

Università degli Studi di Salerno
Facoltà di Scienze Matematiche Fisiche e Naturali

Tesi di Laurea di I livello in
Informatica

Progettazione ed implementazione di un plug-in su Firefox per raccogliere il feedback degli utenti durante la navigazione web

Relatore
Prof. Alberto Negro

Candidato
Cristiano Raimondi
matr. 0512100280

Anno Accademico 2011-2012

Dediche e ringraziamenti

I primi ringraziamenti sono rivolti in ambito accademico al mio relatore di tesi, *prof.re Alberto Negro*, per la sua grande disponibilità, professionalità e fiducia avuta nei miei confronti durante i mesi che ho svolto tirocinio e tesi in ISISLab. Un sentito grazie va al *dottore Gennaro Cordasco* per avermi dato la possibilità di intraprendere il percorso di tirocinio in ISISLab e per essermi stato vicino, sia moralmente che didatticamente, durante tutto il periodo di svolgimento della tesi. Desidero ringraziare la mia famiglia per il loro continuo supporto grazie al quale sono riuscito a raggiungere questo traguardo. Infine desidero ringraziare tutte le persone a me care che hanno condiviso con me tutti i momenti, di gioia e di difficoltà, che hanno contraddistinto il mio percorso universitario.



Indice

1 Introduzione **1**
- 1.1 Che cos'è un *feedback* ? . . . 1
- 1.2 Il *feedback* quale strumento di indagine di mercato . . . 1
- 1.3 L'importanza dei *feedback* nel *web* . . . 2
- 1.4 L'idea di uno strumento informatico per la raccolta dei *web feedback* . . . 2
- 1.5 Organizzazione della tesi . . . 3

2 Le architetture di riferimento **4**
- 2.1 Modello *client/server* . . . 4
  - 2.1.1 I vantaggi del modello *client/server* . . . 9
  - 2.1.2 Gli svantaggi del modello *client/server* . . . 10
- 2.2 Modello *client/proxy/server* . . . 10
  - 2.2.1 I vantaggi del modello *client/proxy/server* . . . 10
  - 2.2.2 Gli svantaggi del modello *client/proxy/server* . . . 11
- 2.3 L'architettura proposta . . . 11

3 Tecnologie utilizzate **13**
- 3.1 Il *client* . . . 16
  - 3.1.1 *Add-on SDK* . . . 16
- 3.2 Il *server* . . . 23
  - 3.2.1 *PostgreSQL* . . . 25
  - 3.2.2 Scegliere *PostgreSQL* . . . 26

4 Il *plug-in youtube-emotion* **27**
- 4.1 La fase di progettazione e implementazione . . . 28
  - 4.1.1 Cartella *lib* . . . 29
  - 4.1.2 Cartella *data* . . . 33
- 4.2 La fase di installazione su *Firefox* . . . 34

4.3 L'utilizzo del *plug-in* sul *web* 36
4.4 Il testing dell'applicazione 38

5 Conclusioni **42**

Bibliografia **44**

A Appendice **47**
A.1 Listati 47
A.1.1 Il *file* "*main.js*" 47
A.1.2 Il *file* "*contentPanel.html*" 50
A.1.3 Il *file* "*scriptPanel.js*" 51
A.1.4 Il *file* "*stile.css*" 52

Capitolo 1

Introduzione

Attraverso la progressiva e capillare diffusione di *Internet* nel corso degli ultimi anni, il numero delle persone che interagiscono con il *web* è notevolmente aumentato. A seguito di tale evoluzione i progettisti hanno capito l'importanza di sviluppare un'area dove la generalità degli utenti potesse esprimere le proprie opinioni circa il servizio, o, il tema trattato nello specifico sito visitato. Lo strumento idoneo ad ottenere il suddetto scopo è il cosiddetto *feedback*.

1.1 Che cos'è un *feedback* ?

Il *feedback*, termine inglese che tradotto in italiano significa retroazione, è una risposta, un opinione, un giudizio che chiunque può fornire su un specifico tipo di argomento. E' uno strumento che ha un vasto campo applicativo. Può, infatti, essere utilizzato al fine di ottenere e analizzare informazioni e dati in molti contesti: nel *marketing*, nell'informatica, per studi psicologici, eccetera [1].

I *feedback* sono altresì utilizzati anche nei social network con l'intento di permettere agli iscritti di esprimere le proprie opinioni in diversi modi: aggiungere un commento, condividere il proprio *post* e/o commentare quello di altri, eccetera.

1.2 Il *feedback* quale strumento di indagine di mercato

Qualsiasi attività di *marketing* viene implementata unicamente quando l'azienda dispone di sistemi di controllo e valutazione delle *performance* pie-

namente funzionanti [2].

In particolar modo, nel marketing le aziende utilizzano i *feedback*, facendo uso anche di appositi siti *web* dedicati, affinché i propri clienti esprimano il loro grado di soddisfazione sui prodotti acquistati.

Le aziende che fanno uso di tali strumenti possono indagare meglio il mercato a cui si rivolgono al fine di ottenere vantaggi in termini di controllo della reputazione e con l'obbiettivo di aumentare la qualità dei prodotti di vendita e/o servizi offerti [3].

Un esempio è rappresentato dal sistema di valutazione degli utenti e dei prodotti acquistati mediante *eBay*.

Per mezzo di tale particolare *feedback* l'utente rilascia un'opinione sui prodotti acquistati che aiutano a comprendere la qualità degli acquisti e la buona fede dei venditori. In secondo luogo, il sistema di *feedback* si propone l'obiettivo di contrastare le note truffe *online* ed evitare raggiri [4].

1.3 L'importanza dei *feedback* nel *web*

Gli utenti durante la loro navigazione web possono rilasciare dei commenti oppure delle opinioni personali (i cosiddetti "*feedback*") riguardanti un qualsiasi contesto argomentativo, presente sul sito *web* che stanno visitando, oppure offerto dall'applicazione *web* che stanno utilizzando. Gli sviluppatori di siti *web*, i ricercatori aziendali, gli studiosi, ecc., utilizzano queste informazioni rilasciate dagli utenti per formulare delle recensioni, dei sondaggi, dei questionari e delle statistiche sugli argomenti oggetto dei *feedback*.

Un *feedback* può confermare, smentire o suggerire quali aspetti di un determinato prodotto visitato sono buoni, quelli che sono pessimi o totalmente insignificanti, o peggio controproducenti.

1.4 L'idea di uno strumento informatico per la raccolta dei *web feedback*

Un meccanismo di *feedback* nella pratica, ad esempio, può essere utilizzato allo scopo di ottenere informazioni rilevanti dai fruitori di *internet*, raccoglierle, archiviarle e dunque renderle disponibili a coloro che intendono studiarle per le proprie attività di ricerca.

Con il presente lavoro di tesi il mio intento è quello di fornire uno strumento informatico che consenta di acquisire e conservare le suddette informazioni.

L'obiettivo di questo lavoro di tesi è progettare e sviluppare un *plug-in* sul *browser Firefox* che raccolga dei particolari *feedback* dalla piattaforma *web Youtube* e, realizzare un *database* centralizzato che ne consenta la memorizzazione e la consultazione.

Il *plug-in* è stato progettato e implementato per consentire agli utenti di esprimere uno stato d'animo quando visualizzano un qualsiasi video su *Youtube*. Mentre il *database* centralizzato, residente su un *server* autonomo, è stato creato per raccogliere i dati in questione.

Questi risultati saranno utilizzati per la definizione di un database di video, associato alle corrispondenti emozioni. Tale database verrà utilizzato dai Ricercatori in Psicologia della Seconda Università di Napoli per indurre delle emozioni e studiare i loro effetti.

Il *plug-in* sviluppato è facilmente modulabile ed adattabile per altri scopi (ad esempio per altri siti o per altri tipi di *feedback*).

1.5 Organizzazione della tesi

Nel secondo capitolo dell'elaborato saranno illustrati due modelli di architettura: quello *client/server* e quello *client/proxy/server* evidenziando i vantaggi e gli svantaggi di entrambi. Dopodiché, saranno chiarite le motivazioni a suffragio della scelta di adottare, per la costruzione dello strumento informatico, il modello *client/server*.

Nel terzo capitolo saranno descritte le tecnologie che sono state utilizzate per lo sviluppo del *plug-in* (lato *client*) e per la realizzazione del *database* centralizzato (lato *server*). Si mostreranno dapprima le tecnologie impiegate in maniera trasversale e poi in particolare, saranno analizzate quelle da utilizzare lato client e quelle da adoperare lato server.

Nel quarto capitolo sarà descritta la progettazione e l'implementazione del *plug-in*, denominato "*youtube-emotion*", e come lo si utilizza durante la navigazione *web*. Sarà inoltre descritto come viene installato sul *browser Firefox*. In aggiunta, saranno illustrati gli strumenti per il *testing* dell'applicazione realizzata e come sia possibile scaricare sul proprio computer il *report* delle informazioni, oggetto di studio degli Psicologi della Seconda Università di Napoli, memorizzate nel *database*.

Nel quinto capitolo saranno infine tratte alcune conclusioni sul lavoro di tesi svolto e i possibili sviluppi futuri.

Capitolo 2

Le architetture di riferimento

In questo capitolo saranno esposti in maniera dettagliata due differenti tipi di architettura: il modello *client/server* e quello *client/proxy/server*.

Si descriveranno i vantaggi e gli svantaggi di entrambe le architetture e si motiverà la scelta di adottare il modello *client/server* piuttosto che quello *client/proxy/server*.

2.1 Modello *client/server*

Questo tipo di modello può essere considerato come una formalizzazione del paradigma di accesso ai servizi utilizzabili in rete che si basa su un meccanismo di domanda e risposta. Supponiamo che in rete ci sia un entità che richiede un particolare servizio e che un'altra entità risponde alla prima offrendo tale servizio. Ottenere la suddetta risposta corrisponde a beneficiare del servizio richiesto.

L'architettura *client/server* presuppone l'esistenza di due entità: una che offre un servizio e l'altra che lo richiede. Queste due entità vengono normalmente identificate rispettivamente con i nomi *server* e *client* (Figura 2.1) [5].

I dati vengono trasmessi tra *client* e *server* attraverso una rete e un determinato *Software* di comunicazione. E' opportuno suddividere un sistema *client/server* in componenti *client* e componenti *server*. In Figura 2.2 è illustrata la tipica struttura di un applicazione *client/server*.

Di seguito saranno descritti i quattro moduli che compongono la suddetta struttura. Dopodiché saranno evidenziati i moduli che rappresentano le componenti *client* e quelli che, invece, rappresentano le componenti *server*.

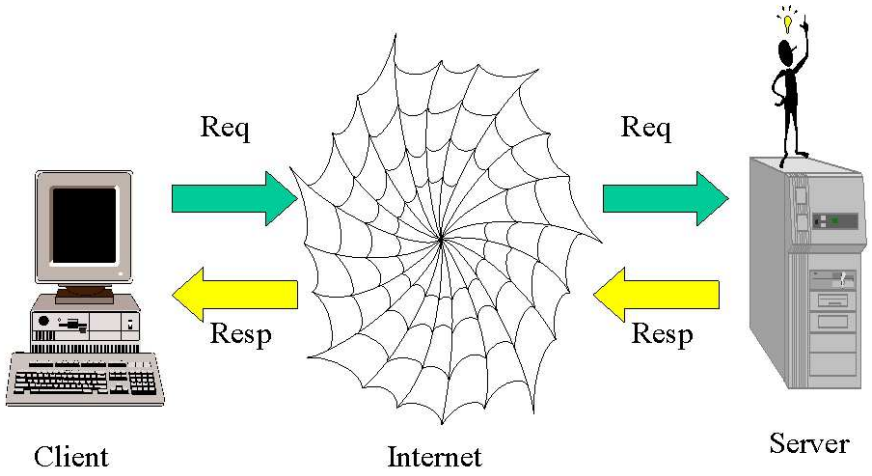


Figura 2.1: Il modello *client/server*.

Interfaccia utente	Logica applicativa	Logica di accesso ai dati	Accesso ai dati

Figura 2.2: Struttura di un'applicazione *client/server*.

Interfaccia utente Di questo modulo fanno parte tutti quei componenti applicativi che consentono all'utilizzatore del programma di interagire con esso per ottenere determinati risultati. I programmi per la posta elettronica e i *browser web* sono esempi di componenti applicativi appartenenti al corrente modulo.

Logica applicativa In questo modulo rientrano tutte quelle componenti applicative che fanno si che l'esecuzione di un programma porti a specifici risultati. Appartengono a questa categoria le componenti di calcolo, di stampa, di elaborazione di un testo o di quanto altro realizza la funzione per la quale l'utente ha richiamato in esecuzione un determinato componente

software.

Logica di accesso ai dati Si tratta, di fatto, di tutti quegli elementi che gestiscono in maniera corretta i dati forniti dall'utente al sistema o viceversa reperiti per la loro consultazione dai rapporti di archiviazione. Fanno parte della logica di accesso ai dati tutte le procedure di controllo sui dati stessi, da una semplice validazione, come può essere quella su un valore minimo o massimo, a procedure di controllo della validità dei riferimenti tra i dati.

Accesso ai dati In questo gruppo rientrano tutte le componenti che si occupano di archiviare e reperire i dati sui supporti di archiviazione del sistema. Queste operazioni sono normalmente svolte da un *DataBase Management System*(*DBMS*) dovendo essere fornite garanzie di integrità fisica dei dati, possibilità di accesso concorrente, ottimizzazione dei tempi di lettura e scrittura e indipendenza nei confronti del sistema *hardware* utilizzato.

Risulta intuitivo, dalle precedenti descrizioni, che la suddivisone ideale di un sistema *client/server* in componenti *client* e componenti *server* sia quella illustrata in Figura 2.3.

