rispettivo URL. Appena la pagina viene caricata, il Network Layer apre un WebSocket con il server, attraverso il metodo connect(). Se il socket viene aperto e la connessione stabilita, il Network Layer associa al socket un handler che gestisca la ricezione di messaggi e un handler per la gestione della chiusura della connessione. Non

appena ciò accade, il server invia al client un ID generale univoco (GUID) che lo identifichi. Per l'invio di messaggi attraverso il socket è disponibile il metodo sendData, che invia la stringa passata come parametro. Grazie alle proprietà dei WebSocket descritti nel capitolo precedente, il Network Layer è in grado, in ogni momento, di sapere se la connessione è ancora attiva o se dei messaggi sono stati ricevuti, e gestire tali eventi con gli handler. L'operazione di invio di un messaggio è quindi semplice: è sufficiente scrivere nel socket precedentemente aperto. Per questo motivo, ogni singolo pixel disegnato viene inviato singolarmente al server, in modo altri client connessi vedano la linea disegnarsi pixel per pixel. Ciò avviene seguendo questo flusso:

1. Il Controller (Presentation Layer) cattura il movimento del mouse nell'area di disegno e lo comunica alla PaintBoard (Application Layer) specificando le coordinate.

2. La PaintBoard disegna il pixel e comunica al Network Layer le coordinate da inviare, il colore e lo spessore della linea.

3. Se l'applicazione è in modalità online, il Network Layer invia le informazioni ricevute e il GUID al server scrivendo nel socket, altrimenti non fa nulla.

Le informazioni che l'utente deve ricevere dal server sono: utenti connessi, stato della connessione ed eventuali linee disegnate da altri. Per ottenere tutti questi dati è sufficiente l'handler in ascolto in ricezione. A seconda del messaggio ricevuto, richiamerà la View per aggiornare la GUI o la PaintBoard per il disegno della linea.

Di seguito viene riportato il sorgente di parte del NetworkLayer WebSocket, contenente i metodi che gestiscono la connessione, la disconnessione e la ricezione dei messaggi.

```
// Connessione al server tramite WebSocket
networkLayer.connect = function() {
   view.logConnection("Connecting..");
   var wsCtor = window['MozWebSocket'] ? MozWebSocket : WebSocket;
   this.socket = new wsCtor(this.SOCKET_URL,
      'paintBoard-protocol');
```

```
    this.socket.onopen = function() {
        networkLayer.isOnline = true;
        view.logConnection("Connected");
        view.changeConnectStatus(0);
    }

    this.socket.onmessage = function(message) {
        networkLayer.msgReceivedHandler(message);
    }

    this.socket.onclose = function() {
        networkLayer.isOnline = false;
        view.logConnection("Disconnected");
        view.changeConnectStatus(1);
        view.clearClients();
    };
};


// Handler richiamato a ogni messaggio ricevuto
networkLayer.msgReceivedHandler = function(message) {
    view.logMessages(message.data, 1);
    var tokens = message.data.split("|");
    if (tokens[0] == "yourguid") {
        this.localGUID = tokens[1];
        view.logClients(this.localGUID+" (you)", 1);
    }
    if (tokens[0] == "newguid") {
        view.logClients(tokens[1], 1);
    }
    if (tokens[0] == "quit") {
        view.logClients(tokens[1], 0);
        paintBoard.removeRemoteCoords(tokens[1]);
    }
    if (tokens[0] == "draw") {
        paintBoard.remoteDraw(tokens[2], tokens[3], tokens[4],
            tokens[1], tokens[5], tokens[6]);
    }
```

```
}


// Disconnessione
networkLayer.disconnect = function() {
   this.socket.close();
}
```

4.2.3 Interazione con Ajax Long Polling

Nella versione che utilizza Ajax, l'interazione dei componenti interni al client non cambia rispetto alla versione con WebSocket. Ciò che invece è molto diverso è il Network Layer e la gestione della comunicazione client-server.

Ajax Long Polling, come discusso nel capitolo precedente, consiste in request successive del client al server, a cui quest'ultimo risponde istantaneamente se ha dei dati da inviare, altrimenti lascia la request in attesa finché essi non siano disponibili, e poi risponde. La comunicazione è quindi composta da serie di request/response. Il compito del Network Layer in questo caso è:

1. Effettuare una request iniziale Ajax, a un indirizzo specifico, che rappresenti una richiesta di connessione. Il server, accettando la richiesta, risponderà con il GUID assegnato all'utente. Ricevuto un GUID, il client si considera connesso.

