ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA
SCUOLA DI INGEGNERIA E ARCHITETTURA
Corso di Laurea in Ingegneria Elettronica, Informatica e Telecomunicazioni

RUOLO DELLE ARCHITETTURE AD EVENTI NELLO SVILUPPO DELLE APPLICAZIONI MODERNE

Elaborata nel corso di: Fondamenti di Informatica LB

Tesi di Laurea di:
ROBERTO REDA

Relatore:
Prof. ALESSANDRO RICCI

ANNO ACCADEMICO 2012–2013
SESSIONE II

PAROLE CHIAVE

asynchronous programming
event-driven programming
javascript

Dedicato a tutti gli amanti della
programmazione asincrona.

Indice

Introduzione **ix**

1 Dalla Programmazione Sequenziale agli Eventi **1**
1.1 Modelli di Esecuzione 1
1.1.1 Modello Sequenziale 2
1.1.2 Modello Parallelo 2
1.1.3 Modello Asincrono 3
1.2 I Tre Modelli a Confronto 5
1.3 Esempio Esplicativo 7
1.4 Criteri di Scelta 9

2 Architetture ad Eventi **11**
2.1 Definizione di evento 11
2.2 Gestione degli Eventi 11
2.3 Event Loop 12
2.4 Sistemi Event-Driven 12
2.5 Sistemi Event-Driven e Programmazione Asincrona 13
2.6 Principali Pattern 13
2.6.1 Pattern Observer 13
2.6.2 Pattern Event Listener 14
2.6.3 Pattern Publish-Subscribe 15

3 Applicazioni Moderne **17**
3.1 Web Application 17
3.1.1 Struttura delle Web Application 18
3.1.2 Vantaggi e Svantaggi delle Web Application 19
3.2 Mobile Application 19

3.2.1 Gestione degli Eventi in iOS 20
3.2.2 Vantaggi e Svantaggi delle Mobile Application 22

4 Implementazione in JavaScript **23**
4.1 JavaScript 23
4.1.1 JavaScript e Web Application 24
4.1.2 JavaScript e Mobile Application 24
4.2 Eventi in JavaScript 24
4.2.1 Scheduling degli Eventi 26
4.3 Pattern Publisher Subscriber 27
4.3.1 PubSub in JavaScript 28
4.4 Pattern Promise 29
4.4.1 Promise in Q.js 30
4.4.2 Promise in jQuery 31
4.4.3 Combinazione di Promise 32

5 Programmazione Asincrona in Sistemi Multiprocessore **35**
5.1 Eventi e Concorrenza 35
5.2 Web Workers 36
5.2.1 Esempi di Utilizzo 37

6 Conclusioni **41**

Introduzione

Attualmente il panorama informatico è dominato dai dispositivi mobile: smartphone e tablet pc dominano incontrastati la scena del mercato elettronico. Questo comporta un radicale ripensamento e cambiamento del software le *web app* e le *mobile application* richiedono infatti una sempre maggiore reattività dell'interfaccia utente, la persistente connessione a Internet e l'interazione con una moltitudine di dispositivi esterni.

Il progettista di software deve oggi far fronte a tutta una serie di problematiche, l'aumentata complessità dei sistemi e i sempre più ristretti tempi di sviluppo e consegna richiedono compromessi tra la semplicità delle tecniche di progettazione e l'efficienza del prodotto ottenuto.

Le architetture ad eventi *in primis*, unitamente al paradigma di programmazione asincrona, si pongono come soluzione ottimale a queste esigenze.

L'obbiettivo principale di questa tesi è quello di offrire una panoramica generale sullo stato dell'arte delle *architetture ad eventi* focalizzandosi sul ruolo che esse assumono nel contesto delle applicazioni moderne, intendendo principalmente con questo termine le *web application* e le *mobile application*. Partendo dal concetto di programmazione sincrona e parallela si giunge a descrivere un terzo modello, il *modello asincrono*, di fondamentale importanza per i sistemi *event-driven*.

Utilizzando come principale linguaggio di riferimento JavaScript si affrontano le problematiche legate alla stesura del codice per la gestione degli eventi, l'asincronicità intrenseca degli eventi e l'utilizzo di funzioni di *callback* portano a produrre codice di difficile lettura e manutenzione.

Si analizzano quindi in dettaglio i pattern fondamentali e le tecniche attualmente utilizzate per l'ottimizzazione della gestione del codice e delle problematiche esposte fornendo numerosi esempi esplicativi.

Il Capitolo 1 introduce il concetto fondamentale di programmazione asincrona dopo aver analizzato in dettaglio i tre modelli principali di compu-

tazione: il modello sincrono, il modello asincrono e il modello parallelo. Si illustrano in dettaglio le caratteristiche di ognuno mettendone in evidenza i punti di forza comparandoli.

Nel Capitolo 2 viene introdotto il concetto di evento, si introducono le architetture ad eventi e si trattano i sistemi event-driven. Si mostra come gli eventi e la programmazione asincrona diventano entità complementarie.

Nel Capitolo 3 si trattano le applicazioni moderne, riferendosi con questo termine alle web application e alle mobile application che dominano il panorama informatico attuale. Si discute delle loro caratteristiche, di come sono strutturate e di come le architetture ad eventi vengono impiegate nella realizzazione.

Nel Capitolo 4 si analizzano in dettaglio i pattern principali per la gestione degli eventi la programmazione asincrona utilizzando come linguaggio di riferimento JavaScript. Si forniscono numerosi esempi e casi di utilizzo per ogni tecnica proposta.

Il Capitolo 5 affronta il problema di come si possa coniugare l'efficienza del calcolo parallelo con i benifici apportati da un approccio di esecuzione asincrona single-thread.

Capitolo 1

Dalla Programmazione Sequenziale agli Eventi

In questo primo capitolo si affronta uno dei concetti fondamentali di questa tesi: la *programmazione asincrona*. Dopo aver ripreso brevemente il modello di computazione sequenziale e parallelo si introduce il concetto di esecuzione asincrona del codice mettendo in evidenza, come per certe tipologie di applicazioni, questo apporti benefici rispetto al tradizionale modello multithreaded. Si introduce poi il concetto di evento e si chiarisce come questo vada ad inserirsi nel contesto appena introdotto dando così origine alla programmazione sincrona ad eventi. Si fornisce una definizione precisa sul significato del termine *evento* in un contesto di tipo informatico proponendo diversi esempi, si passa poi a definire i sistemi *event-driven* sottolineando le loro caratteristiche principali.

1.1 Modelli di Esecuzione

Possiamo suddividere in tre parti le fasi di esecuzione di un tipico programma. La prima avviene mediante l'inserimento dei dati in ingresso (*input*), la seconda consiste nell'elaborazione dei dati inseriti e la terza la presentazione del risultato, ovvero i dati in uscita (*output*).

La seconda fase avviene mediante il completamento di un certo numero di attività logiche (*task*) che l'elaboratore deve eseguire consumando l'input per ottenere l'output.

Si ricorda che per modello di esecuzione si intende l'ordine temporale con cui i vari task che costituiscono il programma vengono eseguite dal microprocessore (o dai microprocessori in caso di sistemi multicore).

1.1.1 Modello Sequenziale

Nel modello di programmazione sequenziale (*single-threaded*) i vari task previsti dal programma vengono eseguiti al più uno per volta, e una successiva attività non può essere eseguita se la precedente non è stata terminata. Poiché quindi l'ordine di esecuzione è predefinito, l'implementazione di un task successivo ad altri, può assumere come scontato la buona riuscita dei precedenti, terminati correttamente e senza errori con tutta l'elaborazione parziale accumulata disponibile da quel momento in avanti.

La programmazione sequenziale rappresenta il più semplice stile di programmazione. Il programmatore si limita ad indicare le istruzioni che verrano poi eseguite nello stesso ordine cui sono state scritte senza doversi preoccupare d'altro.

Il diagramma in Figura 1.1 rappresenta in maniera grafica un'ipotetica elaborazione mediante l'esecuzione sequenziale di tre diversi task necessari per portare a termine la computazione completa. La freccia del tempo mette in evidenza come questi vengano eseguiti dal calcolatore in precisa successione l'uno con l'altro e mai più di un'attivita diversa contemporaneamente, così per esempio il *taskB* viene eseguito al termine del *taskA* e completato precedentemente l'inizio di esecuzione del *taskC*.

1.1.2 Modello Parallelo

Nel modello di esecuzione parallelo o *multithreaded* ogni task viene eseguito in un separato flusso di controllo e in ordine temporale parallelo rispetto l'esecuzione di altri eventuali task.

Nel modello di esecuzione parallelo o multithreaded l'elaborazione viene portata a termine mediante l'esecuzione simultanea dei vari task. Un singolo task viene eseguito in un separato flusso di controllo (thread) in parallelo rispetto l'esecuzione di altri eventuali task. I threads vengono gestiti dal sistema operativo e in caso di elaboratori fisici dotati di più processori la computazione avviene realmente in maniera simultanea (in senso fisico e non logico).

