UNIVERSITÀ DI PISA
Facoltà di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea Specialistica in Tecnologie Informatiche

Migrazione dello stato JavaScript di un'applicazione web

Relatore:
Prof. Fabio Paternò

Contro Relatore:
Prof. GianLuigi Ferrari

Tesi di Laurea di:
Federico Bellucci

Anno Accademico 2009/2010

Indice

Parte I.

Introduzione

1. Introduzione

Con l'avvento del Web 2.0 i siti internet stanno divenendo sempre più evoluti ed intelligenti. In alcuni casi si tenta di fornire al fruitore della pagina web la sensazione di utilizzare un'applicazione desktop, dando vita alle cosiddette *applicazioni web*. GMail, Flickr e Meebo sono solo alcuni esempi di come tali applicazioni siano ormai divenute di uso comune nel web.

Alla base di queste applicazioni ci sono varie rivoluzioni tecnologiche che hanno stravolto i formalismi del web design:

Separazione di vista, modello e controllo

Ormai le pagine web non sono più composte da un unico documento contenente HTML, scripts e stili, ma da molti documenti diversi, con l'obiettivo di separare e rendere indipendenti i tre componenti del modello MVC (Model View Controller). Ciò ha consentito la realizzazione di siti web in cui le tre componenti venissero sviluppate parallelamente ed un sempre maggiore riuso di componenti già pronte. Negli ultimi anni infatti sono proliferati frameworks per lo sviluppo di applicazioni web che facilitino la creazione di siti avanzati e dinamici consentendo al web master di concentrare gli sforzi sui contenuti e l'usabilità, piuttosto che sulla parte tecnica. Come caso estremo si considerino ad esempio i CMS, Content Management System, che hanno reso possibile la creazione di siti web dinamici e dall'aspetto accattivante, solitamente blog, anche ad utenti che non abbiano alcuna conoscenza di HTML, CSS, JavaScript e PHP.

Spostamento della logica dal server al client

Dato che gli utenti del web hanno a disposizione terminali sempre più potenti, non solo desktop ma anche mobili, gli sviluppatori hanno progressivamente spostato la logica dal server al browser presente nel client. Così facendo si alleggeriscono il carico di lavoro e il traffico sul server e si sfrutta la potenza di calcolo del client, eliminando al tempo stesso le latenze dovute alla richiesta e generazione delle pagine web sul server.

AJAX

AJAX è l'acronimo di "Asynchronous JavaScript and XML" e, come si evince dal nome, non è di per se una tecnologia, ma un insieme di linguaggi e di protocolli atti a raggiun-

gere un obiettivo, quello della definizione di siti web dinamici attraverso la richiesta "al volo" di contenuti. Nello scenario più comune, durante la navigazione, l'utente compie, esplicitamente o implicitamente, richieste asincrone ad un server, il quale risponderà inviando informazioni formattate in XML che il browser utilizzerà, grazie all'intervento di JavaScript, per modificare dinamicamente l'aspetto o i contenuti della pagina web.

Per quanto le applicazioni web tentino di avvicinarsi alla loro controparte desktop, questa somiglianza rimane un'approssimazione dato che alcune caratteristiche vengono implementate solo parzialmente se non addirittura affatto.

Alcuni siti web non consentono ad esempio di salvare la sessione di lavoro, se non in ben determinati punti (ad esempio la conferma di acquisto in un negozio elettronico), rendendo così necessario ricominciare da capo la sessione di lavoro quando l'utente abbia necessità di spostarsi su un'altra macchina o semplicemente di interromperla e di riprenderla in un secondo momento.

1.1. Obiettivo

Con la presente tesi il candidato si pone l'obiettivo di estendere la piattaforma di migrazione OPEN (capitolo 2) implementando il salvataggio e il ripristino dello stato degli script JavaScript di un'applicazione web.

L'obiettivo può quindi essere scomposto in tre problemi principali:

- determinare una rappresentazione per lo stato JavaScript dell'applicazione web;
- salvare lo stato del dispositivo sorgente prima di effettuare la migrazione;
- ripristinare lo stato nel dispositivo destinatario al termine della migrazione.

1.2. Struttura della tesi

La prima parte della tesi tratterà brevemente i concetti e le tecnologie che verranno considerati come noti nei capitoli della seconda parte. Il capitolo 2 espone la struttura e le funzioni della piattaforma di migrazione. Il capitolo 3 analizza alcuni dei concetti più importanti legati a JavaScript, il linguaggio adottato come standard de facto per lo scripting web, mentre il capitolo 4 parla di JSON, il linguaggio di interscambio utilizzato per salvare e ripristinare lo stato dell'applicazione.