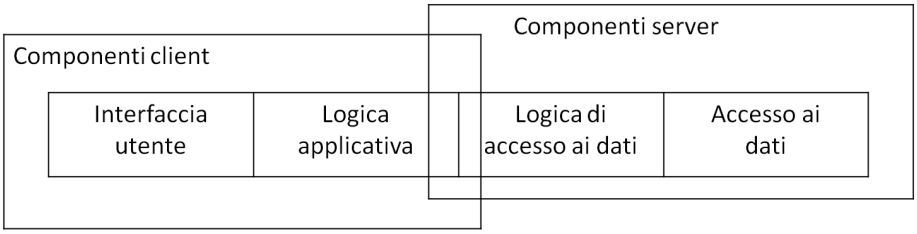


Figura 2.3: Suddivisione di un applicazione *client/server*.

L'intersezione tra componenti *client* e componenti *server* indica la forte difficoltà esistente in un sistema reale a definire una netta distinzione tra le due classi di componenti. Risulta infatti estremamente complesso suddividere la logica di accesso ai dati dalla logica applicativa per molte delle applicazioni di un sistema reale. Normalmente è necessario che alcuni di questi componenti vengano replicati sia sul lato *client* che su quello *server* per poter garantire al tempo stesso integrità nei dati ed interfacce utenti efficaci.

Un caso significativo è quello tipico sui controlli di validità del singolo dato: sebbene debba essere sicuramente implementato un meccanismo che protegga il *database* dall'accettare valori incongruenti, è altrettanto vero che

una buona interfaccia utente dovrà intercettare possibili errori di inserimento dati se non guidare l'utente verso l'inserimento di dati corretti. A questo scopo la logica di controllo dovrà essere parzialmente duplicata su vari componenti dell'applicazione. Ad evitare inconsistenze, i controlli sull'integrità del dato dovranno sempre essere implementati sul lato *server*; tuttavia una buona logica applicativa dovrà comprendere dei meccanismi per acquisire in maniera automatica e dinamica dal *server* le informazioni relative ai vincoli di integrità del dato stesso per poterli riprodurre in locale, possibilmente passando tali informazioni ai componenti di interfaccia progettati per guidare l'utente nella fase di immissione dei dati.

Sebbene l'ultima parola in fatto di controlli spetti al *server*, sarebbe di fatto assurdo che nessun controllo fosse fatto a livello dell'applicazione *client*. Avremmo in questo caso un'elevata quantità di dati rifiutati durante le fasi di inserimento e modifica e di fatto caricheremmo l'utente dell'onere di conoscere tutti i vincoli definiti all'interno dello schema del *database*. La suddivisione in componenti *client* e *server* illustrata si collega anche alla necessità di un moderno *database* di poter gestire vincoli di integrità logica sui dati oltre a quelli di integrità fisica [6].

I *DBMS* appositamente progettati, o comunque profondamente rivisti nel contesto di architetture *client/server*, vengono in certi casi definiti estesi proprio per sottolineare la presenza di meccanismi idonei a definire vincoli di integrità dei dati in essi archiviati. Parte di questi meccanismi sono in realtà implementati anche nei *database* commerciali per sistemi centralizzati. In particolare la struttura stessa del sistema relazionale aveva portato i progettisti di questi prodotti, già nei primi anni del loro utilizzo, alla ricerca di meccanismi per mantenere integri i vincoli di riferimento tra le varie tabelle del *database*.

Tuttavia non era apparso chiaro, o perlomeno rilevante che i vincoli di integrità referenziale sono solo una parte, seppure significativa, della classe più ampia dei vincoli di integrità logica dei dati. Tale visione era certamente condizionata da un approccio nello sviluppo dei sistemi informativi orientato più ai programmi che non ai dati, tipico del periodo antecedente all'uso dei *DBMS*. Questo modo di pensare è nato dalla mancata percezione della necessità di indipendenza tra il dato e l'applicazione e ha portato a considerare i vincoli di integrità sui dati come parte della logica applicativa, da implementare quindi come controlli da inserire in un programma. Le caratteristiche di distribuzione e maggiore flessibilità del sistema *client/server* fanno sì che un dato venga normalmente visto come patrimonio comune di molte applicazioni, rendendo di fatto impossibile un approccio di controllo sulla validità del dato stesso implementata su ogni singolo componente

software. Si rendono quindi necessari meccanismi a livello di *DBMS* per la definizione a livello centralizzato, come parte integrante del dato, dei necessari vincoli di integrità logica [7].

Facendo riferimento alla definizione classica che definisce un'informazione come un dato integrato dalle conoscenze necessarie per interpretarlo, potremmo dire che i *DBMS* stessi sono più dei gestori di informazioni che gestori di dati. Secondo lo schema già visto potremmo affermare che in un *DBMS* esteso esistono dei meccanismi che implementano l'approccio della Figura 2.4.

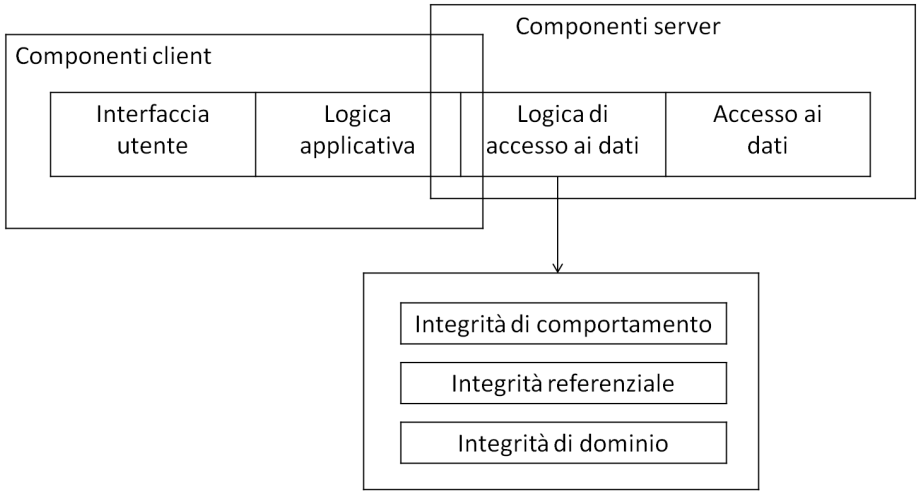


Figura 2.4: Logica di accesso dati ed integrità semantica.

La logica di accesso ai dati, definita come componente chiave di un sistema *client/server*, coincide infatti con i meccanismi definiti per garantire l'integrità dei dati sui *DBMS*. A questo livello, quindi, viene gestita l'integrità referenziale, cioè viene gestito l'insieme di regole utilizzate per assicurare che le relazioni tra i record delle tabelle correlate siano valide e non vengano eliminati o modificati per errore dati correlati; viene inoltre garantita l'integrità di dominio, cioè si controlla che i dati che si vogliono inserire nel *database* appartengano al loro dominio di definizione (cioè se appartengono ad un insieme specifico di valori, definito al momento della creazione delle tabelle del *database* in cui devono essere inseriti). Infine vengono effettuati anche i controlli per garantire l'integrità di comportamento, argomento considerato di particolare interesse per i sistemi informativi moderni, connesso alla visione del dato come correlato a particolari regole di comportamento

(ad esempio, nell'inserimento di un ordine è opportuno assicurarsi che la sua data non sia antecedente ad ordini precedentemente emessi) [8].

L'implementazione di questi vincoli può avvenire sia sfruttando la sintassi standard *SQL* (*Structured Query Language*), che ha però il forte limite nella necessità di basarsi su definizioni espresse al momento della creazione delle strutture dati direttamente sulle colonne o nelle tabelle del *database*, sia con un approccio di tipo procedurale, che consente di realizzare moduli *software* direttamente all'interno del *DBMS* per effettuare operazioni di controllo su determinati eventi del sistema. Il meccanismo principale di gestione dell'integrità logica dei dati tramite la tecnica procedurale è quello del *trigger*, un insieme di istruzioni che vengono eseguite non ad una richiesta specifica dell'utente ma al determinarsi di particolari eventi [9].

In un sistema *client/server* sia l'applicazione *client* che il *server* di *database* collaborano a sostenere il carico di elaborazione di un'applicazione (da cui il termine "elaborazione distribuita"). Si precisa che il *server* gestisce il *database* tra diversi *client*, mentre i *client* inviano, richiedono e analizzano i dati che ricevono dal *server*.

2.1.1 I vantaggi del modello *client/server*

Uno dei vantaggi dell'architettura *client/server* è rappresentato dalla sicurezza dei dati che viene gestita a livello centralizzato.

Prima che un *client* possa accedere alle risorse offerte da un *server*, l'utente deve essere identificato e autenticato in maniera che possa accedere a tali risorse. Grazie alla centralizzazione degli *account* utente, della sicurezza e del controllo degli accessi si semplifica il lavoro dell'amministrazione di rete.

Un altro vantaggio riguarda la semplicità di effettuare il *backup* dei dati. Infatti i dati vengono memorizzati solo ed esclusivamente sul *server* centrale rendendo il *backup* molto più comodo da gestire.

Tale modello, essendo un'architettura distribuita, può utilizzare efficacemente sistemi collegati in rete con molti processori distribuiti, inoltre, è semplice aggiungere un nuovo *server* e integrarlo con il resto del sistema senza influenzare altre parti del sistema.

Un *server* di *database* è intelligente, blocca e restituisce solo le righe che un *client* richiede, il che garantisce la concorrenza, minimizza il traffico di rete e migliora la *performance*.

2.1.2 Gli svantaggi del modello *client/server*

Spostando completamente il lavoro sul *server*, è necessario che esso sia in grado di sopportare totalmente il carico applicativo. Quindi si ha la necessità di avere un *server* con caratteristiche più avanzate e questo costituisce uno svantaggio.

Lo stesso *server* può diventare un *single point of failure* (punto critico per i guasti) del sistema e può causare problemi di funzionamento per la rete. Quindi se il *server* smette di funzionare, anche la rete diventa inutilizzabile fino a quando il *server* non torna nuovamente a funzionare. Inoltre l'*hardware* e il *software* del *server* hanno un costo sostanzialmente alto [10].

2.2 Modello *client/proxy/server*

In questo particolare tipo di modello un *server proxy* (detto anche "server mandatario") non è altro che un terminale che svolge la funzione di intermediario tra i computer di una rete locale e *internet*.

Risulta abbastanza semplice descrivere il funzionamento di tale modello di architettura: quando un utente si connette a *internet* tramite un'applicazione *client* configurata per usare un *server proxy*, questa si connetterà in primo luogo al *server proxy* e gli darà la sua richiesta. Il *server proxy* si connetterà allora al *server* che l'applicazione *client* cerca di raggiungere e gli trasmetterà la sua richiesta. Il *server* risponderà in seguito al *proxy*, che a sua volta trasmetterà la risposta all'applicazione *client* come illustrato in Figura 2.5 [11].

2.2.1 I vantaggi del modello *client/proxy/server*

Il *server proxy* garantisce il *caching* dei dati cioè la capacità di mantenere in memoria le pagine *web* visitate più di frequente dagli utenti della rete locale per poterle fornire il più rapidamente possibile. Questa sua importantissima funzione costituisce un enorme vantaggio.

Inoltre, poiché è il *server proxy* a scaricare il *file* dal sito originale, questa operazione avverrà alla massima velocità possibile, indipendentemente dalla velocità del *modem* con cui l'utente è collegato. In tal modo l'utente risentirà meno di eventuali brevi interruzioni nello scaricamento del *file* dal sito originale, perché il *server proxy* svolge la funzione di "*buffer*" del trasferimento.

Il risultato è quindi una navigazione più fluida ed uno sfruttamento ottimale della propria velocità di connessione.

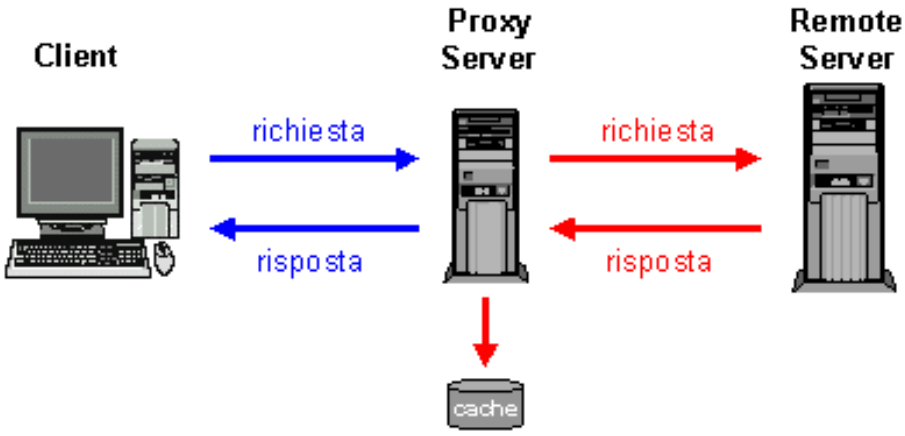


Figura 2.5: Funzionamento del modello *client/proxy/server*.

2.2.2 Gli svantaggi del modello *client/proxy/server*

Alcuni *proxy* non permettono la gestione autonoma da parte dell'utente di *cookies*, *referrer* e *javascript*. Se il *proxy* non permette la gestione di queste impostazioni, alcuni siti potrebbero dare problemi di accesso e visualizzazione delle pagine.

Un *proxy server* consentirà un maggior accesso dall'interno della rete protetta verso l'esterno, ma manterrà l'interno completamente inaccessibile dall'esterno. Ciò implica l'impossibilità di avere connessioni *server* oppure *mail* dirette verso i computer presenti all'interno.

Inoltre, i *proxy server* sono lenti. A causa del sovraccarico maggiore, quasi ogni altro mezzo per ottenere questo accesso sarà più veloce [12].

2.3 L'architettura proposta

Alla luce delle architetture poc'anzi esaminate, tenendo bene in considerazione l'obiettivo del presente elaborato espressamente indicato al paragrafo 1.4, si è scelto di adottare il modello più semplice che presenta più vantaggi: quello *client/server*.

Se utilizzassimo il modello *client/proxy/server* il *proxy* potrebbe generare numerosi colli di bottiglia e rallentare i tempi di richiesta *client-server* nonchè i tempi di risposta *server-client*.

Inoltre, qualora si verificasse un guasto al *proxy* verrebbe compromessa la comunicazione tra il *client* e il *server*. I problemi appena delineati rafforzano,

pertanto, l'idea di adottare il modello *client/server* piuttosto che quello *client/proxy/server*citebib13.