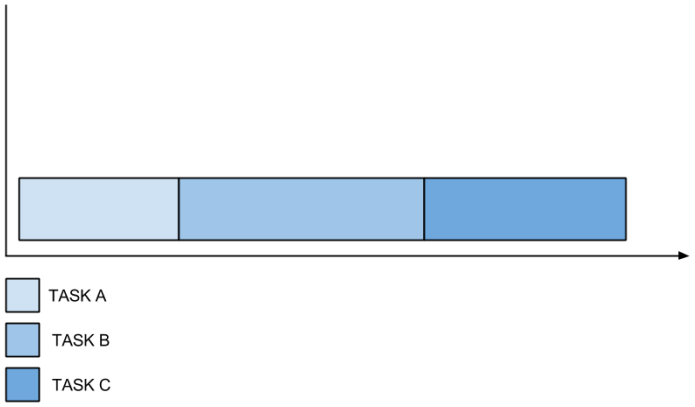

Figura 1.1: Esecuzione sequenziale.

Il diagramma in Figura 1.2 rappresenta lo stesso caso di elaborazione descritto nel paragrafo precedente, con la differenza che i task vengono eseguiti simultaneamente lungo la linea del tempo. Appare evidente il vantaggio che si ottiene in termini di tempo rispetto ad un'esecuzione sequenziale (quanto appena detto, è valido su sistemi multiprocessore).

Per quanto semplice possa sembrare il diagramma soprastante, nella realtà pratica la programmazione multithread diventa velocemente piuttosto complessa per il programmatore a causa della necessità di dover coordinare e mettere in comunicazione i thread l'uno con l'altro.

1.1.3 Modello Asincrono

Ai due modelli sopra descritti si aggiunge il terzo, il *modello asincrono*, quest'ultimo riveste un'importanza fondamentale congiuntamente al concetto di eventi.

Il modello di esecuzione delle attività asincrono può essere implementato mediante singolo flusso di controllo principale, sia in sistemi monoprocessore, sia sistemi multiprocessore con particolari accorgimenti.

Nel modello di esecuzione concorrente asincrona l'esecuzione dei vari task si interseca lungo la linea temporale, il tutto avviene sotto l'azione di un singolo flusso di controllo (single-threaded).

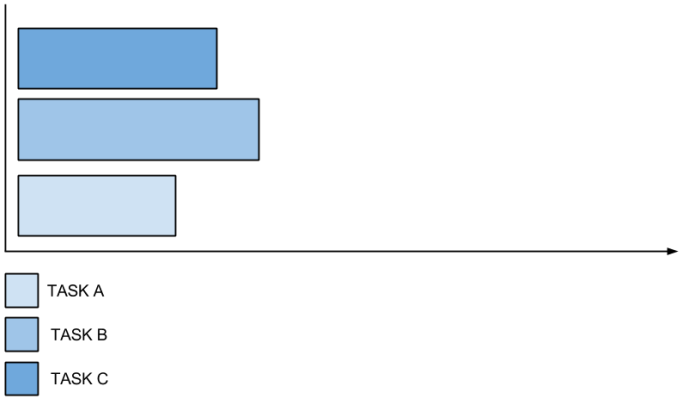

Figura 1.2: Esecuzione parallela.

L'esecuzione di un'attività una volta iniziata può essere sospesa e poi ripresa nel tempo alternandosi con l'esecuzione di altri aventuali task presenti.

Il diagramma in Figura 1.3 esplica in maniera chiara questo concetto. L'esempio di riferimento è il medesimo dei precedenti, l'elaborazione avviene mediante l'espletamento dei tre task a singolo flusso di controllo utilizzando un sistema monoprocessore, qui si evidenzia però come questi non siano eseguiti in successione cronologica fissa, ma incrociata.

Il programmatore qualora decida di utilizzare il modello asincrono non deve pensare alla stesura del codice come il modello concorrente parallelo multi-threading.

Una differenza sostanziale tra il modello concorrente parallelo multi-threaded e il modello concorrente asincrono single-threaded risiede nel fatto che nel primo caso il sistema operativo decide sulla linea temporale se sospendere l'attività di un thread e avviarne un'altra. Ciò rimane fuori dal controllo del programmatore che al contrario del modello asincrono l'esecuzione o terminazione di un task continua finché esplicitamente richiesto.

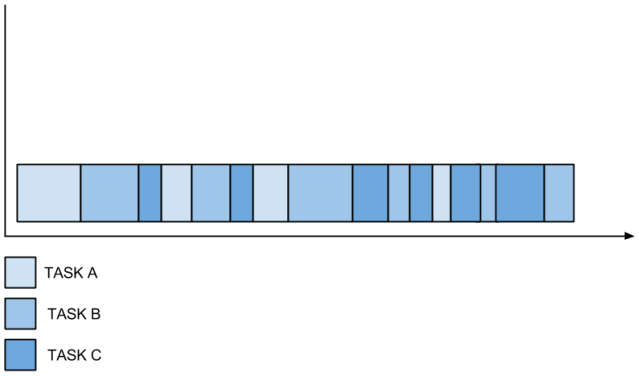

Figura 1.3: Esecuzione asincrona.

1.2 I Tre Modelli a Confronto

Consideriamo un sistema monoprocessore e single-threaded, confrontando i diagrammi in Figura 1.1 e 1.3 appare evidente che in assenza di parallelismo non vi sia alcun vantaggio in termini temporali sul modello asincrono rispetto quello sequenziale, inoltre il fatto che lo svolgimento dei task possa incrociarsi nel tempo può sembrare una complicazione inutile.

Sotto certe condizioni però il modello asincrono presenta indubbi vantaggi su quello sincrono, in tutte le situazioni in cui i task siano forzati ad attendere o debbano essere per qualche motivo bloccati. Tali problematiche si manifestano frequentemente nei casi di lettura o scrittura dai dispositivi di memorizzazione di massa o ricezione di dati da rete o sensori.

Generalmente il tempo impiegato per le operazioni di input-output o trasferimento dati è di ordine di grandezza decisamente superiore a quello di funzionamento della CPU, cosi un programma asincrono che debba effettuare un notevole numero di letture o scritture su disco o trasferire ingenti quantità di dati con un altro dispositivo impiegherà gran parte del tempo di esecuzione totale rimanendo bloccato in attesa. Per questa ragione i programmi sincroni sono anche detti programmi bloccanti (blocking-program).

Il diagramma in Figura 1.4 rappresenta un programma che debba effettuare un certo numero di operazioni bloccanti durante la propria esecuzione,

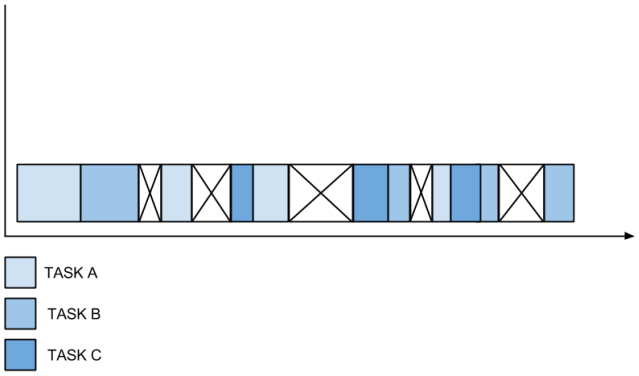

Figura 1.4: Operazioni bloccanti.

viene messo in evidenza come la maggior parte del tempo totale sia impiegata ponendo la CPU in attesa. Si noti come non sia un caso il fatto che il diagramma in Figura 1.4 assomigli al diagramma 1.3 di un programma asincrono, ovvero si ripresenta l'esecuzione intermittente dei task.

Da queste considerazioni emerge l'idea di fondo che sta alla base del modello asincrono. Quando è necessario eseguire task che richiedono molteplici operazioni bloccanti, il tempo speso in attesa può essere impiegato per proseguire altre attività riattivando poi il task precedente una volta che le operazioni di I/O siano concluse, così che l'esecuzione si sospenda solo quando non vi sono ulteriori compiti da svolgere. Per via di queste motivazioni i programmi asincroni sono anche detti programmi non bloccanti (non-blocking programs).

Un programma che debba eseguire un notevole numero di potenziali task bloccanti può beneficiare considerevolmente in termini di tempo adottando un approccio asincrono riducendo così i tempi di attesa da parte della CPU che può così portare a termine lo svolgimento di ulteriori attività.

Il modello di computazione parallela multi-threading potendo svolgere più compiti diversi contemporaneamente si pone come ottima alternativa nel caso in cui si debbano gestire più task bloccanti. Inoltre proprio per via della sua natura intrenseca simultanea risulta decisamente più veloce in molte occasioni rispetto al modello sincrono.

In realtà in più situazioni il modello asincrono è preferibile anche rispetto al modello parallelo multithreading, infatti quest'ultimo richiede un'intensa collaborazione da parte del sistema operativo che deve occupparsi della gestione dei thread gravando, sia in termini di memoria, sia in termini di calcolo(*overhead*) tali per cui la gestione asincrona del medesimo problema risulta conveniente.