Nella parte II è invece esposto il lavoro svolto. Nei capitoli 5 e 6 viene spiegato come sono stati scelti il modo e il formato di salvataggio dello stato. Il capitolo 7offre una

breve panoramica sui problemi affrontati, che verranno poi analizzate in dettaglio nei capitoli 8 e 9. Infine nei capitoli 10 e 11 vengono esposti i risultati della tesi in due esempi di applicazioni JavaScript, *JavaScript-PacMan*^{plus} e *JsTetris*.

1.3. Scenario

Vediamo un esempio di scenario in cui si rende necessaria la migrazione dello stato JavaScript di un'applicazione web.

> «Alice si sta recando a lavoro in autobus. Per passare il tempo si collega con il suo palmare ad un sito web per giocare ad una versione JavaScript del Pacman.
>
> Arrivata in ufficio in anticipo e non avendo lavori da sbrigare, Alice gioca ancora un po', ma decide di migrare la sua partita sul PC desktop che usa abitualmente per lavoro, in modo da poter sfruttare la tastiera e il monitor per una migliore esperienza di gioco.
>
> Pochi minuti dopo riceve una chiamata in cui le comunicano di recarsi alla sala conferenze, così Alice migra nuovamente la partita sul suo palmare, approfittando degli ultimi minuti liberi...»

Vedi figura 10.1 a pagina 76 per un'immagine del Pacman.

1.4. Lavori correlati

Controllo remoto

In "Highlight: A System for Creating and Deploying Mobile Web Applications" [17] si vuole fornire ai dispositivi mobili, in particolare a quelli dotati di risorse più modeste, la possibilità di visualizzare siti web dinamici attraverso un'interfaccia semplificata, costruita ad hoc per un determinato sito web e per uno specifico utilizzo. Tali applicazioni ad hoc vengono costruite con un apposita IDE e sono programmate in JavaScript. Un applicazione ad esempio può essere un utility per vedere i profili degli amici di FaceBook che sono attualmente online.

Il sito web originale viene "navigato" in un proxy a cui il client mobile accede in remoto. Ogni interazione dell'utente con il client mobile - come il click di un link - si traduce in una procedura remota sul proxy che si occuperà di interpretare la richiesta interagendo con il sito web ed inviando i risultati al client mobile.

Questo meccanismo consentirebbe quindi di migrare la sessione di lavoro di un utente, ma solo per i siti web per cui siano state realizzate a priori delle "interfacce Highlight". La piattaforma di migrazione utilizzata in questa tesi consente invece di migrare qualsiasi sito web, la cui interfaccia viene generata al volo adattandola al dispositivo destinatario della migrazione, non richiedendo quindi uno sviluppo di interfacce ad hoc per ogni sito che si vuole migrare.

Suspend and resume

In "Internet Suspend/Resume" [15] viene proposto l'utilizzo di due tecnologie note: i *Virtual Machine Monitor* (VMM) e i *File System distribuiti*, per consentire ad un utente di sospendere la sua sessione di lavoro - come se chiudesse il monitor del portatile - e di riprenderla anche a miglia di distanza, magari su un'altra macchina.

Il file system distribuito consente di migrare lo *stato persistente*, ovvero di accedere ai propri file e cartelle in remoto come se fossero in locale. Mentre la VMM si occupa dello *stato volatile*, ovvero consente di avere un'astrazione dell'architettura hardware e software e di incapsulare tutti i file temporanei a cui il sistema operativo accede durante la sessione di lavoro dell'utente.

A differenza della piattaforma di migrazione utilizzata nella tesi, un'architettura *ISR* (Internet Suspend/Resume) consente potenzialmente di preservare (e di migrare) lo stato dell'intero sistema operativo, e non solo di un sito web.

Per contro ha dei requisiti software molto pesanti: oltre ad un server per conservare le sessioni di lavoro in sospeso, richiede che su ogni client sia presente un sistema operativo *host* dotato di file system distribuito e di una VMM, all'interno della quale sia installato un sistema operativo *guest*. Gli utenti che desiderino effettuare una migrazione all'interno della nostra piattaforma di migrazione non necessitano invece di altri software oltre al browser.

Altri lavori correlati sono citati nel capitolo 5, "Modi di salvataggio", in cui si analizzano i vari modi per salvare lo stato JavaScript di un'applicazione web in modo da poterlo caricare in un diverso dispositivo.