Nel modello scelto e adoperato per la realizzazione dello strumento informatico oggetto del nostro lavoro di tesi, la componente lato *client* è il *browser Firefox* su cui è installato il *plug-in* "*youtube-emotion*" mentre il *database* centralizzato risiede nella componente lato *server* (Figura 2.6).

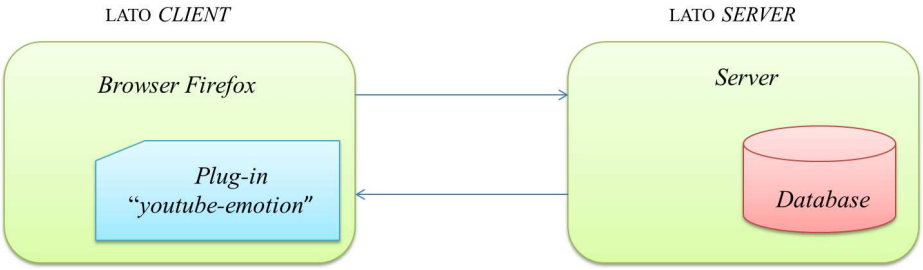


Figura 2.6: Rappresentazione del modello *client/server*.

In realtà il *plug-in* "*youtube-emotion*" comunica anche con il *server* utilizzato dalla piattaforma *web Youtube* per richiedere delle determinate informazioni. Rimandiamo tale dettaglio al quarto capitolo.

Capitolo 3

Tecnologie utilizzate

In questo capitolo saranno descritte le tecnologie (ambienti di sviluppo e linguaggi di programmazione) che sono state adoperate per la progettazione e l'implementazione del *plug-in* "*youtube-emotion*" oltre a quelle necessarie alla realizzazione del *database* centralizzato.

Prima di esporre in maniera dettagliata tale classificazione si descriveranno linguaggi di programmazione e le tecniche che sono stati utilizzati su entrambe le componenti (in maniera trasversale).

Successivamente saranno esposte le tecnologie impiegate sulla componente lato *client* (il *plug-in*) e quelle adottate sulla componente lato *server* (il *server* dove risiede il *database*).

Il linguaggio *HTML*

HTML (*HyperText Markup Language*) è il linguaggio di *markup* per ipertesti predominante nelle pagine *web*. La sua sintassi è stabilita dal *W3C*.

E' stato sviluppato alla fine degli '80 da *Tim Berners-Lee* al *CERN* (Organizzazione Europea per la Ricerca Nucleare) di Ginevra assieme al noto protocollo *HTTP* che supporta invece il trasferimento di documenti in tale formato. Verso il 1994 ha avuto una forte diffusione in seguito ai primi utilizzi commerciali del *web*.

Nel corso degli anni ha subito molte modifiche e miglioramenti raggiungendo oggi la versione 4.01, rilasciata il 24 dicembre 1999. Dal 2007 il *W3C* ha ripreso l'attiva di specifica con la definizione, ancora in corso, di *HTML* 5.

Caratteristica di *HTML* è che esso è stato concepito per definire il contenuto logico e non l'aspetto finale del documento. Attualmente i documenti *HTML* sono in grado di incorporare molte tecnologie, che offrono la pos-

sibilità di aggiungere al documento ipertestuale effetti grafici, interazioni dinamiche con l'utente, animazioni interattive e contenuti multimediali [14].

Il linguaggio *CSS*

Il *CSS* (*Cascading Style Sheets* o Fogli di stile) è un linguaggio usato per definire la formattazione di documenti *HTML*, *XHTML* e *XML*. Le regole per comporre il *CSS* sono contenute in un insieme di direttive emanate a partire dal 1996 dal *W3C*.

L'introduzione del *CSS* si è resa necessaria per separare i contenuti dalla formattazione, permettendo così una gestione separata della presentazione dai contenuti veri e proprio, ricordando che, come specificato sopra, *HTML* era nato solo per descrivere il contenuto logico e non l'aspetto finale del documento.

Ciò permette di presentare lo stesso contenuto in differenti modi, permettendo di scegliere un diverso stile in base ai bisogni specifici e ai dispositivi utilizzati dagli utenti. L'ultima revisione relativa alla versione 2.1 è stata effettuata nel giugno scorso (la direttiva 2.0 risale invece al 1999). Attualmente la versione 3 è in via di sviluppo [15].

Il linguaggio *Javascript*

Javascript è un linguaggio di programmazione interpretato e leggero, che dispone di funzionalità orientate agli oggetti. Fù originariamente sviluppato da *Brendan Eich* della *Netscape Communications* con il nome di *Mocha* e successivamente di *LiveScript*, ma in seguito è stato rinominato "*JavaScript*" ed è stato formalizzato con una sintassi più vicina a quella del linguaggio *Java* di *Oracle*.

JavaScript è stato standardizzato per la prima volta tra il 1997 e il 1999 dalla *ECMA* con il nome *ECMAScript*. Il nucleo del linguaggio è stato incorporato all'interno di *Netscape*, di *Internet Explorer* e degli altri *browser web* più diffusi. Inoltre è stato arrchito con funzionalità che lo hanno reso adatto all'editoria *web*: sono stati aggiunti oggetti rappresentativi della finestra di navigazione e del suo contenuto.

La versione di *Javascript* lato *client* permette l'inserimento di contenuti eseguibili all'interno di pagine *web*; così le pagine *web* non sono più pagine *HTML* statiche, ma possono comprendere programmi che interagiscono con l'utente, controllando il *browser* e creando nuovi contenuti *HTML*.

Sintatticamente, il linguaggio di base *Javascript* somiglia al *C*, al *C++* e a *Java*. Dispone di costrutti di programmazione come l'istruzione *if*, il

ciclo *while* e l'operatore *&&*. Le somiglianze, purtroppo, si fermano qui. *Javascript* è un linguaggio debolmente tipizzato: le variabili non devono nesessariamente essere dichiarate con un tipo [16].

La tecnologia *AJAX*

AJAX, acronimo di *Asynchronous JavaScript and XML*, è una tecnica di *web development* per creare più facilmente e più velocemente applicazioni *web* interattive (*Rich Internet Application*). Lo scopo è quello di rendere le pagine *web* più reattive scambiando (in *background*) piccole quantità di dati con il *server* di modo che la pagina *web* non debba essere completamente ricaricata ogni volta che l'utente richiede un cambiamento.

Infine sembra interessante far notare che *Ajax* non è una singola tecnologia, ma un'insieme di tecnologie. *HTML* (o *XHTML*) e *CSS* vengono usati per il *markup* e lo stile. Il *DOM* (*Document Object Model*), manipolato attraverso un linguaggio *ECMAScript* come *JavaScript* o *JScript*, viene usato per mostrare le informazioni ed interagirvi. L'oggetto *XMLHttpRequest*, invece, viene usato per l'interscambio asincrono dei dati tra il *browser* dell'utente e il *web server*.

In alcuni framework *Ajax* e in certe situazioni, può essere usato un oggetto *Iframe* invece di *XMLHttpRequest* per scambiare i dati con il *server* e, in altre implementazioni, *tag* <*script*> aggiunti dinamicamente (*JSON*); in genere viene usato *XML* come formato di scambio dei dati, anche se di fatto qualunque formato può essere utilizzato, incluso testo semplice, *HTML* preformattato, *JSON* e perfino *EBML*. Questi file sono solitamente generati dinamicamente da *script* lato *server*.

Nelle applicazioni *web* tradizionali gli utenti compilano ed inviano i moduli ad un *web server*. Il *web server* risponde alle richieste degli utenti bloccando o mostrando una nuova pagina. Gli utenti, però, rimangono in attesa che il *web server* elabori i moduli e risponda alle loro richieste.

Nelle applicazioni basate su *Ajax* invece, il motore *Ajax* fà da intermediario tra l'utente e il *web server*. Questo motore, scritto in *Javascript*, permette che l'interazione dell'utente con l'applicazione avvenga in modo asincrono, cioè indipendentemente dalla comunicazione con il *server*. In questo caso, gli utenti non devono aspettare che il *web server* finisca l'intera elaborazione e restituisca tutta la pagina. In Figura 3.1 è mostrato il confronto tra i due tipi di applicazione appena descritti [17].

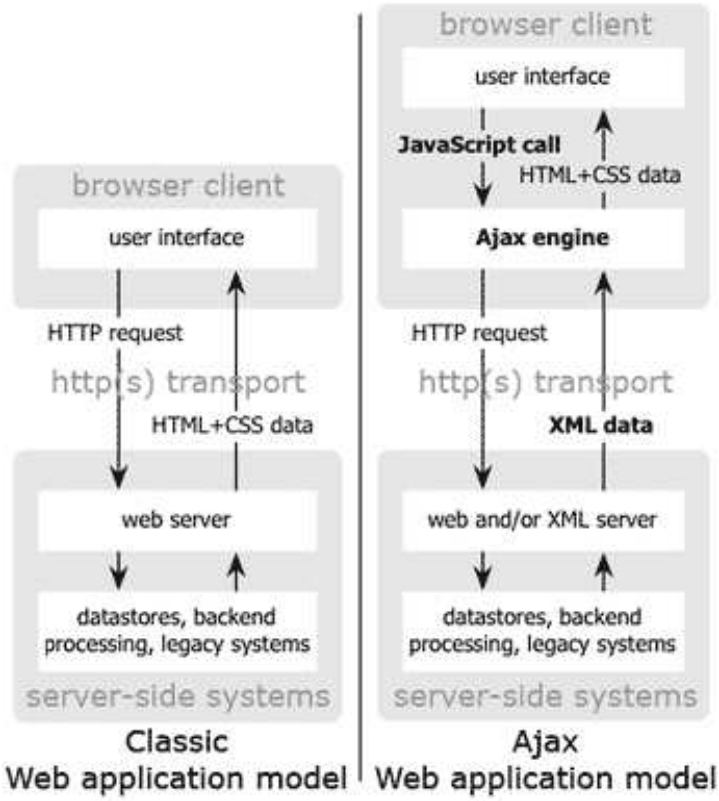


Figura 3.1: Confronto tra il modello di interazione tradizionale delle applicazioni *web* e il modello *AJAX*.

3.1 Il *client*

In questo modulo ricopre un ruolo molto importante l'utilizzo dell'*Add-on SDK* come strumento per creare *plug-in* su *Firefox* utilizzando le tecnologie descritte in precedenza. Infatti, tale strumento è stato proprio adoperato per sviluppare il *plug-in* "*youtube-emotion*" che nel modello di architettura scelto, quello *client/server*, rappresenta il componente *client*.

3.1.1 *Add-on SDK*

L'*add-on* è un pacchetto *software* che aggiunge nuove funzionalità alle applicazioni di *Mozilla*, tra cui sono presenti anche i *plug-in* per *Firefox*. L'*add-on SDK* include: un set di moduli che forniscono *API Javascript* che possono

essere usate per creare *add-ons*; un insieme di *tools* per creare, eseguire, testare e fare il *packaging* dell'*add-on* in un ambiante sicuro.

Quest'ultimo punto offre un'enorme vantaggio agli sviluppatori di *plug-in* perché è possibile eseguire il *plug-in* su un'istanza del *browser* creata appositamente per il *testing*. Inoltre, è anche presente un buon sistema di controllo e notifica degli errori. Il sistema succitato si affianca ad un opzione presente nel *software* che consente di creare i propri *log* in maniera davvero semplice. Questi metodi vengono forniti attraverso due strumenti: *console* e *cfx*.

Il primo è un oggetto che permette di registrate messaggi utili per avere il *log* delle azioni che il *plug-in* compie. Il secondo è uno strumento a linea di comando che consente di accedere alla documentazione dell'*SDK* e permette di eseguire, testare, e costruire il pacchetto che andrà poi eseguito sul *browser*.

Per ciò che attiene il primo punto, invece, sono presenti vari moduli che forniscono molti dei servizi solitamente usati nella creazione di *plug-in*. Alcuni di questi moduli sono: *clipboard* che consente ai "chiamanti" di interagire con il sistema di *clipboard*, impostando e recuperando il suo contenuto; *context-menu* che permette di aggiungere elementi al menu contestuale nella pagina; *hotkeys* che permette di creare "*keyboard shortcut*", ovvero tasti di scelta rapida, per certe operazioni; *notifications* che permette di visualizzare messaggi transitori all'utente; *page-mod* che permette agli sviluppatori di eseguire *script* nel contesto di una specifica pagina, potendola quindi modificare dinamicamente; *page-worker* che fornisce il meccanismo per creare una permanente, invisibile pagina e accedere ad essa attraverso il suo *DOM*; *panel* che crea un finestra di dialogo *pop-up* che appare sopra i contenuti *web* ed è persistente fino a che non viene dismessa dagli utenti o dai programmi; *passwords* che permette all'*add-on* di interagire con il "*Password Manage*" di *Firefox* per aggiungere, richiamare, rimuovere le credenziali memorizzate; *private-browsing* che permette di accedere alla modalità di navigazione privata di *Firefox*, individuando se è attiva e quando avvengono cambiamenti di stato; *request* che consente di effettuare richieste di rete in modo semplice ma potente; *selection* che fornisce un meccanismo per ottenere ed impostare il testo e le selezioni *HTML* nella pagina corrente; *self* che fornisce l'accesso ai dati interni all'*add-on* stesso e fornisce l'accesso al "*Program ID*", valore unico per ogni componente aggiuntivo; *simple-storage* che consente di memorizzare i dati in modo persistente; *tabs* che fornisce un facile accesso alle schede e ai loro eventi; *timers* che consente di accedere alle funzionalità *web* di temporizzazione; *widget* che permette di accedere alle funzionalità dell'*add-on* attraverso una semplice interfaccia utente; *windows* che forni-

sce un facile accesso alle finestre del *browser*, le loro schede e alle correlate funzioni di apertura e chiusura [18].