1.3 Esempio Esplicativo

A titolo esemplificativo di quanto appena detto si riporta in l'implementazione in C# sia sincrona sia quella asincrona di un programma che debba scaricare dati dalla rete stampando infine il risultato.

Listing 1.1: Versione Sincrona

```
1  public static void Download()
2  {
3    Console.WriteLine("---- Download ----");
4
5    WebRequest client =
         HttpWebRequest.Create("http://www.google.com");
6    var response = client.GetResponse();
7
8    using (var reader = new
         StreamReader(response.GetResponseStream()))
9    {
10     // esecuzione sincrona della richiesta
11     var result = reader.ReadToEnd();
12
13     Console.WriteLine(result);
14   }
15 }
```

Il codice di esempio sopra riportato utilizza la classe WebRequest e, successivamente, StreamReader, per recuperare i dati dall'indirizzo specificato e, fintanto che il trasferimento non risulta ultimato, il thread principale resta bloccato e in attesa del completamento. Sfruttando *async* e *await* possiamo realizzarne la versione sincrona in maniera estremamente semplice.

Listing 1.2: Versione Asincrona

```
public static async Task DownloadAsync()
{
  Console.WriteLine("---- DownloadAsync ----");

  WebRequest client =
      HttpWebRequest.Create("http://www.google.com");
  var response = client.GetResponse();

  using (var reader = new
      StreamReader(response.GetResponseStream()))
  {
    // esecuzione asincrona della richiesta
    var result = await reader.ReadToEndAsync();

    Console.WriteLine(result);
  }
}
```

Sfruttando le parole chiave *async* e *await* si realizza la versione asincrona in maniera estremamente semplice evitando di bloccare il thread principale.

L'implementazione del metodo richiede infatti di specificare la keyword async nella sua dichiarazione, mentre il tipo di dato restituito è variato da void a Task. Si tratta di una direttiva che indica al compilatore l'intenzione di gestire all'interno di questo metodo delle chiamate asincrone. Il passo successivo è quello di utilizzare il metodo ReadToEndAsync, che rappresenta la variante asincrona di ReadToEnd: esso, infatti, restituisce un oggetto di tipo Task¡string¿ ed esegue l'operazione di download in un altro thread. Con la parola chiave await si preleva poi il risultato dell'esecuzione asincrona e assegnandolo alla variabile result di tipo string.

Diversamente nel secondo caso la funzione *asynchDownload()* è di tipo asincrono quindi non bloccante, di conseguenza il thread principale in attesa che il download dei dati sia terminato può eseguire la seconda funzione *otherFunction()*. Al momento di stampare i dati, si noti l'uso della keyword *await*, se il download non è ancora terminato si pone in attesa che la funzione *asynchDownload()* restituisca il risultato e poi stampa. Se la funzione *otherFunction()* per essere eseguita impiega un tempo inferiore del trasferimento dei dati, il tempo totale di esecuzione diversamente dal caso

precedente non sarà uguale alla somma dei singoli tempi di completamento ma sarà uguale al solo tempo di attesa per il download dei dati.

1.4 Criteri di Scelta

La scelta del tipo di approccio da addottare è strettamente collegata al tipo di problema che si intende risolvere. Programmi che prevedono l'esecuzione di un ridotto numero di task bloccanti possono essere risolti adottando uno stile di programmazione sequenziale di più facile implementazione e verifica (*testing*) consentendo un più celere deployment.

Al contrario per programmi che richiedono l'esecuzione di molteplici task bloccanti convenientemente ci si orienterà ad un approccio asincrono come giusto compresso tra facilità di progettazione, economia e sfruttamento efficiente delle risorse di calcolo e memoria evitando così l'overhead introdotto per la gestione di più threads.

La programmazione multi-threading resterà la soluzione migliore nei casi in cui la mole di lavoro (in termini di CPU) è tale da giustificare le complicazioni di progettazione dovute al coordinamento e alla comunicazione dei thread e l'overhead di gestione da parte del sistema operativo.

Per trarne a pieno benifici dallo stile concorrente è altresì importante che il sistema che ospita un programma di quest'ultimo tipo sia multiprocessore, ovvero permetta un'elaborazione simultanea in senso fisico diversamente dall'approccio asincrono che può trarre vantaggio anche su sistemi monoprocessore.

Capitolo 2

Architetture ad Eventi

Si tratta ora di come alla programmazione asincrona si accosta un altro importante concetto: il concetto di evento. In un contesto di programmazione asincrona gli eventi si inseriscono in modo del tutto ottimale a completare il quadro, poiché essi stessi sono intrinsecamente di natura asincrona.

2.1 Definizione di evento

Nel linguaggio comune la parola evento designa *"un avvenimento, caso, fatto che è avvenuto o che potrà avvenire"*, in maniera analoga in informatica per *evento* si intende *"un'azione intercettata dal programma che possa essere gestita dal programma stesso"*. A titolo esemplificativo un evento potrebbe essere la pressione virtuale di un tasto da parte dell'utente durante l'interazione con l'iterfaccia grafica, oppure la pressione di un tasto sulla tastiera fisica, un segnale interrupt esterno, , la scadenza di un timer preimpostato o in maniera più astratta l' avvenuta ricezione di dati attraverso la rete, ma più in generale qualsiasi altra forma di "fatto accaduto, avvenimeto" che possa essere in qualche modo rilevata e poi gestita (*custom events*).

2.2 Gestione degli Eventi

All'interno di un sistema l' entità che può generare eventi viene chiamata sorgente o *event source*, mentre l'entità che si occuppa di gestire un evento verficatosi viene chiamato gestore o (event handler). Talvolta può essere

presente un terzo attore con il compito di intermediario tra la sorgente e il gestore chiamato dispatcher particolarmente utile nel caso in cui un singolo evento debba poter essere gestito da più event handler (*event loop*).

2.3 Event Loop

Il costrutto di programmazione *event loop* realizza la funzionalità di gestire gli eventi all'interno di un programma. Più precisamente l'event loop ciclicamente durante tutta l'esecuzione del programma mantiene traccia degli eventi che si sono verificati all'interno di una struttura dati a coda per poi processarli uno per volta richiamando l'event handler, qualora il thread principale sia libero.

Si riporta in pseudocodice (sintassi JavaSript like) una possibile implementazione di getore di eventi *event loop*.

Listing 2.1: Event Loop

```
1 while(true){
2   var ev = eventQueue.getNext()
3   process(ev);
4 }
```

2.4 Sistemi Event-Driven

I sistemi che modificano il proprio comportamento in seguito alla gestione di eventi vengono chiamati *event-driven system* e normalmente vengono impiegati quando vi è la necessità di gestire attività esterne asincrone. Gli eventi per loro stessa natura sono intrinsecamente asincroni, non è noto a priori l'ordine temporale con cui questi si manifestano.

Uno scenario tipico di event-driven system è quello in cui sono coinvolte le interfacce grafiche o GUI (Graphical User Interface) che deveno gestire una moltitudine diversa di possibili azioni provenienti dall'esterno (pressione di un tasto, chiusura di una finestra, movimento del mouse).

Si pensi all'utente che interagisce con un programma mediante la pressione di bottoni virtuali, non si conosce a priori quali e quante volte i bottoni verrano premuti e in che ordine. In aggiunta il programma potrebbe allo

stesso tempo dover gestire l'acquisizione di dati da più fonti esterne, per esempio da rete o da sensori, e anche qui non sono note le tempistiche, né l'ordine con cui i dati giungeranno dalle proprie sorgenti.

Allo stesso tempo ad un sistema di questo tipo è richiesta una costante interattività durante lo svolgimento dei vari compiti, per esempio l'interfaccia utente non deve bloccarsi, mentre si eleborano i dati acquisiti, oppure nell'attesa di ricevere le informazioni dalla rete deve poter comunque essere possibile svolgere un altro compito contemporanemente.

2.5 Sistemi Event-Driven e Programmazione Asincrona

Da questo semplice caso analizzato emerge in maniera chiara ed evidente la necesità di abbandonare il modello di esecuzione sequenziale tipico dei sistemi di elaborazione di tipo batch (I sistemi di tipo batch possono essere visti come l'esatto opposto dei sistemi event-driving).

I sistemi event-driving sono infatti strettamente connessi ad un altro importante concetto: quello di programmazione asincrona.

Spesso i termini event-driven system e programmazione asincrona parallela vengono utilizzati per riferirsi allo stesso concetto, mentre in realtà pur essendo sottile la differenza sono concetti diversi, infatti un programma può essere event-driven, ma essere eseguito da un singolo flusso di controllo (single threaded).

2.6 Principali Pattern

Si considerano ora i pattern principali adottati nella progettazione dei sistemi event-driven. Si descrive inizialmente il fondamentale *pattern observer* utilizzato nella programmazione *object oriented* poiché questo costituisce le fondamenta del pattern *event-listener* e del pattern *publisher-subscriber* utilizzati poi nella gestione degli eventi.