2. Piattaforma di migrazione

Il lavoro di questa tesi si inserisce all'interno di una piattaforma di migrazione realizzata per il progetto OPEN[1] [20, 13] dal laboratorio HIIS[2] del CNR di Pisa. La piattaforma consente ad un utente che si trovi al suo interno di interrompere la sessione di lavoro avviata in un sito web e di riprenderla in un altro dispositivo, eventualmente molto diverso sia per software che per hardware. Per fare in modo che il passaggio da un dispositivo all'altro sia il più possibile fluido, la struttura della pagina viene modificata in modo da adattarsi al dispositivo di destinazione (che ad esempio potrebbe avere uno schermo molto piccolo, un ridotto numero di colori o un'interfaccia sonora per consentirne l'uso ai non vedenti) e in modo che l'utente possa interagire con la pagina nel dispositivo di destinazione allo stesso modo in cui vi interagiva nel dispositivo di partenza.

La piattaforma è in grado di gestire anche pagine generate da linguaggi come JSP, PHP ed ASP, in quanto la migrazione prende in considerazione una pagina generata sul client, ovvero ciò a cui l'utente ha effettivamente accesso. La piattaforma prevede anche un supporto minimo per il JavaScript, in particolare per la conservazione degli handler associati a nodi del DOM.

In pratica la piattaforma fornisce ad ogni sito la possibilità di migrare, indipendentemente da come è realizzato e senza che lo sviluppatore ne debba tener conto.

2.1. Architettura

Lo schema dell'architettura è mostrato in figura 2.1. Per poter accedere alla piattaforma di migrazione l'utente deve utilizzare nei suoi dispositivi un Client Software[3] che dovrà notificare (1) la presenza del dispositivo al Migration Server.

[1] Open Pervasive Environments for migratory iNteractive Services: http://www.ict-open.eu/
[2] Human Interfaces in Information Systems: http://giove.isti.cnr.it/
[3] Nei dispositivi dotati di "tabbed" browser, è possibile utilizzare un client web, completamente platform-independent, che può essere aperto in un tab del browser.

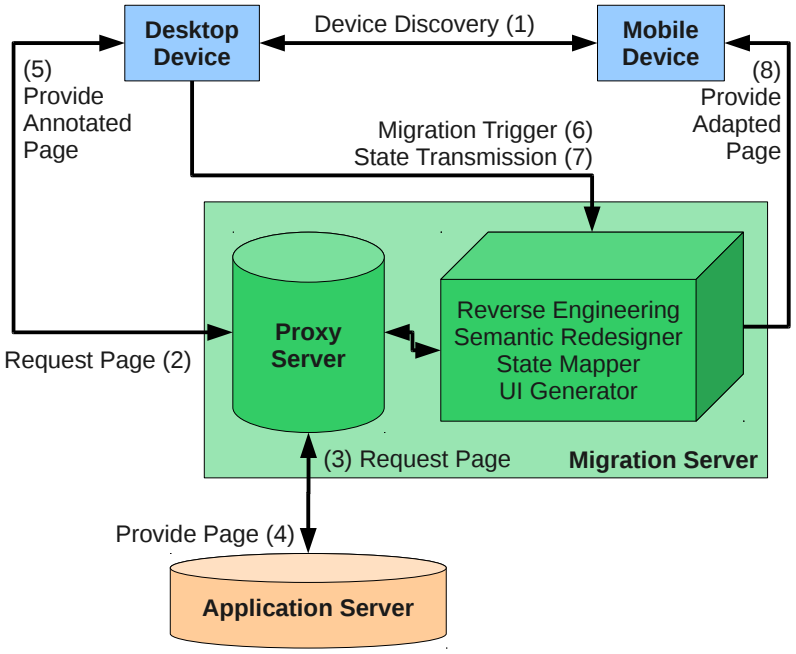

Figura 2.1.: Architettura della piattaforma di migrazione (tratta da [13])

Il Migration Server integra anche un Proxy che fa da intermediario tra il browser dell'utente e l'Application Server, annotando la pagina prima di inviarla all'utente e arricchendola degli script necessari per innescare la migrazione (2, 3, 4, 5).

Attraverso il Client Software, l'utente può attivare la migrazione (6), ovvero invocare gli script di migrazione integrati nella pagina, che si occuperanno di salvare lo stato corrente dell'applicazione (elementi selezionati, testo inserito, cookies, scripts...) e di inviarlo al Migration Sever mediante uno script AJAX (7). A questo punto il Migration Server, attraverso i suoi vari moduli, genera ed invia al Client una pagina adattata per il dispositivo di destinazione, annotata con gli script di migrazione e consistente con lo stato della pagina originale (8).