Firefox si sta muovendo verso un modello in cui vengono usati processi separati per visualizzare l'interfaccia utente, gestire i contenuti *web* ed eseguire i componenti aggiuntivi. Il principale codice *add-on* (chiamato anche "*add-on code*") verrà eseguito nel processo di *add-on* e non avrà accesso in maniera diretta ai contenuti *web*. Ciò significa che un componente aggiuntivo che deve interagire con i contenuti *web* deve essere strutturato in due parti: lo *script* principale viene eseguito nel processo di *add-on*; qualsiasi codice che ha bisogno di interagire con i contenuti *web* viene caricato nel processo dei contenuti *web* come uno *script* separato. Questi *script* separati sono chiamati *content script*. Alla luce di quanto appena descritto, per creare un *add-on*, dunque, si fa uso di due differenti tipi di *script*: l'*add-on code* e il *content script*. Entrambi utilizzano il linguaggio *JavaScript* ma si differenziano per lo scopo per il quale sono scritti, le cartelle in cui vanno inseriti e gli insiemi di *API* alle quali possono accedere. Nelle due sottosezioni seguenti, i suddetti *script* saranno descritti in maniera più approfondita. Dopodichè, in un'altra sottosezione, verrà fatto un confronto tra i due tipi di *script* [19].

Add-on code

L'*Add-on cod*e è il codice che viene usato per creare e sviluppare la logica principale del *plug-in*. Viene implementato tramite una collezione di uno o più moduli *CommonJS*. Ogni modulo è rappresentato tramite uno *script* memorizzato nella cartella "*lib*". In questa cartella si trova il *file* più importante che costituisce il cuore del componente aggiuntivo: il *file* "*main.js*". Da questo *file*, poi, si possono richiamare altri moduli tramite la funzione "*require()*".

CommonJS, invece, è l'infrastruttura sottostante sia dei moduli dell'*add-on SDK* che del *plug-in* stesso. Esso definisce le specifiche per i moduli e i *package* (insiemi di moduli).

Moduli *CommonJS* Un modulo *CommonJS* è un pezzo di codice *JavaScript* riutilizzabile. Infatti permette di esportare alcuni oggetti che possono così essere utilizzati da codice esterno. Per farlo definisce: un oggetto chiamato *exports* che contiene tutti gli oggetti che un modulo *CommonJS* vuole rendere disponibili ad altri moduli; una funzione chiamata *require* attraverso la quale un modulo può importare gli oggetti *exports* degli altri moduli. La Figura 3.2 rende molto più chiara l'interazione tra gli oggetti *exports* e la

funzione *require*. L'*SDK* congela gli oggetti *exports* restituiti dall'invocazio-

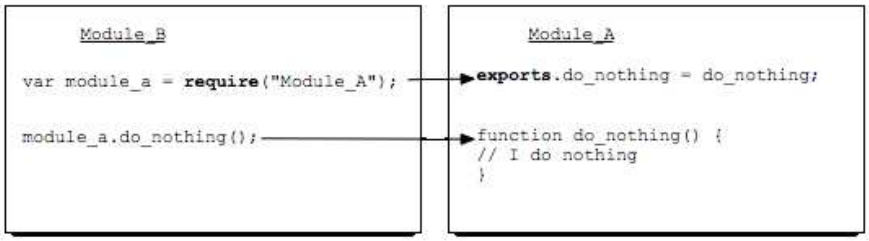


Figura 3.2: Esempio di utilizzo dei moduli *CommonJS*.

ne della funzione require in modo tale da non permettere di modificare le proprietà di tali oggetti [20].

Package CommonJS Un *package CommonJS* è una struttura che racchiude una collezione di moduli e permette di distribuire, installare e gestire in modo più semplice i moduli. I moduli di un *add-on* costituiscono un *package* che prende il nome del componente stesso. L'*add-on SDK* è formato da due *package*: l'*add-on kit* e le *api-utils*. Il primo fornisce *API* di alto livello che possono essere utilizzate dagli sviluppatori per creare in modo semplificato i loro *plug-in*; sono contenuti in esso tutti i moduli che permettono l'interazione diretta con il *browser*: *clipboard*, *context-menu*, *hotkeys*, *notifications*, *page-mod*, *page-worker*, *panel*, *passwords*, *request*, *private-browsing*, *selection*, *self*, *simple-storage*, *tabs*, *timers*, *widget*, *windows* (dei quali è stata fornita una breve descrizione in 3.1.1) e tanti altri. Il secondo fornisce un'infrastruttura *CommonJS* di base per sviluppare *add-on* e applicazioni *XULRunner*; costituisce la base per l'*add-on SDK*. I moduli dei vari *package* possono comunicare tra di loro sfruttando il meccanismo descritto prima, degli oggetti *exports* e delle funzioni *require*, come nell'esempio di figura 3.3 dove il *package* relativo ad un certo *add-o*n (chiamato, ad esempio, *translator*) comunica con il *package addon-kit*. Un *package* deve sempre contenere un file che lo descrive chiamato: "*package.json*" in cui inserire l'autore, la descrizione, le dipendenze, ecc. *JSON* è l'acronimo di *JavaScript Object Notation*, un formato facilmente comprensibile per gli umani e creabile/analizzabile per le macchine. Inizia e finisce con le parentesi graffe, ed è formato da coppie "nome" : "valore" separate dalla virgola. Per esempio il contenuto del file "*package.json*" di un *add-on* può essere quello in Figura 3.4 [21].

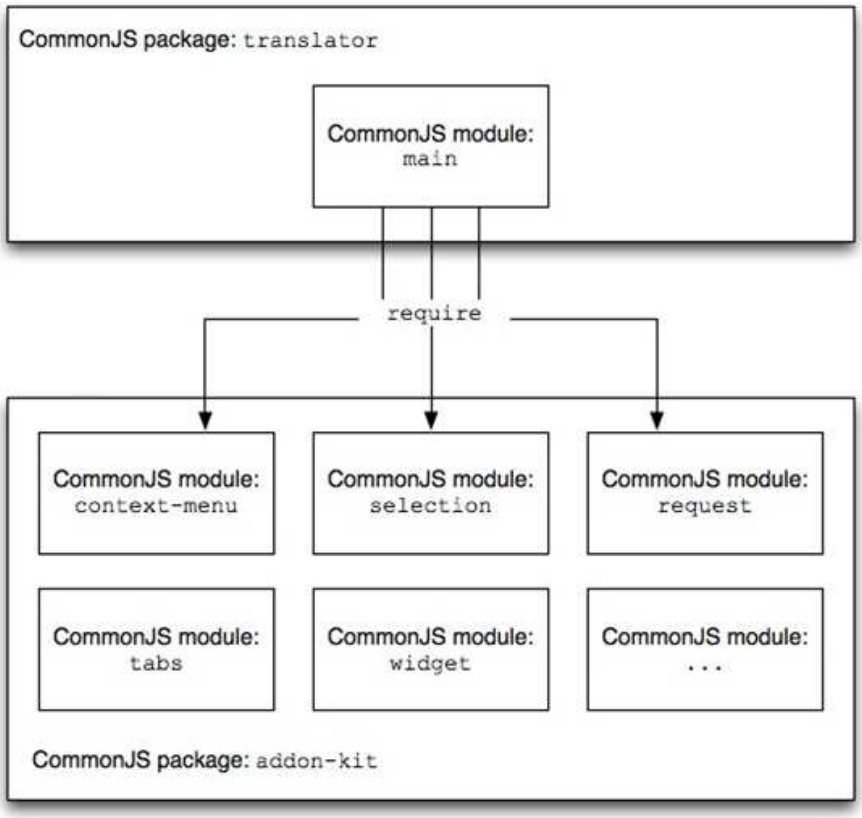


Figura 3.3: Dipendenze tra il *package* dell'*add-on translator* e il *package addon-kit* dell'*SDK*.

Content script

I *content script* sono file *JavaScript* che permettono di interagire con le pagine *web*, e consentono di manipolare e modicare il contenuto di tali pagine. Questi *script* vengono introdotti nelle pagine *web* usando *API* definite in alcuni moduli dell'*SDK* come *page-mod*, *panel* e *widget*.

In realtà questi *script* possono essere inseriti in due modi differenti: tramite stringhe letterali da inserire dove occorre nel *file* "*main.js*"; mantenuti in *file* separati e referenziati attraverso il loro nome e posizione nella cartella "*data*".

Se il *content script* è memorizzato in un *file* separato, si può anche utilizzare qualsiasi altra libreria *JavaScript*, come *JQuery*, che andrà inserita

```
{
    "name": "new-addon",
    "fullName": "New Add-on",
    "description": "This is an example of addon description.",
    "author": "Name Surname",
    "license": "MPL",
    "version": "0.1"
}
```

Figura 3.4: Esempio di codice contenuto del *file* "*package.json*".

nella cartella "*data*". I file *JavaScript* del *content script* possono interagire solo con il modulo *main* col quale possono scambiarsi informazioni attraverso specifici eventi. Un *add-on* deve sempre avere il *file main.js* nella cartella "*lib*", mentre non è detto che abbia bisogno di *content script*.

Add-on code vs Conten script

In precedenza, abbiamo accennato al fatto che i due tipi di *script* si differenziano anche per le *API* a cui possono accedere, oltre che per le finalità e le cartelle in cui vengono inseriti. La Figura 3.5 riassume quali *API* possono essere disponibili nei differenti *script* [22].

La Figura 3.6 mostra una panoramica dei componenti principali e le loro relazioni. La parte colorata di grigio rappresenta il codice scritto dallo sviluppatore dell'*add-on*.

Struttura

Come introdotto in precedenza, tutti i moduli dell'*add-on* forniti nella cartella "*lib*" creano un *package*, che prende il nome del *plug-in* stesso. Questo *package* comunica con i *package* dell'*SDK* come si può visualizzare in Figura 3.7.

Scheletro Il progetto *add-on SDK* mette a disposizione vari strumenti (come visto nel dettaglio in 3.1.1). Attraverso "*cfx*" si può inizializzare la struttura del *plug-in*; per farlo basta eseguire da linea di comando "*cfx init*". Questo comando va eseguito dopo aver attivato "*cfx*" (sempre da linea di comando) e aver creato la cartella che andrà a contenere tutto il *plug-in*. Lo scheletro creato da tale comando ha la struttura di *file* in Figura

API	Add-on code	Content script
Oggetti globali definiti nel cuore del linguaggio *JavaScript*, come *Math*, *Array* e *JSON*.	✓	✗
La funzione globale *require*() e gli oggetti globali *exports* definiti nei moduli *CommonJS*. Da ciò dipende anche l'utilizzo degli oggetti *exports* forniti dai moduli della *SDK*.	✓	✗
La globale console fornita dall'*SDK*.	✓	✓
Le globali definite da *HTML* 5, come *windows*, *document* e *localStorage*.	✗	✓
La globale *self*, usata per comunicare tra *content script* e *add-on code*.	✗	✓

Figura 3.5: Differenze nella disponibilità delle *API* tra *content script* e *add-on code*

3.8: Ogni cartella ha uno scopo preciso. In quella denominata *data* vanno inserite varie risorse come *icone*, file *HTML* e ogni *content script*, ovvero *script* che permettono di interagire con il contenuto delle pagine *web* o con l'interfaccia del *browser*. La cartella *doc* permette di includere tutta la documentazione relativa all'*add-on* sviluppata. In *lib*, invece, vanno inseriti i moduli *JavaScript CommonJS*, ovvero gli *script* di tipo *add-on code*, che permettono di implementare le funzionalità del *plug-in*; è in quest'ultima cartella che andrà inserito il *file* "*main.js*", che è il cuore di tutta la struttura, ed eventuali altri moduli. Infine, nella cartella *test* verrà inserito il codice per effettuare gli "*unit test*". *Unit test* è un metodo tramite il quale pezzi di codice sono testati individualmente per determinare se si comportano come progettato. Il *file* "*package.json*" permette di inserire informazioni sull'*add-on*, come autore, descrizione, codice univoco (*ID*), dipendenze, ecc. Nel *file* "*README.md*" si possono inserire informazioni sul contenuto di ogni cartella interna all'*add-on*, sull'uso, i requisiti, ecc.

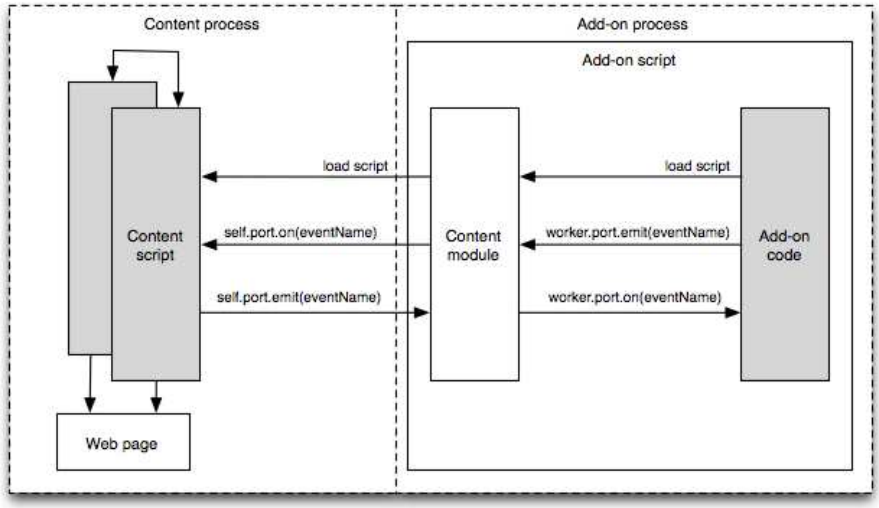


Figura 3.6: Comunicazione tra le componenti di un *add-on*.

3.2 Il *server*

Per questo modulo è stata usata la piattaforma *LAMP* (*Linux*, *Apache*, *MySQL* e *PHP*). A volte viene utilizzata in alternativa anche la definizione "*the LAMP stack*", ovvero la pila *Linux*, *Apache*, *MySQL* e *PHP* i cui ruoli dei singoli componenti sono facilmente identificabili.

Il *Linux* logicamente viene adoperato come sistema operativo. L'*Apache* viene utilizzato come *web server*. *MySQL* viene impiegato come *RDBMS* (*Relational DataBase Management System*). *PHP* come linguaggio di scripting. In Figura 3.9 è illustrato lo *stack LAMP*.