2.6.1 Pattern Observer

Nella programmazione ad oggetti l'intento del *pattern observer* è quello di definire una relazione tra uno e più oggetti così che, quando lo stato di un

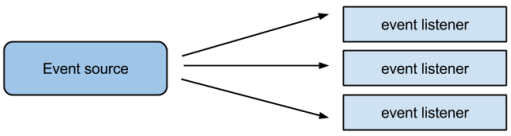

Figura 2.1: Pattern event listener.

oggetto (*object*) cambia, tutti gli altri oggetti dipendenti (*observers*) sono notificati automaticamente dell'avvenuto aggiornamento. La motivazione alla base del pattern observer è mantenere la consistenza tra gli oggetti correlati senza però creare un forte accopiamento tra essi.

La notifica del cambiamento verificatosi avviene quindi senza presupposti da parte del soggetto monitorato riguardo gli oggetti osservatori.

Il soggetto deve quindi poter tener traccia della lista degli osservatori e mettere a disposizione di questi un'interfaccia per registrarsi o cancellarsi dalla lista stessa. Gli osservatori da una parte implementerano un'iterfaccia per la notifica dell'aggiornamento, dall'altra l'oggetto dispone di un metodo per notificarli non appena sarà avvenuto il cambiamento di stato.

2.6.2 Pattern Event Listener

Come accennato in precedenza il pattern *event listner* è basato sul pattern observer. Questo pattern è adottato dalla maggioranza dei gestori di eventi nelle interfacce grafiche. Nel pattern event listener si distingue l' *event source* o sorgente e gli *event listener* o ascoltatori. Gli event listener devono essere notificati quando si verifica un evento nella sorgente, gli event listner si registrano quindi alla sorgente e implementano un'interfaccia che specifica il metodo che deve essere richiamato all'occorenza dell'evento. La sorgente è in grado di generare eventi e deve poter permettere agli event listner di registrarsi o cancellare la registrazione. Un'implementazione di quanto appena descritto è l'*event model AWT* di Java, utilizzato per la realizzazione delle interfacce utente. A differenza del pattern observer l'event model AWT definisce (nella versione 1.1) 11 differenti tipi di interfacce per gli ascoltatori, ciascuna dedicata ad un differente tipo di *GUI event*.

2.6.3 Pattern Publish-Subscribe

Il pattern *Publish-Subscribe* è anch'esso basato sul pattern *observer*. Le entità componenti interagiscono annunciando eventi: ciascuna componente si registra per la ricezione di notifica di una determinata classe di eventi rilevanti per il proprio scopo. Ogni entità componente può essere sia produttore, sia consumatore di eventi. Il pattern publish-subscribe disaccoppia produttori e consumatori di eventi, mittenti e destinatari dialogano infatti attraverso un tramite detto *dispatcher* o *broker*. Il mittente di un messaggio (*publisher*) non deve essere consapevole dell'identità dei destinatari (*subscriber*) esso infatti si limita a "pubblicare" il proprio messaggio al dispatcher. I destinatari si rivolgono a loro volta al dispatcher "abbonandosi" (*subscribe*) alla ricezione di messaggi, il dispatcher quindi inoltra ogni messaggio inviato da un publisher a tutti i subscriber interessati a quel messaggio. Il sistema di sottoscrizione consente inoltre ai subscriber di precisare in modo più speficifico a quali messaggi sono interessati. Per esempio un subscriber potrebbe registrarsi solo per la ricezione di messaggi provenienti da determinati publisher, oppure aventi certe caratteristiche. Questo schema implica che ai publisher non sia noto quanti e quali sono i subscriber e viceversa, contribuendo così alla scalabilità del sistema. Il pattern Publish Subscribe è ripreso nel Capitolo 5 dove si forniscono anche esempi in linguaggio JavaScript.

Capitolo 3

Applicazioni Moderne

Si chiarisce innanzitutto che con l'espressione "applicazioni moderne" in questa tesi si intendono principalmente le web application e le mobile application. Potendo poi estendere la linea di confine, si include anche almeno in parte il cloud computing, perché spesso queste tre cose sono strettamente collegate.

Nel capitolo oltre ad un'ampia trattazione degli argomenti sopra citati si prenderanno anche in considerazioni i dispositivi fisici che ne permettono la fruizione. Si mostreranno quindi con numerosi esempi come questi sistemi si prestano ottimamente ad essere programmati con le tecniche descritte nei capitoli precedenti, oggetto di studio di questa tesi.

3.1 Web Application

Con il termine web application si intende essenzialmente un'applicazione accessibile mediante il browser. Si tratta quindi di un'applicazione software scritta in un linguaggio supportato dal browser utilizzato normalmente per la navigazione sul web. Generalmente la maggior parte delle web application sfruttano un insieme di tecnologie in primis il linguaggio Javascript unitamente ad HTML5 e CSS. L'ubiquità odierna dei browser su un sempre più vasto numero di dispositivi costituisce un buon proposito per l'impiego per questa tipologia di sistemi software.

Oltre al fatto di poter essere accessibili da un numero vastissimo di potenziali utenti che utilizzano piattaforme differenti evitando l'installazione

di software sul sistema locale, le web application hanno anche come punto di forza il fatto di poter essere facilmente aggiornabili e manutenute.

Tutti questi sono i motivi chiave che hanno contribuito alla capillare diffusione in sempre più ambiti e settori diversi delle web application negli ultimi anni: dai programmi di gestione delle email, ai word processor fino a veri e propri programmi di disegno tecnico 2D e 3D o grafico-artistico.

Si ritiene opportuno citare a titolo d'esempio Gmail realizzato dall'azienda multinazionale Google pioneristica in questo campo. Gmail è un sistema di gestione di posta elettonica completamente accessibile attraverso il browser (web-based email) realizzato sfruttando intensamente Javascript e la tecnologia AJAX come sistema di comunicazione client-server.

3.1.1 Struttura delle Web Application

Le strutture delle web application sono generalmente organizzate a strati, ad ogni strato è assegnato un ruolo ben preciso.

Nel caso di web application più semplici è presente un solo strato che risiede interamente sulla macchina client. L'utente attraverso il browser scarica totalmente il codice dell'applicazione e tutte le varie componenti in locale e l'esecuzione non necessità della stretta interazione con altre parti al di fuori del browser stesso per il proprio funzionamento.

Aumentando il livello di complessità ci si riconduce generalmente ad una struttura a tre strati (three-tiered application), nella forma più comune questi tre strati sono chiamati presentazione, applicazione e immagazzinamento dati (storage).

Il browser concretizza il primo strato, quello di presentazione, ovvero la parte client-side che include l'interfaccia grafica quella mediante la quale l'utente ha accesso e interagisce con il sistema.

Il secondo strato, lo strato di mezzo, riguarda la parte server-side dove avviene l'elaborazione della logica principale dell'applicazione e le tecnologie usate, i linguaggi e i framework impiegati per realizzarlo sono i più disparati ASP, CGI, JSP/Java, PHP, Python, Ruby on Rails, Javascript.

Il terzo strato, ovvero lo strato di storage è generalmente un DBMS di tipo relazionale SQL, normalmente anche questo risiede interamente sul server in quanto lo storage a livello locale si limita a funzioni di cache e poco altro.

3.1.2 Vantaggi e Svantaggi delle Web Application

Tra i vantaggi principali delle web application come accennato sopra nell'introduzione si trova l'assenza di complicate procedure e tempi di attesa lunghi per il deployment delle stesse, l'unica cosa necessaria che deve essere presente in locale è un browser compattibile con le tencologie utilizzate. Il fatto di dover impiegare solo il browser risolve anche i problemi di compattibilità tra piattarforme diverse (cross-platform) non è piu necessario il porting del codice per sistemi operativi diversi.

Per il funzionamento non sono necessarie ingenti quantità di spazio sul disco sulla macchina client e non sono richieste procedure di aggiornamento (upgrade) da parte dell'utilizzatore finale poiché tutte le eventuali modifiche vengono effettuate sul server e automaticamente raggiungono l'utente.

Grazie al recente avvento della tecnologia HTML5, il progettista può realizzare ambienti interattivi a livello nativo all'interno del browser utilizzando quindi suoni, video e animazioni ampliando così ulteriormente il panaroma di settori diversi.

Tra gli svantaggi principali invece si ricorda l'assoluta necessità di un browser pienamente compattibile, spesso infatti nonostante gli standard, browser diversi o non aggiornati alle più recentei versioni realizzano medesime funzionalità in maniera differente. Questo costituisce un aggravio per il programmatore che non può non tenerne conto in fase di stesura del codice.

Un altro svantaggio importante è il fatto che la web application a meno dei casi più semplici ed elementari richiedono una connessione internet persistente per funzionare. L'assenza di connettività può infatti rendere l'applicazione del tutto inutilizzabile o utilizzabile solo in parte.

3.2 Mobile Application

Una mobile application (letteralmente applicazione mobile) è un'applicazione software progettata per funzionare su dispositivi mobili principalmente smartphones e tablet computer.

I moderni dispositivi mobili sono calcolatori elettronici di ridotte dimensione e peso che non supera i 900g , facilmente trasportabili dispongono di schermi sensibili al tocco (touchscreen) e tastiere miniaturizzate o virtuali.