2. Una volta connesso, effettuare chiamate Ajax in long polling a due indirizzi: uno per ottenere aggiornamenti sugli utenti connessi, e uno per ottenere le linee disegnate dagli altri client.

3. Per l'invio dei dati eseguire una request a un indirizzo apposito usando come metodo POST, inviando il proprio GUID e le informazioni.

Il Network Layer è quindi un componente attivo: deve eseguire delle request, attendere le risposte e analizzarle: se contengono dati validi le invierà al layer applicativo, se non contengono alcun dato non farà nulla. In entrambi i casi, non appena riceve la response, rinvierà istantaneamente la stessa request. Per limitare il numero di POST e di request/response necessarie

all'invio e alla ricezione dei punti tracciati, il Network Layer memorizza un intera linea, considerata dall'inizio della pressione del mouse fino al suo rilascio, e solo alla fine la invia in un unico messaggio. Dal punto di vista dell'utente finale, ciò viene percepito come una minore reattività dell'applicazione, ma garantisce una miglior gestione della rete.

Ecco una parte significativa del NetworkLayer Ajax, che gestisce la connessione e la request delle linee disegnate.

```
// Connessione al server con Ajax
networkLayer.connect = function() {
   this.aborted = false;
   view.logConnection("Ajax long polling started");
   view.changeConnectStatus(0);
   view.logMessages("GET /firstRequest", 0);
   this.connectAjax = $.ajax({
      url: "firstRequest",
      type: "GET",
   });
   this.connectAjax.done(function( msg ) {
      var messages = msg.split("-");
      for (var i = 0; i < messages.length; i++) {
         if (messages[i] != "") {
            var tokens = messages[i].split("|");
            if (tokens[0] == "yourguid") {
               networkLayer.localGUID = tokens[1];
               view.logClients(networkLayer.localGUID+" (you)", 1);
            }
            if (tokens[0] == "newguid") {
               view.logClients(tokens[1], 1);
            }
         }
      }
      if (msg != "timeout") {
         view.logMessages("response received", 0);
      }
      networkLayer.isOnline = true;
      networkLayer.userRequest();
      networkLayer.drawRequest();
```

```
    });
    this.connectAjax.fail(function( jqXHR, textStatus ) {
        view.logMessages("-closed-", 0)
        networkLayer.disconnect();
    });
}


// Richiesta al server di aggiornamenti sulla linee disegnate
networkLayer.drawRequest = function() {
    view.logMessages("GET /getDraw/"+networkLayer.localGUID, 1);
    this.drawAjax = $.ajax({
        url: "getDraw/"+ networkLayer.localGUID,
        type: "GET",
    });
    this.drawAjax.done(function( msg ) {
        if (msg != "timeout") {
            var messages = msg.split("-");
            for (var i = 1; i < messages.length; i++) {
                if (messages[i] != "") {
                    var tokens = messages[i].split("|");
                    paintBoard.remoteDraw(tokens[0], tokens[1],
                        tokens[2], messages[0], tokens[3], tokens[4]);
                }
            }
            view.logMessages("response received", 1);
        }
        networkLayer.drawRequest();
    });
    this.drawAjax.fail(function( jqXHR, textStatus ) {
        view.logMessages("-closed-", 1)
        networkLayer.disconnect();
    });
}
```

4.3 Architettura server-side

La parte server dell'applicazione è stata pensata come un unica applicazione che, una volta lanciata, è in grado di comunicare con client connessi sia con WebSocket che tramite Ajax contemporaneamente. I due tipi di comunicazione rappresentano però contesti separati: un client WebSocket vede online e interagisce solo con altri client WebSocket, stessa cosa per Ajax.

4.3.1 Struttura

Il back-end dell'applicazione è formato dai seguenti componenti:

1. **Server HTTP.** Il server HTTP ha il compito di rilevare le request alla porta 80, processarle e inviare le response. Sono presenti due tipi di request: quelle che richiedono pagine statiche, che rappresentano l'entry point dell'applicazione (clientWebSocket.html e clientAjax.html), e quelle che riguardano Ajax. Il server HTTP gestisce direttamente il primo tipo, mentre delega a un altro componente (l'Handler Ajax) la risoluzione del secondo tipo.