Sebbene rappresentino un riferimento, questi non sono gli unici a poter comporre una applicazione *LAMP*. Basti pensare infatti che la "P" potrebbe stare per *Python* o *Perl*, o che l'acronimo potrebbe mutare in *LAPP* (*PostgreSQL* al posto di *MySQL*), *WAMP* (*Windows* al posto di *Linux*), *MAMP* (*Macintosh* al posto di *Linux*) e altre combinazioni ancora.

Nel presente elaborato, si è scelto di adottare la variante *LAPP* preferendola alla classica piattaforma *LAMP*. Come è facile intuire da quanto appena delineato, la piattaforma *LAPP* racchiude il *database* "centralizzato" che nel modello *client/server*, insieme al *server* ove risiede, riproduce il componente *server*.

La definizione *LAMP* solitamente, si riferisce a delle piattaforme *web*

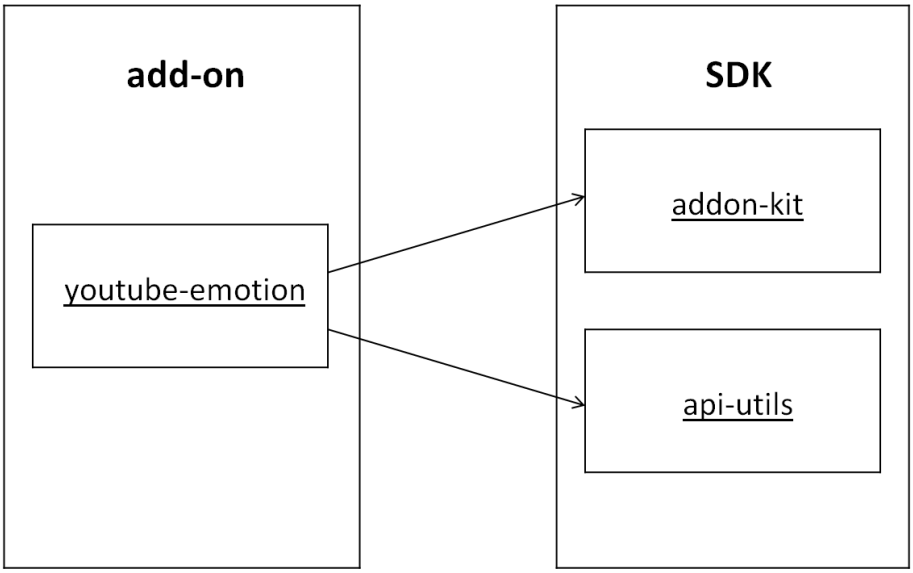


Figura 3.7: Dipendenze tra il *package* del *plug-in youtube-emotion* e i *package* dell'*SDK*.

basate su *software OpenSource*. Proprio per questo stretto legame con l'*OpenSource*, *LAMP* viene spesso adoperato al fine di evidenziare e sfruttare al massimo i vantaggi legati all'approccio *FOSS* (*Free OpenSource Software*).

Di seguito, in maniera molto sintetica, riportiamo le caratteristiche di tale approccio: *Software* liberamente scaricabile e modificabile da chiunque; Nessuna licenza o costo che ne limitano la diffusione tra gli utenti; Accesso ai codici sorgenti. In altri termini qualsiasi utente può adattare il *software* alle proprie esigenze; Veloce rilascio di *patch* e aggiornamenti grazie alle *community online*. Ancora, le piattaforme *LAMP* sono in grado di offrire ottime capacità di scalabilità e prestazioni permettendo di scindere i vari componenti su più livelli [23]. Per le motivazioni appena enucleate è stato scelto di adottare questa piattaforma.

Come detto in precedenza la variante *LAPP* differisce dalla classica piattaforma *LAMP* per l'utilizzo dell'*RDBMS PostgreSQL* al posto di *MySQL*. Di seguito si descriverà in maniera generica *PostgreSQL* e si motiverà la scelta di adoperare tale *RDBMS* preferendolo a *MySQL*.

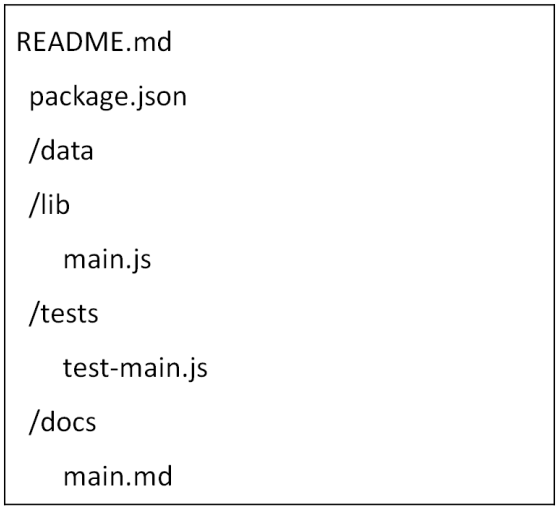


Figura 3.8: Scheletro del *plug-in*.

3.2.1 *PostgreSQL*

PostgreSQL è un sistema di gestione di *database* relazionali ad oggetti (*Object-Relational Database Management System*, *ORDBMS*) sviluppato in varie fasi fin dal 1977. Oggigiorno, questo sistema è considerato diffusamente il più avanzato *database open source* del mondo e mette a disposizione una ricchezza di funzionalità che possono essere trovate soltanto nei *database* commerciali come *DB2* o *Oracle*.

Il modello dei dati utilizzato da *PostgreSql* è quello relazionale ad oggetti, che lo rende capace di gestire procedure e regole complesse. Esempi delle sue funzionalità avanzate sono: le *query* dichiarative *SQL*, il controllo della concorrenza multi-versione, la multi-utenza, le transazioni, l'ottimizzazioni delle *query*, l'ereditarietà e gli *array*.

PostgreSQL ammette l'uso di operatori, funzioni, metodi di accesso e tipi di dati definiti dall'utente. Inoltre, supporta l'integrità referenziale che viene impiegata per garantire la validità dei dati in un *database*. Benchè *PosgreSQL* sia comunemente considerato un *RDBMS*, o semplicemente un *database*, potrebbe non essere compreso da tutti cosa si intende con la parola *database*.

Un *database* all'interno di *PostgreSQL* è un implementazione relazionale ad oggetti di ciò che è formalmente noto come schema in *SQL99*. In parole semplici, un *database* è un insieme di dati memorizzati correlati logicamente.

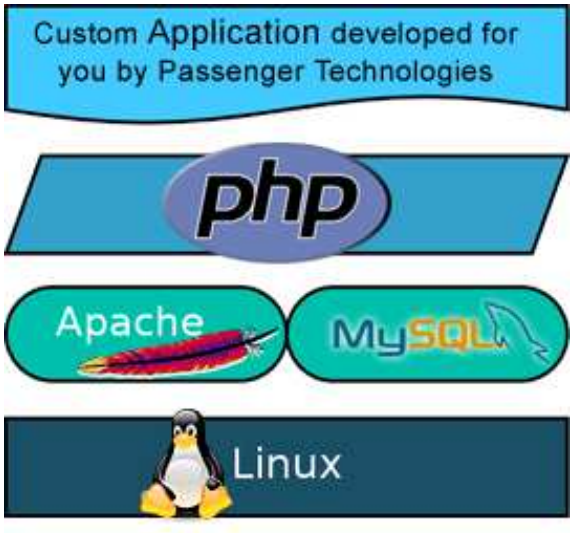


Figura 3.9: Lo *stack LAMP*.

Di solito a questi dati si può accedere attraverso un ambiente multiutente e questo è proprio il caso di *PostgreSQL*, sebbene ci siano regole ben definite e vengono imposte limitazioni a questo accesso [24].

3.2.2 Scegliere *PostgreSQL*

PostgreSQL è l'*RDBMS* più ricco dal punto di vista delle funzionalità e del supporto dello standard *SQL*. E' in grado di gestire una grande quantità di dati e presenta un ampio set di *data types* ("tipi di dati"), nel quale include anche la valuta.

Inoltre, supporta la definizione di nuovi tipi di dati e vengono utilizzate frequentemente le *Stored Procedures*. La possibilità di scrivere le *Stored Procedures* in quattro linguaggi (*pgSQL*, *Tcl*, *Perl*, *Python*) è una particolarità di *PostgreSQL*. Le *Stored Procedures* vengono utilizzate frequentemente per implementare della *business logic* all'interno dei *db-server* in maniera da centralizzare il trattamento dei dati.

Le caratteristiche appena descritte lo differenziano da *MySQL* dove tali funzionalità non sono supportate, e ne consolidano l'affidabilità [25]. Quindi, alla luce di quanto appena esposto si è scelto di utilizzare *PostgreSQL* piuttosto che *MySQL*.

Capitolo 4

Il *plug-in youtube-emotion*

L'obiettivo del presente lavoro di tesi, come è stato già descritto nel primo capitolo, è quello di realizzare uno strumento informatico che consenta di raccogliere e conservare i *feedback* degli utenti che navigano sul *web* per poter creare dei questionari, sondaggi e statistiche su un particolare argomento di interesse o studio.

Al fine di realizzare detto strumento informatico si è pensato di sviluppare un *plug-in* su *Firefox* per acquisire i suddetti *feedback* e di realizzare, inoltre, un *database* "centralizzato" per memorizzare gli stessi *feedback*. Tale strumento è uno dei tanti mezzi informatici attraverso cui è possibile raggiungere il sopraindicato obiettivo.

L'architettura che è stata utilizzata per sviluppare il *plug-in* e per realizzare il *database* è quella *client/server* che è stata già descritta nel secondo capitolo.

Le tecnologie (ambienti di sviluppo e linguaggi di programmazione) che sono state utilizzate per implementare il componente *client* (il *plug-in*) e il componente *server* (il *database*) sono quelle che sono state appena esposte al terzo capitolo.

Pertanto, nel corrente capitolo sarà esposto in maniera approfondita il *plug-in* "*youtube-emotion*". In particolare verranno descritte: la fase di progettazione e implementazione; la fase di installazione su *Firefox*; infine, la fase di effettiva applicazione sul *web*. Dopodiché verrà esposto uno strumento per testare il corretto funzionamento dell'applicazione trattata, nonché la possibilità di poter scaricare sul proprio computer il *report* delle informazioni memorizzate nel *database*.

4.1 La fase di progettazione e implementazione

In questa sezione viene descritto come è stato progettato e implementato il *plug-in* "*youtube-emotion*". Il *plug-in* è stato realizzato per raccogliere i *feedback* degli utenti, che visualizzano un video su *Youtube*, su una particolare area di interesse della psicologia: quella delle emozioni. Più precisamente, all'utente viene data la possibilità di scegliere l'emozione che scaturisce dal video che sta visualizzando su *Youtube*.

Le informazioni riguardanti tali scelte saranno poi memorizzate in un *database* che risiede su un *server*. I ricercatori e gli studiosi di psicologia utilizzano tali informazioni per formulare delle proprie statistiche e osservazioni sulle emozioni. Le emozioni in questione sono le emozioni primarie individuate dallo psicologo statunitense *Paul Ekman*: rabbia, disgusto, tristezza, gioia, paura e sorpresa [26].

Il *plug-in* è stato creato utilizzando l'*add-on SDK* esaminato nel terzo capitolo. L'utente si interfaccia con il *plug-in* attraverso un pulsante presente nell'*add-on bar* del *browser Firefox*. Al pulsante è stato fissato un pannello che si apre quando si clicca sopra al pulsante. All'interno del pannello sono state inserite le sei emozioni, un'area di testo e un bottone. Il pannello si apre solamente se l'utente sta guardando un video su *Youtube*, in caso contrario ogni azione di *click* sul pulsante non produrrà alcun risultato. All'apertura del pannello il *plug-in* recupera delle informazioni riguardanti il video visualizzato dall'utente (titolo e durata del video). L'utente sceglie l'emozione, inserisce del testo e clicca sul bottone all'interno del pannello. Tali scelte insieme alle informazioni del video vengono inviate al *server* e memorizzate all'interno del *database*.

Come già accennato nel capitolo due il *plug-in* comunica non soltanto con il *server* della piattaforma *LAPP* ove risiede il *database* ma anche con quello della piattaforma *Youtube*. Il *plug-in* infatti, come descritto in precedenza, richiede al *server* di *Youtube* delle informazioni che riguardano il particolare video che l'utente sta visualizzando. Il *server* di *Youtube* risponde alla richiesta del *plug-in* inviandogli un documento *xml* dove sono presenti tali informazioni. La duplice comunicazione *client-server* tra il *plug-in* e i *server* delle piattaforme *LAPP* (*Linux Apache PostgreSQL Php*) e *Youtube* è illustrata in Figura 4.1.

Di seguito viene riportata l'implementazione di ciascun modulo o *file* di cui è composto il *plug-in*. I *file* sono contenuti nelle cartelle "*lib*" e "*data*".

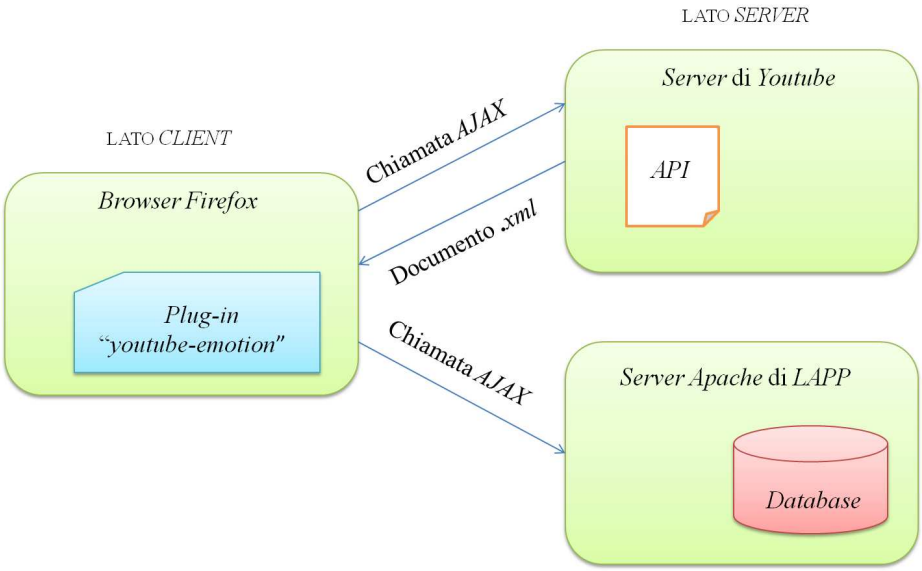


Figura 4.1: Comunicazione *client/server* tra il *plug-in* e i *server* delle piattaforme *LAPP* e *Youtube*.