Questi sitemi permettono un collegamento persistente con la rete internet mediante connessione senza fili Wi-Fi e collegamenti di telefonia mo-

bile, sono equipaggiati di svariati sensori tra i più importanti sensori di movimento, geolocalizzazione GPS, telecamere e microfoni.

Le applicazioni mobile sfruttano le risorse hardware in stretta dipendanza del sistema operativo di cui tutti i dispositivi mobile sono dotati, sono quindi a differenza delle mobile application dipendenti dalla piattaforma.

Le applicazioni mobile vengono distribuite attraverso internet mediante dei portali digitali che prendono il nome di App Store, veri e propri negozi e vetrine virtuali di software. Prima di poter essere acquistati dall'utente le applicazioni devono essere approvate dal distributore ovvero devono essere conformi a certi requisiti stabiliti dal contratto.

Le funzionalità offerte da questo genere di software sono le più disparate esattamente come i programmi per i personal computer (Desktop Application), in particolare negli ambiti di produttività generale e recupero informazioni. Per citare qualche esempio ricordiamo gestori di posta elettronica, calendari, rubriche, quotazioni dei titoli di borsa, gestione dei conti bancari, servizi meteo e simili.

3.2.1 Gestione degli Eventi in iOS

Si considera ora l'ambiente iOS a titolo esemplificativo di come gli eventi utente vengono gestiti in una mobile application. In iOS gli eventi sono oggetti che vengono inviati all'applicazione per informarla delle azioni compiute dall'utente e vengono generati ogniqualvolta l'utente interagisce con l'interfaccia grafica così come interagisce con il sensore accelerometro o utilizza un accessorio esterno. Esistono poi anche eventi generati dal cambiamento di posizione (in senso fisico) del dispositivo da un luogo ad un altro.

Il *main run loop* dell'applicazione è responsabile della gestione degli eventi generati dall'utente (*user related events*). L'oggetto UIApplication inizializza il main run loop all'inizio dell'esecuzione dell'applicazione che come suggerisce il nome stesso viene eseguito dal main thread ovvero il thread principale, questo assicura che gli user events siano processati serialmente nell'ordine con cui sono stati ricevuti.

La Figura 3.1 mostra l'architettura del main run loop e come gli eventi vengono gestiti trattati. Quando l'utente interagisce con il dispositivo fisico gli eventi correlati all'interazione sono generati dal sistema e consegnati all'applicazione mediante una speciale porta dall'*UIKit*. Gli eventi vengono

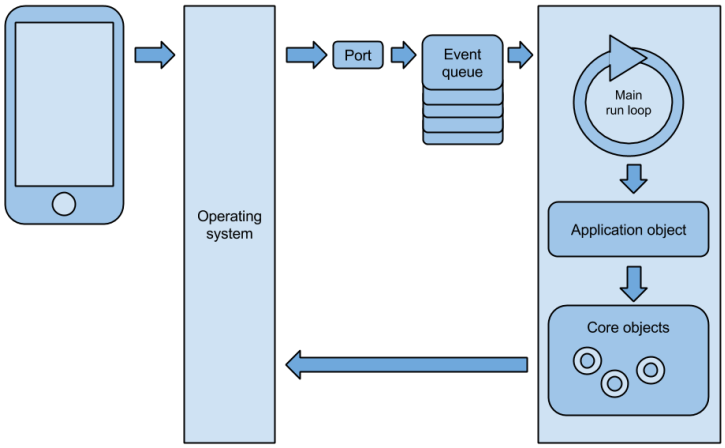

Figura 3.1: iOS Event cycle.

poi inseriti all'interno di una coda e gestiti uno per volta dal main run loop. L'oggetto UIApplication è il primo a ricevere l'evento e prende decisioni in merito alla ritrasmissione dello stesso, in un secondo passaggio ad un'altrà entità. Per esempio un evento di tipo *touch event* che corrisponde al tocco dello schermo è generalmente inviato dall'UIApplication all'oggetto *window object* che a sua volta lo invia alla *view* (oggetto che gestisce una certa area dellinterfaccia grafica) in cui è avvenuta la pressione. Altri tipi di eventi seguiranno quindi un percorso leggermente differente.

Ovviamente esistono anche altre categorie di eventi, infatti non solo gli eventi utente possono essere consegnati all'applicazione iOS, la maggior parte di questi è però sempre gestita dal main run loop ma possono esistere eccezioni. Gli eventi generati dall'accelerometro sono infatti consegnati direttamente ad un oggetto delegato (*delegate object*) specificato dal programmatore.

3.2.2 Vantaggi e Svantaggi delle Mobile Application

Tra i vantaggi delle mobile application si ricorda il supporto nativo dell'applicazione stessa che permette uno sfruttamento ottimale delle risorse di calcolo e memoria consentendo la realizzazione anche di programmi facenti uso di eleborazioni grafiche tridimensionali, specialmente nel campo videoludico (mobile gaming).

Il processo di distribuzione mediante gli App Store facilita anche la fase di deployment che non necessita di supporti fisici e permette di raggiungere potenziali clienti a livello globale, per lo stesso motivo anche le procedure di aggiornamento e manutenzione sono notevolmente semplificate, l'utente deve solo approvare il processo di aggiornamento tramite semplici procedure.

Il processo di revisione da parte del distributore intermediario, ovvero il gestore dell'App Store, contribuisce ad aumentare la sicurezza intrinseca evitando la diffusione di virus e applicazioni malware e simili, potenzialmente dannosi per il sistema ospitante.

Tra gli svantaggi si ricorda invece la scarsa portabilità ovvero la stretta dipendenza dell'applicazione all piattaforma per il quale è stata programmata, differenti dispositivi dotati di differenti sistemi operativi richiedono quindi la traduzione del codice da un linguaggio ad un altro (code porting) con il conseguente aggravio economico e tempo di sviluppo maggiore.

Le applicazioni per essere distribuite necessitano dell'approvazione da parte del gestore dell'App Store e questo comporta la conformità ad un regolamento stabilito dal gestore stesso, talvolta questi requisiti possono essere piuttosto restrittivi non permettendo quindi piena autonomia ai progettisti o limitando la libertà di scelta dei contenuti.

Capitolo 4

Implementazione in JavaScript

Si affrontano ora gli aspetti pratici della programmazione asincrona ad eventi nell'ambito delle web application e delle mobile application. Si è scelto Javascript come linguaggio di riferimento poiché attualmente è il più utilizzato nello sviluppo di applicazioni e sistemi web e allo stesso tempo è impiegato con successo anche nell'ambito mobile, pur non essendo un linguaggio nativo compilato ma interpretato. Si presta bene ad essere integrato con linguaggi come Objective-C. Inoltre JavaScript, intrinsecamente possiede tutte le caratteristiche che lo rendono linguaggio perfettamente idoneo per sistemi event-driven. I concetti di base della programmazione asincrona e delle architetture ad eventi ovviamente prescindono dal linguaggio utilizzato per l'implementazione ed hanno quindi valenza generale.

4.1 JavaScript

JavaScript è un linguaggio di scripting orientato agli oggetti, sviluppato in origine da Brendan Eich per la Netscape durante i primi anni di vita del web, standarizzato poi dalla ECMA con il nome di ECMAScript. JavaScript attualmente è il linguaggio più utlizzato per la programmazione web, la maggior parte dei siti web moderni fa uso più o meno intenso di questa tecnologia e tutti i web browser disponibili oggi comprendono un interprete JavaScript, rendendolo quindi di fatto il linguaggio più pervasivo esistente.

Javascript è un linguaggio interpretato, prototype-based, debolmente tipizzato e possiede funzioni di primo ordine, multiparadigma supporta lo stile di programmazione orientata agli oggetti, imperativo e funzionale.

4.1.1 JavaScript e Web Application

Nell'ambito delle web application il codice JavaScript per quanto riguarda il lato client viene eseguito dall'interprete incorporato nel browser, quando una pagina web viene caricata. Le interfacce che consentono a JavaScript di rapportarsi con il browser sono chiamate DOM (Document Object Model). Nelle normali pagine web questo meccanismo viene normalmente sfruttato per realizzare piccole funzioni di utilità, come controllare i valori nei campi di input, nascondere o visualizzare determinati elementi, tutte azioni che non sarebbero possibili con il solo utilizzo di HMTL. Un uso decisamente più intenso viene invece fatto nelle web application dove con JavaScript viene realizzata tutta la logica di funzionamento dell'applicazione stessa, insieme con la tecnologia AJAX è possibile trasferire i dati dal client al server e viceversa, sempre con JavaScript si gestisce l'intera interazione con l'utente ovvero l'interfaccia grafica.

4.1.2 JavaScript e Mobile Application

In un contesto di mobile application, JavaScript è utilizzato come linguaggio di scripting integrato all'interno di un programma ospite supportato a livello nativo dalla piattaforma. Il programma ospite fornisce un insieme ben definito di API che consentano l'interazione tra lo script e il programma ospite stesso. Lo script quindi quando eseguito utilizza questi riferimenti API per attuare operazioni specifiche non previste dai costrutti del linguaggio JavaScript in sé, esattamento lo stesso meccanismo che viene adottato anche in linguaggi come C o Java quando il programma si affida a librerie per svolgere funzioni non previste dal linguaggio che permettono di effettuare operazioni quali l'IO o l'esecuzione di chiamate a funzioni di sistema.