2. **Server WebSocket.** Questo componente gestisce la comunicazione tramite WebSocket (connessione, ricezione ed invio) con i client che utilizzano questo metodo di comunicazione ed implementa la business logic server side della versione WebSocket dell'applicazione. Attraverso degli handler, effettua delle operazioni in risposta a eventi associati a WebSocket connessi. Il suo comportamento verrà analizzato nel dettaglio nella prossima sezione.

3. **Handler per le request Ajax.** I client che utilizzano la seconda versione dell'applicazione effettuano request Ajax che vengono gestite da questo componente. Esso estende il server HTTP fornendo nuove risorse (tramite nuovi URL) di cui i client. attraverso chiamate asincrone Ajax, effettuano richieste in long polling. Tali risorse rappresentano gli utenti connessi, le nuove linee disegnate, o l'indirizzo a cui inviare tramite POST le proprie linee disegnate. La logica applicativa del server è quindi racchiusa in questo componente. Come per il server WebSocket, le prossime sezioni ne analizzeranno il comportamento in dettaglio.

4. **GUID Generator.** Il GUID Generator genera ID univoci. A ogni client che si connette, il server assegna il proprio GUID richiedendo un nuovo GUID a questo componente.

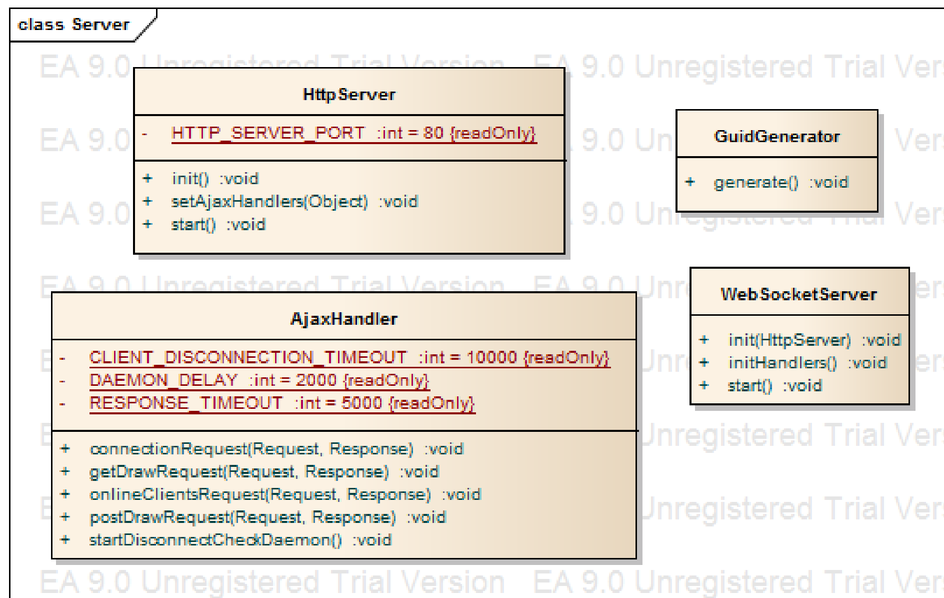

Figura 4.5: Struttura della parte server-side dell'applicazione.

4.3.2 Interazione con WebSocket

Come anticipato nella sezione precedente, il componente che si occupa della comunicazione tra client e server nella versione WebSocket è ServerWebSocket. Quando un client, attraverso le apposite API Javascript, richiede una connessione, esso:

1. Accetta la connessione e genera un nuovo GUID, memorizzando internamente l'associazione WebSocket-GUID per il riconoscimento.

2. Invia al client appena connesso il suo GUID, e la lista di tutti gli altri GUID attualmente connessi, prelevandoli dalla sua memoria interna.

3. Invia in broadcast a tutti i WebSocket già connessi il GUID del client appena connesso.

A ogni WebSocket connesso viene associato un handler di ricezione che, a ogni messaggio ricevuto, invia in broadcast a tutti gli altri socket (diversi dal mittente) il messaggio stesso. A ogni disconnessione, un altro handler invia, sempre con la stessa modalità, un messaggio di disconnessione specificando il GUID associato al WebSocket disconnesso. Grazie a questi semplici handler il WebServer non ha bisogno di implementare nessuna logica particolare. Le uniche informazioni che deve memorizzare sono la lista dei WebSocket connessi e il GUID assegnato a ognuno.