4.1.1 Cartella *lib*

In questa cartella sono inseriti tutti i *file* relativi all'*add-on code*, ovvero i moduli *CommonJS* che possono utilizzare, tramite la funzione "*require()*" le funzionalità rese disponibili dai moduli dell'*SDK*. Nella cartella "*lib*" è presente un file corrispondente ad altrettanto modulo *CommonJS*: "*main.js*".

Modulo "*main.js*"

Questo è il *file* cuore del *plug-in*; gestisce e controlla tutti i componenti dell'*add-on*. In questo *file* vengono gestite le interazioni tra la parte di *add-on code* e quella di *content script*, permettendo lo scambio di dati tra le due. Esso importa molti oggetti forniti dai moduli dell'*SDK*.

In questo modulo sono state inserite tutte le funzionalità principali del *plug-in*. Tali peculiarità funzionali verranno illustrate di seguito in modo molto approfondito con lo scopo di dare un'idea della centralità di questo modulo.

Innanzitutto, è stato creato il *widget* utilizzando il modulo *widget* fornito dalle *API* dell'*add-on SDK*. Di seguito è riportato il codice con cui è stato

creato il *widget*.

```
18  var widget = require("widget").Widget({
19    id: "youtube-emotion",
20    label: "Youtube Emotion",
21    contentURL: "http://www.youtube.com/favicon.ico"
22  });
```

Al *widget* è stato associato un pannello la cui interfaccia può essere modificata attraverso codice *HTML* e *JavaScript*. Il pannello è stato creato utilizzando il modulo *panel* fornito dalle *API* dell'*add-on SDK*.

```
10  var panel = require("panel").Panel({
11    width: 544,
12    height: 244,
13    contentURL: data.url("contentPanel.html"),
14    contentScriptFile: data.url("scriptPanel.js")
15  });
```

Nella riga 13 è stato associato al pannello il *file* "*contentPanel.html*" dove è presente il contenuto del pannello. Nella riga 14, invece, è stato collegato al pannello il *content script* rappresentato dal *file* "*scriptPanel.js*".

E' stata poi ideata una particolare funzione (all'interno della quale è presente un espressione regolare) attraverso cui è possibile controllare che l'utente, nel momento in cui clicca sul *widget*, stia realmente visualizzando un video su *Youtube*; nel caso in cui l'utente abbia visualizzato il video, la funzione permette di recuperare l'*id* del video.

```
60  function getYouTubeVideoID(url) {
61
62      var regExp = /^.*((youtu.be\/)|(v\/)|(\/u\/\w\/)|(embed\/)|(watch\?))
63                  \??v?=?([^#\&\?]*).*/;
64      var match = url.match(regExp);
65      if (match&&match[7].length == 11)
66          return match[7];
67      else
68          return null;
69
70  }
```

Nelle righe 62-63 è presente l'espressione regolare che, confrontata con l'*url* della pagina del *browser* su cui sta navigando l'utente, restituisce l'*id* del video.

Tramite un'altra parte di codice è stata creata un'ulteriore funzionalità che consente di recuperare le informazioni relative al video, effettuando una chiamata *AJAX* alle *API* di *Youtube* correlate al video in questione (usando l'*id* recuperato). Al *plug-in* viene restituito un documento in formato *xml* contenente le informazioni del video.

```
73  function ajaxFunction(idVideo) {
74
75      var {XMLHttpRequest} = require("xhr");
76      var xmlHttp = new XMLHttpRequest();
77
78      xmlHttp.onreadystatechange = function() {
79          if(xmlHttp.readyState == 4 && xmlHttp.status == 200) {
80              var xmlDoc = xmlHttp.responseXML.documentElement;
81              title = xmlDoc.getElementsByTagName('title')[0].firstChild.
82                      nodeValue;
83              duration = xmlDoc.getElementsByTagName('media:content')[0].
84                         getAttribute('duration');
85          }
86      }
87
88      xmlHttp.open("get","http://gdata.youtube.com/feeds/api/videos/"+idVideo+
89                   "?v=2",true);
90      xmlHttp.send(null);
91
92  }
```

Nelle righe 81-84 si accede al documento *xml* attraverso la *dot notation* e si recuperano titolo e durata del video.

Quando il pannello viene mostrato, l'*add-on code* invia un messaggio, denominato "*show*", al *content script*.

```
26  panel.on("show", function() {
27    panel.port.emit("show");
28  });
```

Dopo che l'utente ha scelto l'emozione il *content script* invia un messaggio all'*add-on code* contenente l'emozione scelta.

```
33  panel.port.on("emotion-choice", function (emt) {
34    emotion = emt;
35  });
```

Successivamente, il *content script* invia all'*add-on code* il testo scritto dall'utente (testo vuoto se l'utente non scrive nulla). Dopodichè vengono passate le informazioni al *server* e il pannello si chiude.

```
40  panel.port.on("text-entered", function (txt) {
41    text = txt;
42    ajaxServerFunction();
43    panel.hide();
44  });
```

Infine, è stata elaborata un'altra parte di codice con il quale inviare al *server* le informazioni relative: al video che l'utente ha guardato, alla sua personale scelta afferente l'emozione e, ai dati concernenti il testo inserito.

```
97  function ajaxServerFunction() {
98
99      var {XMLHttpRequest} = require("xhr");
100     var xmlHttp = new XMLHttpRequest();
101
102     xmlHttp.onreadystatechange = function() {
103         if(xmlHttp.readyState == 4 && xmlHttp.status == 200) {
104
105         }
106     }
107
108     var url = require("simple-prefs").prefs.urlServer;
109     url += "/getDati.php?title="+title+"&duration="+duration+"&emotion="
110            +emotion+"&text="+text+"&urlTab="+urlTab;
111     xmlHttp.open("get",url,true);
112     xmlHttp.send(null);
113
114 }
```

Nella riga 108, attraverso il modulo "*simple-prefs*", si recupera l'*url* del *server* a cui devono essere trasmesse le informazioni. Nella riga successiva si inviano allo script "*getDati.php*", residente sul *server*, il titolo e la durata del video, l'emozione e il testo inserito dall'utente, e l'*url* del video.

4.1.2 Cartella *data*

Nella cartella *data* è presente tutta la parte di *content script*, codice *HTML*, codice *CSS* ed è articolata in vari *file* come si può vedere in Figura 4.2.



Figura 4.2: Struttura dei *file* della cartella *data*.

Modulo "*contentPanel.html*"

Serve per creare il contenuto del *panel* e rappresenta l'interfaccia con l'utente che deve scegliere l'emozione scaturita dal video che sta visualizzando su *Youtube*. All'interno del *panel* sono presenti: sei *radio button*, ciascuno associato ad un'emozione; un'area di testo dove l'utente inserisce del testo; e un bottone per inviare al *server* la scelta dell'emozione e il testo inserito (testo vuoto se l'utente non scrive nulla). L'area di testo e il bottone sono inizialmente disabilitati, e vengono abilitati quando si clicca su uno dei *radio button*.

```
15  <div id="radioButton">
16   <input type="radio" value="anger" name="emotion"> anger </input>
17   <input type="radio" value="disgust" name="emotion"> disgust </input>
18   <input type="radio" value="sadness" name="emotion"> sadness </input>
19   <input type="radio" value="happiness" name="emotion"> happiness </input>
20   <input type="radio" value="fear" name="emotion"> fear </input>
21   <input type="radio" value="surprise" name="emotion"> surprise </input>
22  </div>
23  <div id="textArea">
24   <textarea rows="5" cols="20" id="edit-box" disabled>Here you can enter
25   the optional text.</textarea>
26  </div>
27  <div id="button">
28   <input type="button" id="bt" value="send" disabled></input>
29  </div>
```

Modulo "*scriptPanel.js*"

Rappresenta il *content script* che comunica con il modulo "*main.js*". La comunicazione tra *content script* e *add-on code* utilizza l'oggetto *port*. In questo *file* sono state scritte: la funzione che serve per abilitare l'area di testo e il bottone, la funzione che permette di inserire un massimo di 40 caratteri nell'area di testo; e il codice con il quale il *content script* comunica con l'*add-on code*. Il *content script* invia all'*add-on code* l'emozione scelta dall'utente e il testo inserito. Di seguito sono riportate le due istruzioni di codice di quanto è stato appena detto.

```
self.port.emit("emotion-choice", emt);
self.port.emit("text-entered", txt);
```

Modulo "*stile.css*"

Viene usato per dare un senso estetico al contenuto del pannello nonchè al pannello stesso.

4.2 La fase di installazione su *Firefox*

La procedura di installazione del *plug-in* è relativamente semplice e può essere eseguita anche da utenti non esperti. Il *plug-in* è un *file* con estensione "*.xpi*". Basta aprire la pagina *web Firefox*, cliccare sul menù in alto a sinistra denominato "*Firefox*" e all'interno del menù, poi, cliccare nuovamente sulla voce "Componenti aggiuntivi" come mostrato in Figura 4.3.

In una nuova scheda appare una schermata dove sono presenti tutti i componenti aggiuntivi installati dall'utente. Successivamente bisogna cliccare sul menù in alto a destra denominato "Strumenti per tutti i componenti aggiuntivi". All'interno di quest'ultimo, selezionare la voce "Installa componente aggiuntivo da file..." come mostrato in Figura 4.4.

Dopodichè appare una finestra da dove è possibile selezionare il file con estensione "*.xpi*", denominato "*youtube-emotion*", presente in una cartella qualsiasi del computer dell'utilizzatore. Una volta che il *file* è stato trovato, selezionare il *file* e cliccare sul bottone "Apri". A questo punto, compare una nuova finestra e dopo un paio di secondi si abilita il bottone denominato "Installa adesso". Bisogna, infine, cliccare su codesto ultimo pulsante come mostrato in Figura 4.5.

Dopo aver eseguito tale procedura, il componente aggiuntivo viene installato su *Firefox* ed inserito nel menu dei componenti aggiuntivi come mostrato in Figura 4.6.



Figura 4.3: La fase di installazione su *Firefox*.

A questo punto il *plug-in* "*youtube-emotion*" è pronto per poter essere utilizzato durante la navigazione sul *web*. Per questo *plug-in*, a differenza di altri, non è richiesto il riavvio di *Firefox*. Per essere sicuri che il *plug-in* è presente su *Firefox* subito dopo l'installazione vi appare nell'*add-on bar* un *widget* o bottone in basso a destra come mostrato in Figura 4.7.

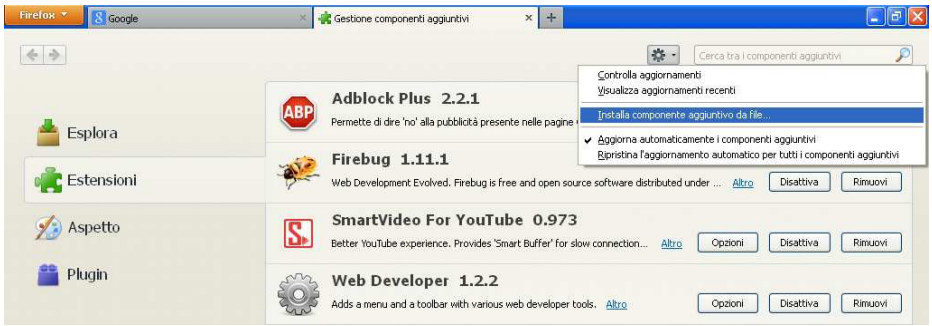


Figura 4.4: La fase di installazione su *Firefox*.

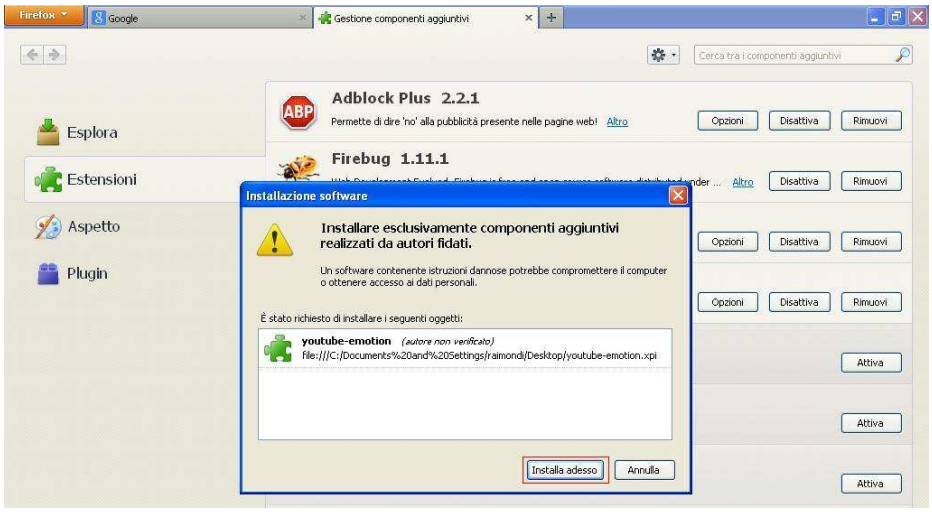


Figura 4.5: La fase di installazione su *Firefox*.

4.3 L'utilizzo del *plug-in* sul *web*

Il *plug-in* "*youtube-emotion*" funziona solo se si sta visualizzando un video sulla piattaforma *Youtube*. Infatti, se l'utente prova a cliccare sul *widget* in basso a destra non succede nulla, a meno che, non ci si trovi nella situazione descritta in precedenza. Supponiamo che egli stia visualizzando un qualsiasi video su *Youtube* come mostrato in Figura 4.8.