4.2 Eventi in JavaScript

I programmi JavaScript usano un modello di programmazione event-driven asincrono single-threaded. In questo stile di programmazione il browser, per esempio, genera un evento quando qualcosa accade al documento o al browser stesso o a qualche oggetto associato, per esempio il browser genera un evento quando finisce di caricare il documento, oppure quando l'utente muove il mouse sopra un hyperlink o quando preme un tasto sulla tastie-

ra. Gli eventi in JavaScript semplicemente sono occorrenze che il brower notifica al programma. Il tipo di evento (event type) è una stringa che specifica la natura dell'evento stesso. Il tipo mouse-move per esempio sta ad indicare che l'utente ha mosso il mouse, il tipo keydown che l'utente ha premuto un tasto, il tipo load indica invece che il documento è stato caricato completamente dalla rete.

Il termine event target indica invece l'oggetto dal quale si verifica l'evento o l'oggetto il cui evento è assocciato. L'oggetto Window, Document sono i più comuni event target nelle applicazioni client-side, ma in generale gli eventi possono essere prodotti anche da altri tipi di oggetti: per esempio l'evento readystatechange è prodotto dall'oggetto XMLHttpRequest che gestisce lo scambio di dati via rete.

Se il programma JavaScript deve occuparsi di gestire un particolare evento può registrare una o più funzioni che saranno invocate inseguito al verificarsi dell'evento oggetto di interesse. Questo tipo tipo di gestione non è una peculiarità di JavaScript o della programmazione web, bensì è un modello comune a tutte le applicazioni che fanno uso di interfacce grafiche.

Si definisce event handler o event listener una funzione che gestisce la risposta ad un evento. L'applicazione registra le funzioni event handler specificando l'event type e l'event target. Quando l'evento si manifesta sullo specifico event target viene invocata la funzione di gestione, queste funzioni prendono il nome di callbacks in quanto nella pratica vengono passate come parametri alle funzioni che effettuano la registrazione.

Esistono differenti modi per registrare gli event handler e i dettagli su come questi vengono svolte e i modi di gestione saranno oggetto di analisi dettagliata in seguito.

Un oggetto evento o event object in JavaScript è un oggetto associato ad un particolare evento e contiene dettagli circa l'evento stesso, gli event object sono passati come argomenti alle funzioni event handler. Tutti gli event object specificano l'event type e l'event target, per esempio un oggetto associato ad un evento del mouse include anche le coordinate del puntatore del mouse sullo schermo, oppure un oggetto associato ad un evento della tastiera contiene dettagli sullo specifico tasto premuto. In genere gli oggetti evento contengono poche altre informazioni oltre a quelle standard come il tipo e il target, ciò che più interessa di questi è solo il fatto che si siano verificati.

4.2.1 Scheduling degli Eventi

Come accennato in precedenza in JavaScript quando si vuole eseguire del codice in risposta al verificarsi di un evento, si fa uso delle callbacks. E' importante ora chiarire in che ordine queste vengano richiamate. Poiché i progammi JavaScript adottano il modello event-driven asincrono e single-threaded, il flusso di esecuzione non viene mai interrotto dal verficarsi dell'evento, bensì questo viene gestito mediante una coda e la corrispondente procedura di gestione eseguita solo dopo che sia terminata la parte attualmente in corso.

Listing 4.1: Scheduling eventi.

```
1 for (var i=1; i<=3;i ++){
2   setTimeout(function(){ console.log(i); }, 0);
3 }
```

Il semplice esempio di codice sopra riportato illustra quanto in precedenza detto. La funzione setTimeout stampa il valore della variabile i dopo 0ms, ovvero genera un evento che richiamerà la funzione di callback console.log al suo verificarsi. Poiché l'istruzione è inserita all'interno di un ciclo for verrà eseguita tre volte. Il risultato finale dell'operazione sarà quello di stampare tre volte il valore 4, questo potrebbe non sembrare ovvio a prima vista, vale la pena esaminarlo in dettaglio. La stampa del valore della variabile i dovrebbe avvenire esattamente nell'istante stesso in cui viene richiamata la funzione setTimeout, poiché il tempo è impostato su 0ms, di conseguenza intuitivamente ci si aspetta che vengano stampati in sequenza i valori di i, incrementati di volta in volta. In realtà se così fosse il flusso del thread verebbe interrotto violando quindi i presupposti del modello asincrono single-threaded. Ciò che accade realmente è il fatto che gli eventi timeout sono messi in coda e gestiti non appena è terminata l'esecuzione del ciclo for, di conseguenza al momento di stampare il valore la variabile è stata incrementata tre volte, il tempo di 0ms non è stato rispettato su una linea cronologica assoluta ma con valenza a partire dal momento in cui gli eventi potessero essere gestiti.

In maniera analoga vengono gestiti tutti gli altri tipi di eventi, per esempio se l'utente preme un tasto e questo deve avviare una certa procedura questa non verrà eseguita immediatamente in senso cronologico bensì l'e-

vento sarà messo in coda e gestito non appena il thread principale abbia terminato il codice precedente.

Questo meccanisco di scheduling degli eventi rende JavaScript un ottimo linguaggio di riferimento per la programmazione asincrona così come presentata in questa tesi, il codice non può mai essere interrotto gli eventi vengono messi in coda ed eseguiti non appena il thread principale è libero.

4.3 Pattern Publisher Subscriber

Si riprende brevemente il concetto di funzione sincrona e funzione asincrona, ricordando che il codice di una funzione sincrona viene eseguito immediamente quando questa viene chiamata e non può essere interrotta prima che termini, questo tipo di funzione viene anche definita bloccante. Diversamente una funzione asincrona come setTimeout o setInterval permette al codice della callback di essere eseguito in maniera differita.

Come esposto in precedenza, quando si vuole eseguire una certa procedura in risposta al verificarsi di un evento si assegna a quel particolare evento un event handler, quindi ad ogni evento corrisponde una procedura che lo gestisce. Questo sistema però mostra i propri limiti quando per un determinato evento è necessario attuare più procedure differenti e in numero differente. Si pensi ad esempio ad un elaboratore di testi realizzato mediante una web application ogni qualvolta l'utente preme un tasto questo deve essere visualizzato sullo schermo, la storico delle operazioni effettuate (intese come funzionalità del programma) deve essere aggiornato e sincronizzato con il server, cosi come il contenuto. Il correttore grammaticale deve correggere la nuova parola se attivato e altre funzioni simili devono essere effettuate in concomitanza. Dall'esempio appare evidente che un singolo event handler male si presta ad eseguire un compito simile, occorre quindi un modo per distribuire gli eventi quando una singola azione prevede l'esecuzione di più operazioni.

Il pattern architetturale publish subscribe o abbreviato PubSub è uno dei pattern fondamentali nella progettazione di sistemi event-driven in JavaScript e nelle architetture ad eventi più in generale. L'idea di fondo dietro questo pattern utilizzato per la comunicazione asincrona fra diversi oggetti o agenti è il forte disaccoppiamento tra le varie entità permettendo di scrivere codice event-driven più compatto e semplice da leggere.

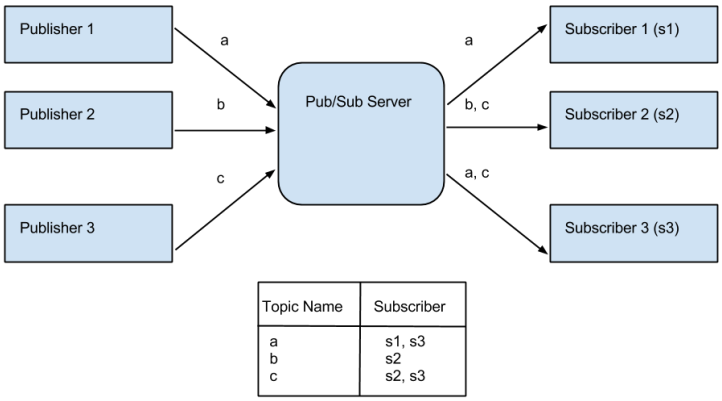

Figura 4.1: pattern Publisher Subscriber

Il pattern in questione prevede che mittenti e destinatari di notifica dell'evento dialogano attraverso un tramite chiamato *dispatcher* o *broker*. Il mittente del messaggio detto *publisher* non deve essere consapevole dell'identità dei destinatari *subscriber*, esso si limita a notificare (*publish*) l'evento avvenuto ai propri destinatari previa la loro registrazione *subscribe* alla ricezione della notifica.

Una o più procedure di callback devono essere eseguite al verificarsi di un evento generato da una sorgente, queste vengono quindi registrate in un elenco di entità da notificare. In seguito al verificarsi dell'evento le callback vengono richiamate e qualora queste non debbano più essere eseguite in seguito, possono cancellarsi dall'elenco degli osservatori.