Il protocollo utilizzato per la comunicazione è il seguente:

- "yourguid/<id>" per comunicare il proprio GUID a nuovo client
- "newguid/<id>" per comunicare la connessione di un nuovo client
- "quit/<id>" per comunicare la disconnessione di un client
- "draw/<payload>" per comunicare informazioni destinate al layer applicativo dei client, come coordinate, spessore e colore della linea, GUID del disegnatore, ecc.

Ecco la parte di codice sorgente che accetta una connessione WebSocket e associa ai vari eventi la rispettiva callback.

```
wss = new webSocketServer({server: httpServer});

wss.on('connection', function(ws) {
   console.log("[WebSocket] " + ws._socket.remoteAddress + ":"+
      ws._socket.remotePort + " connected");

   // Send to the new client his GUID
   var guid = guidGen.generate();
   ws.send("yourguid|" +guid, function(error) {
      if (error != null) {
         console.log("[WebSocket] Cannot send new GUID");
      }
   });
```

```
    // And all the other clients connected
    for (var i=0, len = clients.length; i < len; i++) {
       ws.send("newguid|" +clients[i][1], function(error) {
          if (error != null) {
             console.log("[WebSocket] Cannot send connected clients'
                 GUID");
          }
       });
    }
    clients.push(new Array(ws, guid));
    console.log("[WebSocket] GUID: "+guid);


    // Broadcast the new GUID to all the others
    wss.broadcast("newguid|"+guid, ws);


    // Closing handler
     ws.on('close', function() {
       for (var i=0, len = clients.length; i < len; i++) {
          if (clients[i][0] == ws) {
             wss.broadcast("quit|"+clients[i][1], ws)
             console.log("[WebSocket] GUID "+clients[i][1]+"
                 disconnected");
             clients.splice(i, 1);
             break;
          }
       }
     });

     ws.on('message', function(message, flag) {
       if (flag.binary == undefined && flag.masked == true) {
          wss.broadcast(message, ws);
       }
     });
});
```

4.3.3 Interazione con Ajax Long Polling

I compiti che il server deve svolgere nella versione con Ajax Long Polling sono invece diversi e più complessi. Il server mette a disposizione alcune risorse virtuali, accessibili tramite specifici che URL, che rappresentano variazioni dello stato degli utenti online o delle linee disegnate. Eseguendo long polling su queste risorse, un client può essere aggiornato in modo relativamente reattivo. Tali risorse sono:

1. **<app-url>/firstRequest**. La prima request che il client effettua deve essere a questo indirizzo. Nella response il server inserisce il GUID assegnato al client e la lista degli utenti connessi. Il server memorizza localmente il GUID associandolo al timestamp corrente. Inoltre, risponde a tutte le request alla risorsa /users in attesa inviando il GUID appena generato.

2. **<app-url>/users/<GUID>**. Subito dopo la prima request, il client deve effettuare una request a questo indirizzo. Il server memorizza tale request e crea una callback che verrà eseguita dopo un certo intervallo di tempo. Allo scadere del timeout, se la request non è ancora stata servita (cioè non si sono connessi nuovi client) verrà inviata una risposta vuota. Inoltre, alla ricezione della request, il server aggiorna il timestamp associato al GUID che effettua la richiesta, allo scopo di memorizzare il momento dell'ultima azione eseguita per ogni client. Ogni client, dopo aver ricevuto la response, deve rieffettuare immediatamente una nuova request.

3. **<app-url>/getDraw/<GUID>**. Come nel caso precedente, anche questo indirizzo deve essere richiamato subito dopo la prima request. La logica è la stessa: la request viene memorizzata, e viene istanziata la callback di timeout. Se un client disegna, il server crea le response con i dati ricevuti e risponde a tutte le request in coda. Se nessun dato arriva, le callback di timeout mandano response vuote alla scadere.

4. **<app-url>/draw**. Quando un client ha disegnato una linea (dal momento della pressione del mouse fino al suo rilascio) deve inviarla al server sotto con metodo POST a questo indirizzo. Alla ricezione, il

server risponde a tutte le request di getDraw in coda, tranne a quella del client che ha inviato la linea stessa.