Qualora cliccasse sul *widget* in basso a destra (cerchiato in rosso in Figura 4.12), apparirebbe un pannello dove vengono mostrati sei *radio button*, ciascuno associato a un emozione , un'area di testo disabilitata e un bottone



Figura 4.6: La fase di installazione su *Firefox*.

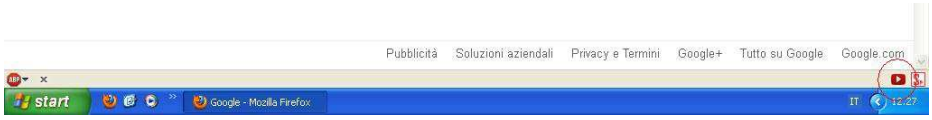


Figura 4.7: La fase di installazione su *Firefox*.

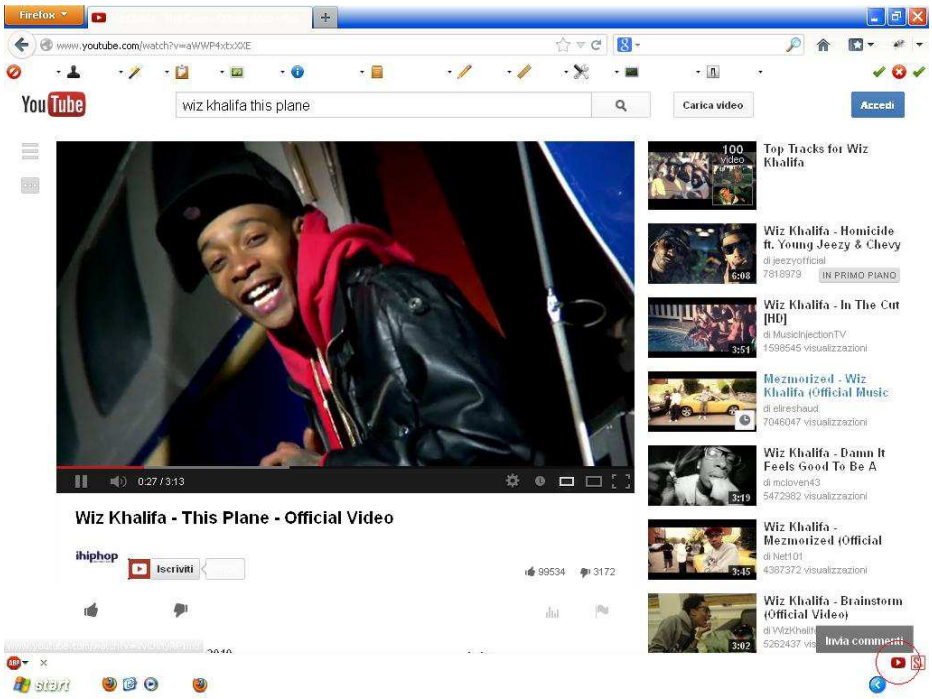


Figura 4.8: L'utilizzo del *plug-in* sul *web*.

denominato "*send*", anch'esso disabilitato, come mostrato in Figura 4.9.

Se si clicca su uno qualsiasi dei *radio button*, l'area di testo e il bottone vengono abilitati. Non è obbligatorio per l'utente scrivere nell'area di testo. L'utente sceglie l'emozione che per lui sembra più appropriata al video che sta visualizzando, scrive una sua opinione nell'area di testo oppure non vi scrive nulla, e preme il bottone "*send*", come mostrato in Figura 4.10.

Le informazioni relative al video e le scelte dell'utente verranno inviate al *server* e memorizzate nel *database*.

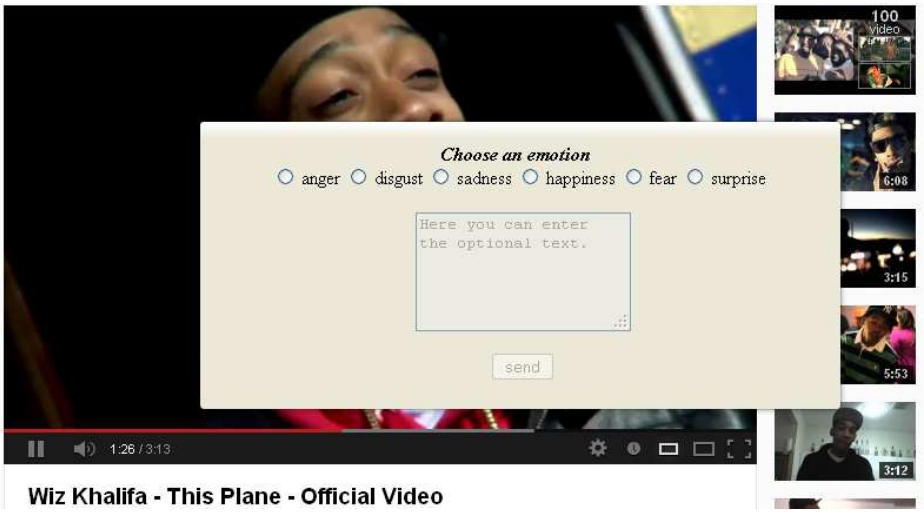


Figura 4.9: L'utilizzo del *plug-in* sul *web*.

4.4 Il testing dell'applicazione

E' possibile testare l'applicazione poc'anzi trattata facendo riferimento ad uno specifico indirizzo *web* ove sono presenti quattro comandi, implementati sotto forma di bottoni (vedi Figura 4.11).

Prima di illustrare le specifiche della funzione di ciascun bottone della pagina, verrà di seguito esposta la logica con cui è stata costruita tale pagina *web* che testa l'applicazione. Sul *server* della piattaforma *LAPP*, insieme al *database* dove sono contenute tutte le informazioni necessarie, sono presenti anche i *file* usati per il *testing* della suddetta applicazione. L'elenco di questi file è mostrato in Figura 4.12.

In "*scegli.html*" è presente un modulo con all'interno quattro pulsanti: "Visualizza contenuto del database", "Cancella l'ultima riga della tabella", "Cancella l'intera tabella" e "Download tabella in formato csv".

Sui primi tre pulsanti, a differenza del quarto, è registrato un evento *javascript*. Infatti, cliccando su uno dei primi tre pulsanti, attraverso una chiamata *AJAX*, vengono passate delle informazioni allo *script php* "*infodb.php*". Tale *script* elabora le informazioni che gli sono state passate e restituisce qualcosa alla porzione di codice che ne ha fatto richiesta.

Se invece si clicca sul quarto ed ultimo pulsante viene invocato un nuovo *script* "*scarica.php*" che permette all'utente di eseguire il *download* di un *file* in formato *csv* sul proprio computer. Il *file* "*stile.css*" viene invece usato



Figura 4.10: L'utilizzo del *plug-in* sul *web*.



Figura 4.11: La pagina *web* per il *testing* dell'applicazione.

per dare un certo aspetto estetico alla pagina oggetto del *testing*.

Dopo aver spiegato i meccanismi che regolano la comunicazione tra i *file* di *testing* appena descritti, di seguito verrà mostrato il funzionamento di ciascun bottone della pagina *web*. Il bottone "Visualizza il contenuto del database" consente di visualizzare il contenuto della tabella all'interno del *database* dove sono memorizzate le informazioni relative ai video visualizzati su *Youtube* per cui l'utente ha espresso il proprio *feedback*. Un esempio è mostrato in Figura 4.13.

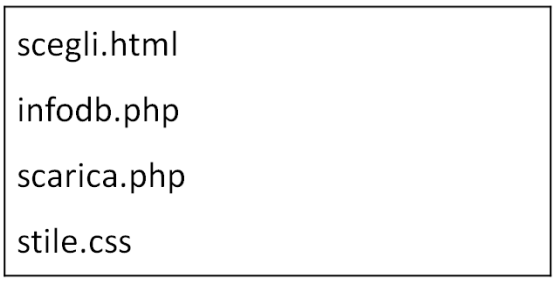
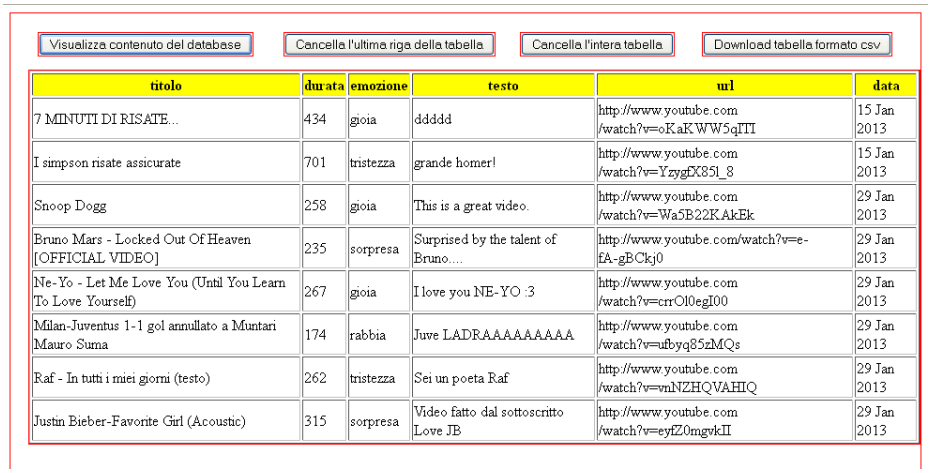


Figura 4.12: I file usati per il *testing*.



titolo	durata	emozione	testo	url	data
7 MINUTI DI RISATE...	434	gioia	ddddd	http://www.youtube.com /watch?v=oKaKWW5qITI	15 Jan 2013
I simpson risate assicurate	701	tristezza	grande homer!	http://www.youtube.com /watch?v=YzygfX85l_8	15 Jan 2013
Snoop Dogg	258	gioia	This is a great video.	http://www.youtube.com /watch?v=Wa5B22KAkEk	29 Jan 2013
Bruno Mars - Locked Out Of Heaven [OFFICIAL VIDEO]	235	sorpresa	Surprised by the talent of Bruno....	http://www.youtube.com/watch?v=e-fA-gBCkj0	29 Jan 2013
Ne-Yo - Let Me Love You (Until You Learn To Love Yourself)	267	gioia	I love you NE-YO :3	http://www.youtube.com /watch?v=crrOl0egI00	29 Jan 2013
Milan-Juventus 1-1 gol annullato a Muntari Mauro Suma	174	rabbia	Juve LADRAAAAAAAAA	http://www.youtube.com /watch?v=ufbyq85zMQs	29 Jan 2013
Raf - In tutti i miei giorni (testo)	262	tristezza	Sei un poeta Raf	http://www.youtube.com /watch?v=vnNZHQVAHIQ	29 Jan 2013
Justin Bieber-Favorite Girl (Acoustic)	315	sorpresa	Video fatto dal sottoscritto Love JB	http://www.youtube.com /watch?v=eyfZ0mgvkII	29 Jan 2013

Figura 4.13: Visualizza contenuto del *database*.

I bottoni "Cancella l'ultima riga della tabella" e "Cancella l'intera tabella" permettono, invece, la cancellazione dell'ultima riga della tabella o addirittura la rimozione dell'intera tabella. Infine, il pulsante "Download tabella formato csv" permette all'utente di poter scaricare sul proprio computer il *report* delle informazioni memorizzate nella tabella del *database*. Codeste informazioni sono raccolte in un *file* in formato *csv*. Un esempio è mostrato in Figura 4.14.

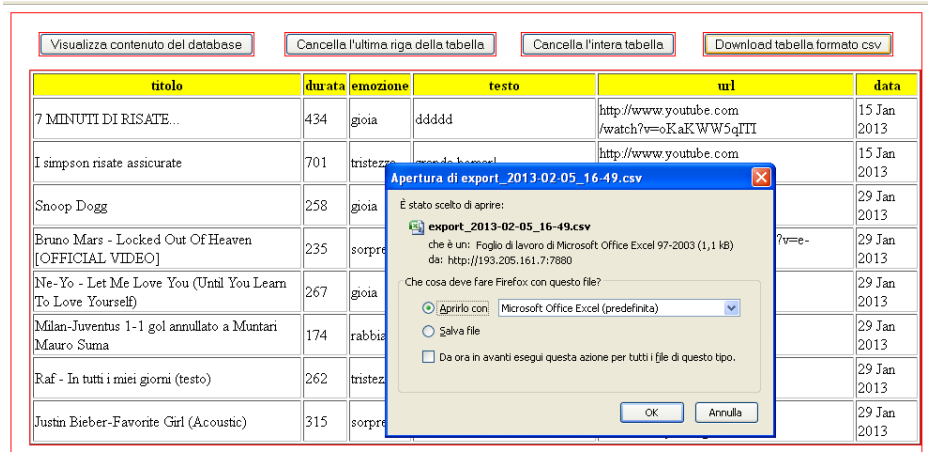


Figura 4.14: Download tabella formato *csv*.

Questo comando è stato implementato per un preciso obiettivo: dare la possibilità agli Psicologi di scaricare sul proprio computer il *report* del *database* dei video associati alle emozioni per poter studiare le emozioni e i loro effetti e creare delle statistiche, recensioni, opinioni sulle emozioni.

Capitolo 5

Conclusioni

L'iniziale obiettivo che ci eravamo proposti di raggiungere attraverso il lavoro di tesi era quello di fornire uno strumento informatico che consentisse la raccolta e la conservazione dei *feedback* che gli utenti rilasciano durante la loro navigazione web.

In virtù di ciò, abbiamo costruito un *plug-in* sul *browser Firefox* che consentisse ai fruitori dei servizi, di intrattenimento forniti dalla piattaforma di *Youtube*, di rilasciare in maniera agevole un *feedback* che descrive le emozioni suscitate dalla visione dei video presenti sul portale. Tale *plug-in* favorisce anche il rilascio facoltativo di una propria opinione o giudizio sul video visualizzato dall'utente.

Inoltre è stato realizzato il *database* atto a contenere le informazioni relative al video (titolo, durata, *url*, ...) e all'emozione (nome dell'emozione e/o testo) sopra esposte.