4.3.1 PubSub in JavaScript

In JavaScript dall'anno 2000 è stato aggiunto alle specifiche del DOM il metodo addEventListener che implementa il pattern sopra descritto. Ogni oggetto del DOM quindi funge da sorgente e la funzione addEventListener() accetta in ingresso due parametri, il primo specifica il tipo di evento per quale l'handler si è registrato, il secondo parametro la funzione che deve essere eseguita all'occorrenza dell'evento. Qualora poi non sia più necessario

rimuovere una procedura registrata in precedenza si può utilizzare il metodo removeEventListener().

L'esempio seguente mostra come qualora vi sia la necessità di collegare più handler per gestire un medesimo evento utilizzando il pattern PubSub non sia necessario ricorrere alle callback innestate rendendo il codice più intricato e meno leggibile.

Listing 4.2: Event handler multipli - PubSub.

```
obj.addEventListener('click', handler1);
obj.addEventListener('click', handler2);
```

Listing 4.3: Event handler multipli.

```
obj.onclick = function(){
   handler1.apply(this, arguments);
   handler2.apply(this, arguments);
}
```

Si ricorda infine che il pattern PubSub per quanto versatile possa essere non sempre è la soluzione ottimale a tutti i problemi, come nel caso più semplice quando si ha un solo handler da gestire, oppure quando il risultato di una funzione asincrona che debba compiere un determinato compito necessita di essere gestito in maniera differente a seconda del successo o meno dell'operazione.

4.4 Pattern Promise

L'utilizzo del meccanismo di callback in larga scala all'interno di applicazioni particolarmente complesse porta ad avere a livello di codice complicanze non indifferenti. Un programma asincrono deve poter continuare l'esecuzione del codice in attesa dei risultati prodotti dalle funzioni asincrone, spesso però la chiamata ad una funzione deve avvenire in sequenza e successivamente al termine di un'altra. Queste condizioni forzano il programmatore a scrivere una gerarchia di callback innestate con il risultato di produrre codice confuso e complicato nonché poco elegante.

Si riporta un esempio esplicativo di quanto sopra scritto, in questo scenario di un' applicazione *chat* prevede tre chiamate asincrone: la prima

registra l'utente al server chat, la seconda chiede al server la disponibilità di una *chat room* disponibile in cui entrare e la terza invia un messaggio iniziale a tutti gli utenti presenti nella chat room. Il codice risulta complicato da capire per via delle callback innestate, necessarie queste per poter gestire eventi multipli che dipendono da altri.

Listing 4.4: Callback innestate.

```
registerToChatOnServer(username,function(rooms){
   joinAvailableRoom(rooms, function(roomname){
       sendChatToAll(roomname, msg, function(reply){
          showChatReply(reply);
       })
   })
})};
```

Un approccio risolutivo al problema imposto dalle callback e dato dal pattern *promises*. Ideate nel 1976 da D. Friedman e D. Wise il costrutto promises rappresenta un eventuale valore di ritorno dal completamento di una singola operazione. Una promise può trovarsi in tre stati: *non realizzato, realizzato, fallito* (unfulfilled, fulfilled, failed). Una promise può solo transitare da uno stato non realizzato allo stato realizzato oppure dallo stato non realizzato, allo stato fallito. La transizione può avvenire una sola volta e lo stato realizzato o fallito non può più essere cambiato, questa caratteristica di immutabilità delle promises previene possibili comportamenti non previsti.

4.4.1 Promise in Q.js

JavaScript non dispone di un sistema nativo per utilizzare le promise pertanto è necessario ricorrere all'uso di librerie esterne, in risposta al problema descritto nel listato precedente nell'esempio che segue si è utilizzata la libreria *Q*. Come si evince dal codice sotto riportato per prima cosa è stata effettuata la chiamata *Q.fcall* che crea una promise partendo dal valore restituito dall'invocazione di una chiamata asincrona successivamente si sono concatenate le altre varie funzioni utlizzando *.then* che a sua volta restituisce una promise.

Listing 4.5: Promise.

```
1
2  Q.fcall(registerToChatOnServer)
3   .then(joinAvailableRoom)
4   .then(sendChat)
5   .then(function(reply){
6     showChatReply(reply)
7     },function (error){
8     // gestisce eventuali errori delle operazioni asincrone
9  })
10 . done();
```

La funzionalità eseguita è la medesima del caso precende con il risultato di aver ottenuto un codice facilmente leggibile ed elegante.

4.4.2 Promise in jQuery

Un altro framework largamente utilizzato come ausilio alla programmazione ad eventi in JavaScript è jQuery, tra la varie e numero funzionalità che jQuery mette a disposizione del programmatore vi è anche il costrutto Promises che prende il nome di *Deferred*. Deferred è sostanzialmente un Promise con la sola differenza che in aggiunta permette di porterlo attivare direttamente, mentre Promise permette solo di aggiungere callback attivate da terze parti. Si riporta ora un caso di utilizzo per realizzare un semplice prompt che richieda all'utente di effettuare una scelta premendo i tasti *Y* oppure *N*.

Listing 4.6: Promise in jQuery.

```
1  var promptDeferred = new $.Deferred();
2  promptDeferred.always(function(){ console.log('Scelta
       effettuata:'); });
3  promptDeferred.done(function(){ console.log('Y key pressed.'); });
4  promptDeferred.fail(function(){ console.log('N key pressed.'); });
5
6  $('#playGame').focus().on('keypress', function(e) {
7    var Y = 121, N = 110;
8    if (e.keyCode === Y) {
9      promptDeferred.resolve();
10   } else if (e.keyCode === N) {
11     promptDeferred.reject();
```

```
12   } else {
13     return false; // Deferred in sospeso...
14   };
15 });
```

Come si intuisce dal listato la prima cosa effettuata è stata quella di creare un'istanza di $.Deferred che rappresenta la decisione dell'utente. Si attiva quindi il Deferred con i metodi *resolve* e *reject* in base al tasto premuto dall'utente. Il risultato che si otterrà dopo aver mandato il codice in esecuzione sarà quello di veder stampata per entrambe le opzioni la stringa *scelta effettuata* una delle altre due a seconda del tasto premuto. Se dopo aver premuto i tasti Y o N si ripete nuovamente l'azione come è giusto aspettarsi non accadrà nulla, in quanto le promise hanno valenza per una sola volta.

4.4.3 Combinazione di Promise

Nella maggior parte di librerie esterne, le promise sono astrazioni di alto livello e possono pertanto oltre ad essere concatenate passate ad altre funzioni e le loro operazioni composte facilmente contrariamente alle callback asincrone.

Un utilizzo piuttosto comune di combinazine di Promises è quello di determinare il completamento di un insieme di task asincroni. Si riporta quindi un esempio di questo possibili utilizzo facendo uso del framework JQuery.

Listing 4.7: Combinazione di Promise.

```
1 $.when(promise1, promise2)
2 .done(function(promise1Args, promise2Args){
3   // si gestiscono i valori restitutiti
4 });
```

Nell'esempio precedente in seguito alla buona riuscita delle due promises vengono passati come parametri i valori di ritorno alla funzione argomento del metodo *done*.

Si riporta un altro esempio utilizzando la libreria Q, il compito è quello di calcolare in modo asincrono due numeri di Fibonacci e stamparne la somma dei risultati.

Listing 4.8: Combinazione di Promise.

```
fibpromise = Q.all([
  computeFibonacci(n-1), computeFibonacci(n-2)
]);
fibpromise.spread(function (result1, result2) {
  console.log(result1 + result2);
}), function(err){
 //si gestiscono eventuali errori
});
```

Capitolo 5

Programmazione Asincrona in Sistemi Multiprocessore

In questo capitolo si discute di come si possano sfruttare le capacità di calcolo dei sistemi multiprocessore mantenendo allo stesso tempo l'efficienza e la facilità d'uso del modello asincrono.

5.1 Eventi e Concorrenza

Nel corso della tesi si è descritta la programmazione sincrona *single threading* come un'alternativa alla programmazione *multithreading* concorrente. Nello specifico la programmaizone ad eventi asincrona sostituisce un tipo di programmazione concorrente in cui più parti del programma vengono eseguite simultaneamente da differenti threads che hanno accesso ai medesimi dati in memoria dando così origine al problema delle corse critiche. Allo stesso tempo è diventato ormai essenziale poter sfruttare i diversi *core* delle CPU moderne poiché in certi ambiti prestazioni simili come quelle messe a disposizione da quest'ultimi non possono essere più raggiunte con un singolo processore *monocore*.

Esiste però un approccio alla programmazione multithreading che ben si concilia con il modello asincrono ad eventi, questa metodologia prevede che si distribuiscano compiti separati su core differenti mediante thread che solo occasionalmente si sincronizzano il tutto gestito da un unico *master thread* che può scambiare messaggi con gli altri thread assimilabili quindi ad eventi

permettendo quindi di fatto lo stesso tipo di gestione che si avrebbe con le normali operazioni IO viste in precedenza.