2.2. Struttura

La modifica della pagina viene effettuata durante una serie di passaggi, ciascuno implementato da un diverso modulo del Migration Server.

- *Reverse Engineering*: costruisce la descrizione logica della pagina desktop presa in considerazione

- *Semantic Redesigner*: trasforma la descrizione logica della pagina desktop in un'altra adattandola al dispositivo di destinazione
- *State Mapper*: associa lo stato della pagina web corrente alla descrizione logica automaticamente generata per il dispositivo destinatario
- *User Interface Generator*: genera l'implementazione dell'interfaccia corrispondente. Tale implementazione è quindi inviata al dispositivo destinatario in modo che l'utente possa immediatamente accedere alla pagina adattata, con lo stato risultante dalle interazioni con il dispositivo sorgente.

3. JavaScript

La fase ascendente di JavaScript inizia parallelamente alla decadenza delle applet Java, che si mostrarono inadatte alla costruzione di applicazioni web a causa delle pesanti restrizioni di sicurezza che imponevano agli sviluppatori. Nel 1995 Netscape fu la prima azienda ad inserire tale linguaggio nel proprio browser, trasformandolo nel linguaggio ufficiale di scripting nel web. Seppure sia stato esteso frettolosamente per far fronte alle esigenze dovute alla sua repentina espansione nel mercato, alla sua base ci sono idee e tecnologie ereditate da altri linguaggi che lo rendono estremamente potente e flessibile.

Nonostante il nome, JavaScript ha ben poco da condividere con Java, avendo invece molti più punti in comune con linguaggi come Lisp e Scheme. Douglas Crockford, uno dei massimi esperti del linguaggio, lo ha definito come "Lisp vestito da C" [11].

Oltre alla sua integrazione in tutti i maggiori browser web, ha contribuito alla sua diffusione la sua estrema facilità d'uso, che consente di ottenere risultati soddisfacenti in tempi brevissimi, a discapito ovviamente della bontà del codice. Proprio per questo si è creata intorno a JavaScript una fama di linguaggio relegato al web e di scarsa qualità. Fama che negli ultimi anni si sta progressivamente stemperando grazie alla creazione di framework che consentano di programmare ad alto livello (JQuery, Prototype, Dojo...), interpreti utilizzabili anche al di fuori del browser (Rhino) e ambienti di sviluppo che ne facilitino la scrittura (Aptana studio).

Di seguito elenchiamo alcune delle caratteristiche che lo contraddistinguono:

dinamico si possono definire a tempo di esecuzione proprietà di un oggetto, funzioni, etc...

debolmente tipato come altri linguaggi di scripting, non si deve (e non si può) specificare il tipo di una variabile, che può inoltre essere cambiato in qualsiasi momento dell'esecuzione del programma;

variabili globali questa è una caratteristica che contribuisce alla facilità d'uso di JavaScript ma al tempo stesso anche alla difficoltà di creare codice portabile e mantenibile;

object oriented JavaScript consente di dichiarare classi ed istanziare oggetti con una sintassi simile a quella dei linguaggi di programmazione derivati da C, ma tale

sintassi è solo una maschera che nasconde meccanismi diversi, fra cui l'eredità prototipale;

eredità prototipale ogni oggetto può essere usato come base per costruire altri oggetti. Modificando il prototipo di un "costruttore" di oggetti, tali modifiche si propagheranno a tutti gli oggetti creati (prima e dopo) a partire da quel costruttore;

oggetti funzione le funzioni in JavaScript sono oggetti di prima classe che possono quindi essere passate come valori e dotate di membri e metodi;

chiusure lessicali una delle caratteristiche più potenti di JavaScript, che verrà analizzata in dettaglio nella sezione 3.3.3, è la possibilità di associare ad una funzione uno stato, eventualmente inaccessibile dal programma se non tramite quella funzione.

3.1. JavaScript nel web

Non esiste sito web che si rispetti che non faccia uso di JavaScript. Agli albori del suo sviluppo, JavaScript veniva usato per semplici funzioni, cablate direttamente all'interno del codice HTML della pagina web, come ad esempio la visualizzazione di alert o di popup o la validazione di campi di form. Grazie al suo stretto legame con il DOM e alla standardizzazione ufficiosa dell'oggetto XmlHttpRequest (che implementa AJAX), le sue possibilità di utilizzo si sono notevolmente ampliate, consentendo grazie al primo di modificare al volo il contenuto e lo stile della pagina web e grazie al secondo di fare richieste al web server ed aggiornare la pagina senza doverla ricaricare.