In conclusione, a livello operativo, è stata creata una specifica pagina *web* per verificare effettivamente che i dati siano stati archiviati nel *database* e che sia possibile scaricali dal proprio computer.

Attualmente, non esiste un *plug-in* ad *hoc* volto a catturare emozioni e opinioni dei naviganti sui contenuti del sito *web Youtube*.

Da quest'ultima singolarità è nata l'idea di costruire quello che potremmo definire "*Plug-in youtube emotion*".

I *reports* ottenuti da tale dispositivo informatico sono molto utili ai fini di varie attività di ricerca. Nel nostro particolare contesto essi saranno utilizzati per uno studio socio-psicologico relativo alle emozioni suscitate dalla visione dei contenuti interattivi della piattaforma *Youtube*.

Gli sviluppi futuri possono essere incentrati sulla creazione di un modulo di autenticazione che consenta di risolvere il problema pratico riguardante

la possibilità che un utente fornisca più di una volta la sua emozione sul medesimo video. Infatti sarebbe estremamente fuorviante, ai fini delle successive elaborazioni dei dati raccolti col *plug-in* in argomento, ottenere tali doppie scelte che non costituiscono altro che valori anomali [27].

Dunque, per non appesantire le tempistiche e i costi monetari delle successive inferenze statistiche sui dati raccolti si potrebbe introdurre un modello di autenticazione costituito da *username* e *password* che limiti le opinioni degli utenti a una per ogni singolo contenuto visualizzato.

In secondo luogo, pare interessante far notare che lo scheletro della programmazione utilizzata per lo specifico *plug-in* può ben essere adoperato per sviluppare altri meccanismi di raccolta dati da agganciare a distinti *browser* (ad esempio: *Google Chrome*, *Safari*, *Opera*, et eccetera), e per recuperare *feedback* su altre piattaforme e per altri scopi.

Un ulteriore sviluppo che può essere apportato al dispositivo di raccolta dei dati può essere l'inserimento di altre tipologie di informazioni rilevanti e afferenti il contenuto visualizzato dall'utente.

Ad esempio, un dato utile per le elaborazioni socio-psicologiche, è quello riguardante il momento preciso del video o contenuto (minuto e secondo) in cui l'utente fornisce la sua scelta emozionale.

Bibliografia

[1] Marco Artrusi. *Internet Marketing experience*. Len Strategy. 2008. p.198. 1.1

[2] Jim Blythe. *Fondamenti di marketing*. Edizione italiana a cura di Elena Cedrola. Pearson Education. 2006. p. 263. 1.2

[3] Diennnea MagNews. *Le best practice più efficaci dell'e-mail marketing*. Faenza. p. 31. 1.2

[4] Giuseppe Cassano e Iacopo Pietro Cimino. *Diritto dell'Internet e delle nuove tecnologie telematiche*. Wolters Kluwer Italia. 2009. p. 226. 1.2

[5] Dario Meggiorini. *Introduzione alla programmazione client-server*. Pearson. 2009. 2.1

[6] *Evoluzione dei modelli di elaborazione*. Disponibile da: `http://www.sezioneb.com` 2.1

[7] Ramez A. Elmasri and Shamkant B. Navathe. *Sistemi di basi di dati. Fondamenti*. Edizione italiana a cura di Silvana Castano. Pearson. Addison Wesley. 2007. 2.1

[8] Giuseppe Naccarato. *Java Database e programmazione client/server*. Apogeo. 2001. 2.1

[9] Chuck Urwiler, Gary DeWitt, Mike Levy and Lesli Koorhan. *Client/Server Applications with Visual FoxPro and SQL Server*. Hentzenwerke Publishing. 2000. 2.1

[10] Wendell Odom and Tom Knott. *Fondamenti di networking*. Cisco System. 2006. 2.1.2

[11] *Server proxy (server mandatari) e reverse proxy*. Disponibile da: `http://it.kioskea.net` 2.2

[12] Douglas E. Comer. *Internetworking con TCP/IP. Principi, protocolli e architetture*. Pearson. Prentice Hall. 2006. 2.2.2

[13] George Reese. *Cloud computing. Architettura, infrastrutture, applicazioni*. O'Reilly. Hoops. 2010.

[14] Tiziano Daniotti. *HTML 4*. Apogeo. 2001. 3

[15] Gianluca Troiani. *CSS. Guida completa*. Apogeo. 3

[16] David Flanagan. *Javascript La guida, seconda edizone*. Apogeo. 2002. p. 1. 3

[17] Nicholas C. Zakas, Jeremy McPeak and Joe Awcett. *Professional AJAX*. Wrox. 3

[18] *Add-on SDK Version 1.12*. Mozilla. 2012. Disponibile da: `https://addons.mozilla.org/en-US/developers/docs/sdk/latest/`. 3.1.1

[19] *Content Scripts*. Mozilla. 2012. Disponibile da: `https://addons.mozilla.org/en-US/developers/docs/sdk/1.13/dev-guide/guides/content-scripts/index.html`. 3.1.1

[20] *CommonJS, Modules, Packages, and the SDK*. Mozilla. 2012. Disponibile da: `https://addons.mozilla.org/en-US/developers/docs/sdk/1.13/dev-guide/guides/modules.html` 3.1.1

[21] *Package Specification*. Mozilla. 2012. Disponibile da: `https://addons.mozilla.org/en-US/developers/docs/sdk/1.13/dev-guide/package-spec.html` 3.1.1

[22] *Two Types of Scripts*. Mozilla. 2012. Disponibile da: `https://addons.mozilla.org/en-US/developers/docs/sdk/latest/dev-guide/guides/two-types-of-scripts.html` 3.1.1

[23] Charles Bell. *Expert MySQL*. Apress. 3.2

[24] John C. Worsley and Joshua D. Drake. *Practical PostgreSQL*. Hoops Libri. 2002. 3.2.1

[25] Kevin Kline and Daniel Kline. *SQL. Guida di riferimento.* Apogeo. O'Reilly. 2001. 3.2.2

[26] Paul Ekman and Wallace V. Friesen. *Unmasking the face. A guide to recognizing emotions from facial expressions.* Malor Books. 2003. 4.1

[27] Sergio Zanio e Andrea Cerioli. *Analisi dei dati e data mining.* Giuffrè editore. 2011. p. 79 e ss. 5

Appendice A

Appendice

A.1 Listati

In questa sezione saranno esposti i listati di codice dei moduli o *file* di cui è composto il *plug-in* "*youtube-emotion*". Tali *file* sono stati già elencati in 4.1.1 di cui abbiamo fornito una breve spiegazione delle loro funzionalità all'interno del *plug-in*.

A.1.1 Il *file* "*main.js*"

```
1  /* This Source Code Form is subject to the terms of the Mozilla Public
2   * License, v. 2.0. If a copy of the MPL was not distributed with this
3   * file, You can obtain one at http://mozilla.org/MPL/2.0/. */
4
5  var tabs = require("tabs");
6  var data = require("self").data;
7
8  // Create a panel whose content is defined in "contentPanel.html".
9  // Attach a content script called "scriptPanel.js".
10 var panel = require("panel").Panel({
11   width: 544,
12   height: 244,
13   contentURL: data.url("contentPanel.html"),
14   contentScriptFile: data.url("scriptPanel.js")
15 });
16
17 // Create a widget.
18 var widget = require("widget").Widget({
```

```
19      id: "youtube-emotion",
20      label: "Youtube Emotion",
21      contentURL: "http://www.youtube.com/favicon.ico"
22    });
23
24    // Send the content script a message called "show" when
25    // the panel is shown.
26    panel.on("show", function() {
27      panel.port.emit("show");
28    });
29
30    // Listen for messages called "emotion-choice" coming from
31    // the content script. The message payload is the emotion the user
32    // choice.
33    panel.port.on("emotion-choice", function (emt) {
34      emotion = emt;
35    });
36
37    // Listen for messages called "text-entered" coming from
38    // the content script. The message payload is the text the user
39    // entered.
40    panel.port.on("text-entered", function (txt) {
41      text = txt;
42      ajaxServerFunction();
43      panel.hide();
44    });
45
46    // When you click on the widget retrieves the url and checks if it is
47    // on youtube and you are viewing a video.
48    // If so, the panel is shown.
49    widget.on("click",function() {
50      urlTab = tabs.activeTab.url;
51      var idVideo = getYouTubeVideoID(urlTab);
52      if (idVideo != null) {
53         ajaxFunction(idVideo);
54         panel.show();
55      }
56    });
57
58    // Use a regular expression to check if you are on youtube
```

```
59  // and if you are viewing a video.
60  function getYouTubeVideoID(url) {
61
62      var regExp = /^.*((youtu.be\/)|(v\/)|(\/u\/\w\/)|(embed\/)|(watch\?))
63                  \??v?=?([^#\&\?]*).*/;
64      var match = url.match(regExp);
65      if (match&&match[7].length == 11)
66          return match[7];
67      else
68          return null;
69
70  }
71
72  // Retrieve the title and duration of the video using an ajax call.
73  function ajaxFunction(idVideo) {
74
75      var {XMLHttpRequest} = require("xhr");
76      var xmlHttp = new XMLHttpRequest();
77
78      xmlHttp.onreadystatechange = function() {
79          if(xmlHttp.readyState == 4 && xmlHttp.status == 200) {
80              var xmlDoc = xmlHttp.responseXML.documentElement;
81              title = xmlDoc.getElementsByTagName('title')[0].firstChild.
82                      nodeValue;
83              duration = xmlDoc.getElementsByTagName('media:content')[0].
84                      getAttribute('duration');
85          }
86      }
87
88      xmlHttp.open("get","http://gdata.youtube.com/feeds/api/videos/"+idVideo+
89                  "?v=2",true);
90      xmlHttp.send(null);
91
92  }
93
94  // Send the script on the server called "getDati.php"
95  // the title of the video, the video duration,
96  // emotion and any text entered by the user.
97  function ajaxServerFunction() {
98
```

```
99        var {XMLHttpRequest} = require("xhr");
100       var xmlHttp = new XMLHttpRequest();
101
102       xmlHttp.onreadystatechange = function() {
103           if(xmlHttp.readyState == 4 && xmlHttp.status == 200) {
104
105           }
106       }
107
108       var url = require("simple-prefs").prefs.urlServer;
109       url += "/getDati.php?title="+title+"&duration="+duration+"&emotion="
110              +emotion+"&text="+text+"&urlTab="+urlTab;
111       xmlHttp.open("get",url,true);
112       xmlHttp.send(null);
113
114   }
```

A.1.2 Il *file* "*contentPanel.html*"

```
1   <!-- This Source Code Form is subject to the terms of the Mozilla Public
2        License, v. 2.0. If a copy of the MPL was not distributed with this
3        file, You can obtain one at http://mozilla.org/MPL/2.0/. -->
4
5   <html>
6
7   <head>
8     <link rel="stylesheet" type="text/css" href="stile.css"> </link>
9   </head>
10
11  <body>
12   <div id="container">
13    <form>
14    <span id="text"> Choose an emotion </span>
15     <div id="radioButton">
16      <input type="radio" value="anger" name="emotion"> anger </input>
17      <input type="radio" value="disgust" name="emotion"> disgust </input>
18      <input type="radio" value="sadness" name="emotion"> sadness </input>
19      <input type="radio" value="happiness" name="emotion"> happiness </input>
20      <input type="radio" value="fear" name="emotion"> fear </input>
21      <input type="radio" value="surprise" name="emotion"> surprise </input>
```

```
22      </div>
23      <div id="textArea">
24       <textarea rows="5" cols="20" id="edit-box" disabled>Here you can enter
25       the optional text.</textarea>
26      </div>
27      <div id="button">
28       <input type="button" id="bt" value="send" disabled></input>
29      </div>
30     </form>
31    </div>
32   </body>
33
34   </html>
```

A.1.3 Il *file* "*scriptPanel.js*"

```
1   /* This Source Code Form is subject to the terms of the Mozilla Public
2    * License, v. 2.0. If a copy of the MPL was not distributed with this
3    * file, You can obtain one at http://mozilla.org/MPL/2.0/. */
4
5   self.port.on("show", function () {
6
7     var button = document.getElementById('bt');
8     var textArea = document.getElementById('edit-box');
9     var radio = document.getElementsByName('emotion');
10
11    // When the user click the radio-button, enables the edit-box
12    // and the bt.
13    for (var i=0;i<radio.length;i++) {
14              radio[i].onclick = function() {
15                  textArea.removeAttribute("disabled");
16                  textArea.value = textArea.value.substring(0,0);
17                  button.removeAttribute("disabled");
18              }
19    };
20
21    // The user can enter a maximum of 40 characters.
22    textArea.onkeyup = function() {
23       var text = textArea.value;
24       if (text.length > 40) {
```

```
25          alert("The text you entered more than 40 characters");
26          textArea.value = text.substring(0,text.length-1);
27       }
28    };
29
30    // When the user click the button, send a message to main.js.
31    // The message payload is the contents of emotion and edit box.
32    button.onclick = function() {
33        for (var i=0;i<radio.length;i++) {
34             if (radio[i].checked)
35                 var emt = radio[i].value;
36        };
37        var txt = textArea.value;
38        self.port.emit("emotion-choice", emt);
39        self.port.emit("text-entered", txt);
40        textArea.value = '';
41        for (var i=0;i<radio.length;i++) {
42             radio[i].checked = false;
43        };
44    };
45
46 });
```

A.1.4 Il *file* "*stile.css*"

```
1  /* This Source Code Form is subject to the terms of the Mozilla Public
2   * License, v. 2.0. If a copy of the MPL was not distributed with this
3   * file, You can obtain one at http://mozilla.org/MPL/2.0/. */
4
5  #container {
6     position : relative;
7     width : 530;
8     height : 220;
9  }
10
11 #radioButton {
12    position : absolute;
13    top : 21px;
14    left : 10%;
15 }
```

```
16
17  #textArea {
18     position : absolute;
19     top : 60px;
20     left : 33%;
21  }
22
23  #button {
24     position : absolute;
25     top : 180px;
26     left : 45%;
27  }
28
29  #text {
30     position : absolute;
31     top : 1px;
32     left : 37%;
33     font-weight : bold;
34     font-style : italic;
35  }
```