Per gestire la disconnessione, il server periodicamente controlla il timestamp dell'ultima request effettuata da ogni client. Se è superiore a un certo intervallo, il client sarà considerato disconnesso e una response contenete la notifica di disconnessione sarà inviata a tutti i client che hanno una request per /users/ in coda.

Di seguito è riportato parte del codice sorgente dell'AjaxHandler.

```
// Connected clients' GUID and last action time
var clients = new Array();
// Unsent response
var resps = new Array();
// Unsent getDraw response
var drawResps = new Array();


// Connection request handler
var connectionRequest = function(req, res) {
   var list = "";
   var guid = guidGen.generate();
   for (var i=0, len = clients.length; i < len; i++) {
      list += "newguid|" +clients[i][0]+"-"
   }
   var text = "yourguid|"+guid+"-"+list;
   res.send(200, text);
   var len = resps.length;
   while (len--) {
      resps[len].send(200, "newguid|"+guid);
      console.log("[Ajax] /users request served");
      resps.splice(len, 1);
   }
   clients.push(new Array(guid, new Date().getTime()));
}


// Draw request handler
var getDrawRequest = function(req, res) {
```

```
   var guid = req.params.GUID;
   drawResps.push(new Array(res, guid));
   console.log("[Ajax] "+guid+" /getDraw request accepted..");
   setTimeout(function(){ timeoutDrawCallback(res)},
       RESPONSE_TIMEOUT);
   function timeoutDrawCallback(res) {
      var len = drawResps.length;
      while (len--) {
         if (drawResps[len][0] == res) {
            console.log("[Ajax] /getDraw request timeout");
            res.send(200, "timeout");
            drawResps.splice(len, 1);
            break;
         }
      }
   }
}


// Post draw request handler
var postDrawRequest = function(req, res) {
   var text = req.body.text;
   var guid = req.body.guid;
   res.send(200, "received");
   var len = drawResps.length;
   while (len--) {
      if (drawResps[len][1] != guid) {
         drawResps[len][0].send(200, text);
         console.log("[Ajax] /draw request served");
         drawResps.splice(len, 1);
      }
   }
}


// Start daemon
var startDisconnectCheckDaemon = function() {
   var len = clients.length;
   while (len--) {
```

```
        if (new Date().getTime() - clients[len][1] >
            CLIENT_DISCONNECTION_TIMEOUT) {
            console.log("[Ajax] GUID "+clients[len][0]+"
                disconnected");
            var len2 = resps.length;
            while (len2--) {
                resps[len2].send(200, "quit|"+clients[len][0]);
                console.log("[Ajax] /users request served");
                resps.splice(len2, 1);
            }
            clients.splice(len, 1);
        }
    }
    setTimeout(startDisconnectCheckDaemon, DAEMON_DELAY);
}
```

4.4 Discussione

La Web App è stata sviluppata in due versioni: con WebSocket e con Ajax Long Polling. Dato che le due versioni si distinguono per un meccanismo implementativo diverso, e non per diversi requisiti e/o funzionalità, è lecito pensare che molti componenti di una versione possano essere riutilizzati nell'altra. Nella parte client-side è stato infatti cosi: il layer di presentazione e di business logic sono gli stessi in entrambe le versioni. Il Network Layer è invece, come si poteva intuire, ben diverso. Entrambi i meccanismi possono produrre gli stessi effetti, ma Ajax Long Polling, proprio per la sua logica di polling, è più complicato e difficile da gestire. Nell'applicazione in esame, infatti, le richieste contemporanee erano limitate (utenti online e linee disegnate); in una applicazione complessa dove il loro numero è molto maggiore, ciò porterebbe all'apertura di decine di richieste contemporanee al server per ogni client. Con WebSocket ciò non si verifica. Ogni client usa un solo socket nel quale possono essere inviate e ricevute tutte le informazioni necessarie. Con long polling, poiché a ogni messaggio che il client deve inviare corrisponde la creazione di una nuova request/response, è stato deciso di inviare una linea solo al suo completamento, invece che pixel per pixel. Questo porta, dal punto di vista dell'utente, a una perdita di reatti-

vità, in quanto non è in grado di vedere la composizione della linea stessa.