5.2 Web Workers

Grazie alla nuova tecnologia HTML5 in JavaScript è possibile sfruttare il meccanismo della programmazione asincrona multithreading così come è stata discussa in precedenza utilizzando i *web workers*. I web workers sono a tutti gli effetti thread paralleli, funzionano in un ambiente isolato *self contained* non hanno accesso ad elementi condivisi, non possono modificare il DOM e l'unica forma di comunicazione permessa è quella attraverso il passaggio asincrono di messaggi con il thread principale. I web wokers permettono quindi di poter scrivere funzioni che richiedano un tempo di esecuzione relativamente lungo senza però bloccare l'intera applicazione rendendola inutilizzabile. Il codice all'interno del thread worker viene eseguito in modo sincrono partendo dall'inzio e arrivato alla fine il worker entra in una fase asincrona in cui risponde agli eventi e ai timers. Se per esempio il worker registra un handler per l'evento *onmessage* potrebbe non terminare mai l'esecuzione qualora l'evento registrato non si manifesti affatto se invece non necessita di ricevere messaggi termina non appena siano esaurite e completate tutte le altre attività pendenti e chiamate le relative callback.

La creazione di un nuovo web worker avviene utlizzando il costruttore *Worker()* e specificando come parametro il codice da eseguire all'interno del thread. Una volta che si sia istanziato l'oggetto worker si possono passare i dati a questo utilizzando il metodo *postMessage()* il valore passato al worker attraverso il metodo *postMessage* verrà prima clonato e poi consegnato al worker questo perché come spiegato in precedenza non si possono avere zone di memoria condivise potenziali cause di corse critiche. Parallelamente si potranno ricevere messaggi dal worker registrando l'evento *message* sull'oggetto worker.

Listing 5.1: Passaggio di messaggi.

```
1 worker.onmessage = function(e) {
2   var message = e.data; // recupera il messaggio dall'evento
3   console.log("Message contents: " + message); // gestisce il
        messaggio
4 }
```

Eventuali errori o eccezioni che possono verificarsi durante l'esecuzione del worker, se non gestiti dal worker stesso, possono essere recuperati gestendo l'evento *onerror* come illustrato sotto.

Listing 5.2: Gestione Errori.

```
1 worker.onerror = function(e) {
2    console.log("Error at " + e.filename + ":" + e.lineno + ": " +
         e.message);
3 }
```

Analogamente a tutti gli altri tipi di evento gli oggetti Worker definiscono i metodi standard *addEventListener()* e *removeEventListner()* utilizzabili qualora sia necessario gestire più event handler. L'oggetto Worker inoltre dispone anche del metodo *terminate()* il quale attua l'interruzione immediata dell'esecuzione del thread.

I thread Worker così come sono stati appena descritti sono anche detti *dedicated workers* essi sono associati o dedicati ad un singolo *parent thread*. Le specifiche dei Web Worker tuttavia definiscono anche un'altra tipologia di worker gli *shared worker*. Attualmente però nessun brower implementa quest'ultima categoria che comunque si ritiene opportuno citare per completezza. L'idea di fondo degli shared worker è che questi siano pensati come risorse computazionali che erogano servizi agli altri thread interessati mediante una connessione, in sostanza l'interazione con gli shared worker avviene in maniera analoga alla comunicazioni con un server attraverso un *network socket* nota come *MessagePort*.

5.2.1 Esempi di Utilizzo

Come primo esempio di utilizzo dei Web Worker si riporta il caso che mostra come sia possibile scrivere codice sincrono evitando lo stallo dell'intero programma JavaScript. Lo scopo del codice sotto riportato è quello di eseguire una richiesta sincrona HTTP mediante l'oggetto *XMLHttpRequest*, per quanto questa possa essere una pratica decisamente sconsigliabile è perfettamente giustificabile all'interno di un Web Worker. Nell'esempio il worker riceve un elenco di URLs da cui recupare i dati e restituirli poi attraverso un array di stringhe al thread principale.

Listing 5.3: AJAX sincrono.

```
//Il codice seguente è caricato con un oggetto Worker ed eseugito
//come un thread indipendente

onmessage = function(e) {
  var urls = e.data; // URLs di input
  var contents = []; // Array di output
  for(var i = 0; i < urls.length; i++) {
    var url = urls[i];
    var xhr = new XMLHttpRequest();
    xhr.open("GET", url, false);
    xhr.send();
    if (xhr.status !== 200)
       throw Error(xhr.status + " " + xhr.statusText + ": " + url);
    contents.push(xhr.responseText);
  }

  //L'array viene passato al thread principale
  postMessage(contents);
}
```

Il secondo esempio che si riporta mette in risalto la capacità di poter effettuare operazioni che richiedono calcoli intensivi come l'*image processing* senzo compromettere la reattività dell'intero programma; un'operazione simile in assenza dei Web Worker sarebbe stata impossibile da realizzare in JavaScript sul lato client. Il risultato è quello di ottenere un effetto di tipo *blurr motion* di un immagine. La funzione di partenza *smear()* accetta come parametro di ingresso un elemento HTML di tipo *img*, ne estrae i pixel, crea un oggetto di tipo *ImageData* e lo passa al web worker che si occupa di processare l'immagine.

Listing 5.4: Calcoli intensivi.

```
function smear(img) {
  var canvas = document.createElement("canvas");
  canvas.width = img.width;
  canvas.height = img.height;
  var context = canvas.getContext("2d");
  context.drawImage(img, 0, 0);
  var pixels = context.getImageData(0,0,img.width,img.height);
```

```
8   var worker = new Worker("SmearWorker.js"); // Crea il worker
9   worker.postMessage(pixels); // Copia e invia i pixels
10  // Registra un handler per gestire la risposta del worker
11  worker.onmessage = function(e) {
12     var smeared_pixels = e.data;
13     context.putImageData(smeared_pixels, 0, 0);
14     img.src = canvas.toDataURL();
15     worker.terminate();
16     canvas.width = canvas.height = 0;
17  }
18 }
19
20 //WEB WORKER
21
22 //Riceve un oggetto ImageData dal thread principale, lo processa
       e lo rinvia
23 onmessage = function(e) { postMessage(smear(e.data)); }
24
25 function smear(pixels) {
26   var data = pixels.data, width = pixels.width, height =
         pixels.height;
27   var n = 10, m = n-1;
28   for(var row = 0; row < height; row++) {
29     var i = row*width*4 + 4;
30     for(var col = 1; col < width; col++, i += 4){
31       data[i] = (data[i] + data[i-4]*m)/n;
32       data[i+1] = (data[i+1] + data[i-3]*m)/n;
33       data[i+2] = (data[i+2] + data[i-2]*m)/n;
34       data[i+3] = (data[i+3] + data[i-1]*m)/n;
35     }
36   }
37  return pixels;
38 }
```

Capitolo 6

Conclusioni

Nel corso della tesi sono stati affrontati tutti i temi principali riguardanti le architetture ad eventi. La programmazione asincrona è stata inquadrata in un'ottica di giusto compromesso, tra facilità di implementazione e performance computazionali. In situazioni che richiedono un elevato numero di operazioni IO e scarso impiego di CPU i benifici apportati da un approccio di computazione asincrona single thread superano addirittura quelli apportati dall'approccio multitasking, perché si elimina l'overhead dovuto alla gestione dei thread da parte del sistema operativo. Si sono trattati poi gli aspetti legati all'introduzione degli eventi in un contesto di asincronicità di esecuzione dei programmi. Utilizzando JavaScript si è messo in luce come i linguaggi di programmazione oggi più diffusi per la realizzazione di applicazioni mobile si rendono velocemente inadatti allo scopo con il crescere della complessità funzionale del programma stesso. La gestione degli eventi mediante l'uso di callback porta a scrivere codice complicato di difficile lettura e difficile manutenzione (*spaghetti coding*), pertanto diviene necessario ricorrere a framework e librerie esterne per aggirare il problema quanto più possibile. Si conclude quindi che gli sforzi futuri che verrano posti nelle nuove implementazioni dei linguaggi di programmazione utilizzati nei sistemi event-driven, si orientino verso la risoluzione di queste problematiche in modo da poter offrire al progettista la possibilità di scrivere codice efficiente, di facile leggibilità ed elegante.

Bibliografia

[1] Trevor Burnham., *Async JavaScript*, The Pragmatic Bookshelf. , Dallas, USA, 2012.

[2] David Flanagan., *JavaScript: The Definitive Guide, Sixth Edition*, O'Reilly Media. , Sebastopol, USA, 2011.

[3] Opher Etzion and Peter Niblett., *Event Processing IN ACTION*, Manning. , Stamford, 2011.

[4] Joseph Albahari and Ben Albahari., *C# 5 in a nutshell*, O'Reilly Media. , Sebastopol, USA, 2012.

[5] K. Kambona, E. Gonzalez Boix and W. De Meuter, *An Evaluation of Reactive Programming and Promises for Structuring Collaborative Web Application*, Dayla. , Montpellier, France, 2013.

[6] Dave Peticolas. Twisted Introduction, 2012. `http://krondo.com/?page_id=1327`.

[7] The Main Run Lopp. iOS App Programming Guide, 2013. `http://https://developer.apple.com`.