Lato server, tuttavia, è dove si hanno le più grandi differenze. La prima riguarda come riconoscere la disconnessione di un client. I WebSocket, basandosi su TCP, possono rilevare immediatamente la perdita di connessione e notificarlo all'applicazione tramite un evento che un handler può catturare. Viceversa in Ajax questo non è possibile. La soluzione adottata in PaintBoard è un controllo, per ogni client, dell'intervallo trascorso dall'ultima request effettuata. Quando questo supera un certo valore, il client viene considerato disconnesso. In caso però di congestione della rete, tuttavia, un client potrebbe non essere in grado di effettuare nuove request ma essere comunque connesso. Un problema inoltre può nascere dalla latenza tra client e server. Si supponga questo situazione:

1. un client effettua la requst per lo stato degli utenti online.
2. il server, non rilevando nessuna connessione/disconnessione, risponde allo scadere del timeout con una risposta vuota.
3. il client, ricevuta la response, invia una nuova request.

Qualora un nuovo client si connettesse tra i punti 2 e 3, cioè quando il server ha già inviato la response ma non ha ancora ricevuto la nuova request, il client iniziale non ne sarebbe a conoscenza, nonostante sia a tutti gli effetti connesso. Per superare questo problema il server dovrebbe memorizzare localmente, per ogni client, per ogni tipologia di response, il suo timestamp e tenere una coda degli eventi. Alla successiva request, vedere quali eventi sono successivi all'ultima request e, se presenti, notificarli immediatamente. Ciò rappresenta, rispetto a WebSocket, una complessità maggiore, che aumenta sempre di più al crescere delle request contemporanee.

Il vantaggio fondamentale dei WebSocket è comunque la possiblità per il server di effettuare PUSH in modo semplice e veloce, grazie alla disponibilità continua di una connessione aperta col client.

Capitolo 5

Conclusioni

Questa tesi ha voluto illustrare lo stato attuale delle applicazioni Web, fornendo una base per la loro progettazione e sviluppo. In particolare è stato analizzato l'aspetto della distribuzione e i meccanismi con cui può essere implementato. Sono state evidenziate le differenze tra le tecnologie già esistenti, come Ajax e Flash, e il nuovo protocollo di comunicazione di HTML5, i WebSocket. La caratteristica principale dei WebSocket è il mantenimento di una connessione, sotto forma di Socket TCP, costantemente aperta tra client e server, in modalità full-duplex. Grazie a questo, paradigmi di comunicazione diversi dal semplice request-response tipico di HTTP possono essere realizzati con semplicità; ad esempio, l'invio di dati da parte del server al client senza precedente sollecitazione, chiamato server Push, si può realizzare facilmente con WebSocket, mentre con le tecnologie precedenti era necessario simulare questo comportamento attraverso diverse tecniche, tra cui il Long Polling.

A conferma di quanto espresso nel corso della tesi, ho progettato e realizzato una Web App in cui la comunicazione client-server è l'aspetto preponderante. Tale App, chiamata PaintBoard, è una lavagna virtuale, in cui ogni client connesso può disegnare e vedere in tempo reale le linee create dagli altri client. Ne sono state implementate due versioni, una con WebSocket, e una con Ajax e Long Polling. La versione con WebSocket ha dimostrato che questa tecnologia porta un effettivo vantaggio sotto diversi aspetti, il principale dei quali è la possibilità di eseguire server Push in modo semplice. Alcuni nuovi modelli di interazione come Reactive Extensions (RX) possono

beneficiare di questa caratteristica in modo sostanziale.

In conclusione, quindi, considerare l'arrivo di HTML5 e delle tecnologie a esso associate (WebSocket, WebGL, ecc) come inizio di una nuova era per il Web risulta una decisione appropriata; è probabile che in un futuro non troppo remoto le Web App possano sostituire sempre di più le normali applicazioni dekstop.

Ringraziamenti

Ringrazio il mio relatore, prof. Alessandro Ricci per l'aiuto e il supporto che mi ha fornito nell'elaborazione della tesi.

Ringrazio la mia famiglia, mia mamma, mio babbo e mia sorella, per l'amore e il sostegno durante questo periodo.

Ringrazio la mia fidanzata, Nausica, per essermi stata vicino in ogni momento.

Ringrazio i miei amici per aver condiviso con me i tre anni che hanno portato a questo traguardo.

Bibliografia

[1] World wild web consortium. http://www.w3.org.

[2] Adobe.com. Adobe flash platform. http://help.adobe.com/en_US/as3/dev/WSb2ba3b1aad8a27b0-181c51321220efd9d1c-8000.html.

[3] S. Andrews. Architecture of a modern web app. In *SpringOne 2GX*, 2013.

[4] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. 2000.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1995.

[6] T. Garsiel. How browsers work. http://taligarsiel.com/Projectshowbrowserswork1.htm.

[7] GitHUB. Node.js websocket implementation. http://einaros.github.io/ws/.

[8] N. W. Group. Http/1.1: Method definitions. http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html.

[9] M. K. Gunnulfsen. *Scalable and Efficient Web Application Architectures*. 2013.

[10] HashPHP. Http primer. http://wiki.hashphp.org/HttpPrimer.

[11] HTML5Rocks. Field guide to web application. http://www.html5rocks.com/webappfieldguide/toc/index/.

[12] HTML.it. Guida ajax. http://www.html.it/guide/guida-ajax/.

[13] HTML.it. Html5 websocket.
http://www.html.it/articoli/html5-websocket-1/.

[14] HTML.it. Il modello di comunicazione long polling.
http://www.html.it/pag/33600/il-modello-di-comunicazione-long-polling/.

[15] HTML.it. Introduzione ai websocket.
http://www.html.it/articoli/introduzione-ai-websocket/.

[16] IEEE. Recommended practice for architectural description of software-intensive systems.
http://cabibbo.dia.uniroma3.it/ids/altrui/ieee1471.pdf.

[17] M. Jazayeri. Some trends in web application development. In *Future of Software Engineering*, 2007.

[18] K. S. Kumar. Architecture of a modern web application.
http://www.techiekernel.com/2012/12/architecture-of-modern-web-application.html.

[19] L. Madeyski and M. Stochmialek. Architectural design of modern web applications. In ", 2011.

[20] M. MSDN. Reactive extensions.
http://msdn.microsoft.com/en-us/data/gg577609.aspx.

[21] M. A. O. Mukhtar, M. F. B. Hassan, J. B. Jaafar, and L. A. Rahim. Enhanced approach for developing web applications using model driven architecture. In ", 2013.

[22] D. Nelson. Next gen web architecture for the cloud era. In *Saturn*, 2013.

[23] M. Nowak and C. Pautasso. The design space of modern html5/javascript web application. In *Saturn*, 2013.

[24] G. Prasad, R. Taneja, and V. Todankar. *Life above the Service Tier*. 2007.

[25] J. M. A. Santamaria. The single page interface manifesto. http://itsnat.sourceforge.net/php/spim/spi_manifesto_en.php.

[26] L. Shklar and R. Rosen. *Web Application Architecture: principles, protocols and practices*. John Wiley & Sons, Ltd, 2004.

[27] A. Taivalsaari and T. Mikkonen. The web as an application platform: The saga continues. In *37th EUROMICRO Conference on Software Engineering and Advanced Applications*, 2011.

[28] A. Tanenbaum and M. V. Steen. *Distributed Systems: Principles and Paradigms*. 2007.

[29] W3C. Http/1.1: Header field definitions. http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html.

[30] WHATWG. The websocket protocol. http://www.whatwg.org/specs/web-socket-protocol/.

[31] Wikipedia. Adobe flash. http://it.wikipedia.org/wiki/Adobe_Flash.

[32] Wikipedia. Design patterns. http://it.wikipedia.org/wiki/Design_Patterns.

[33] Wikipedia. Html5. http://it.wikipedia.org/wiki/HTML5.

[34] Wikipedia. Multitier architecture. http://en.wikipedia.org/wiki/Multitier_architecture.

[35] Wikipedia. Push technology. http://en.wikipedia.org/wiki/Push_technology.

[36] WIkipedia. Representational state transfer. http://it.wikipedia.org/wiki/Representational_State_Transfer.

[37] WIkipedia. Web 2.0. http://it.wikipedia.org/wiki/Web_2.0.

[38] S. Zhaoyun, Z. Xiaobo, and Z. Li. The web asynchronous communication mechanism research based on ajax. In *2nd international Conference on Education Technology and Computer*, 2010.