JavaScript è stato standardizzato dall'istituto ECMA nel 1997, sotto lo standard ECMA-262 e ISO/IEC 16262. Attualmente la quinta edizione dello standard è sotto analisi dell'ISO - ma di fatto la terza edizione è quella universalmente utilizzata e ritenuta evidentemente sufficiente alle attuali esigenze di mercato. Purtroppo ogni browser fornisce la sua implementazione di EcmaScript ma, fortunatamente, esiste un'intersezione comune tra le varie implementazioni, che corrisponde grosso modo alla sua terza edizione, rendendo possibile la realizzazione di codice indipendente dal browser.

La tabella 3.1 [1] mostra quali sono i motori di rendering (*layout engine*[1]) integrati

[1] Un layout engine è un modulo software che, preso in ingresso un documento scritto con un linguaggio di markup (come (X)HTML, XML o SVG) e uno o più documenti per la formattazione (CSS, XSL...), visualizza la pagina formattata in una finestra, solitamente del browser, ma anche in altri programmi, come i client di posta, le applicazioni di aiuto online o qualsiasi programma in grado di visualizzare contenuti web. [6]

Layout engine	Release version	Used by
Amaya	11.3.1	Amaya
Gecko	1.9.2.6	All Mozilla software, including Firefox; SeaMonkey and Galeon; Camino; K-Meleon; Flock; Epiphany-gecko; GNU IceCat; Debian Iceweasel, Icedove, Iceape and Iceowl
KHTML		Konqueror
Presto	2.6.30	Opera; Opera Mobile; Nintendo DS & DSi Browser; Internet Channel
Prince	7.1	Prince XML
Trident	4.0 (IE 8), 5.0 (IE 9)	Internet Explorer and other Internet Explorer shells like Maxthon (Microsoft Windows operating systems only)
WebKit	533	Google Chrome, Maxthon 3, Safari (including OS X for desktops and iOS for iPhones and iPads), Shiira, iCab 4, OmniWeb 5.5+, Epiphany, Adobe AIR, Midori, Adobe Dreamweaver CS4, Android browser, Palm webOS browser, Symbian S60 browser, OWB, Steam

Tabella 3.1.: Elenco dei motori di rendering utilizzati dai browser (tratto da [1])

nei principali browser, mentre la tabella 3.2 [1] compara le caratteristiche di JavaScript supportate dai vari interpreti (*JavaScript engine*) raggruppati in base ai motori di rendering che li integrano. Si noti come la specifica EcmaScript 3 sia implementata ormai da tutti i motori di rendering (e quindi da tutti i browser), mentre la specifica 5 è implementata solo da Gecko 2.0 (Firefox 4) e da Trident 5.0 (Explorer 9), entrambi ancora in fase di sviluppo. Le righe della tabella con la dicitura "JavaScript <*version*> extensions" si riferiscono invece alle caratteristiche introdotte dalle varie versioni del linguaggio JavaScript, che è di fatto l'implementazione sviluppata da Mozilla della specifica EcmaScript. Tali estensioni sono state adottate anche dai motori di rendering WebKit e Presto, che però rimangano sempre un passo indietro rispetto a Gecko. Trident rimane al di fuori di questa competizione perché implementa il proprio dialetto EcmaScript, noto con il nome di JScript.

Ad ogni modo, per rendere trasparenti le differenze di implementazione del linguaggio nei vari browser, nel corso degli ultimi anni si sono consolidati numerosi framework, che forniscono interfacce per l'utilizzo di widget grafici, tecniche di interazione avanzate (come il drag&drop) e web remoting (XmlHttpRequest), consentendo al programmatore di poter sviluppare ad alto livello applicazioni web portabili e potenti.

	Trident	**Gecko**	**WebKit**	**Presto**
Name of ECMAScript Engine	*JScript/ Chakra*	*Spidermonkey/ TraceMonkey*	*JavaScriptCore/ SquirrelFish*	*Extreme Linear B/ Futhark/ Carakan*
ECMAScript Edition 3	Yes	0.6	Yes	1.0
ECMAScript Edition 5	5.0	2.0	No	No
JavaScript 1.5 extensions	No	0.6	Yes	1.0
JavaScript 1.6 extensions (excluding E4X)	No	1.8	Partial	Partial
JavaScript 1.7 extensions	No	1.8.1	No	Partial
JavaScript 1.8 extensions	No	1.9	Partial	No
JavaScript 1.8.1 extensions	No	1.9.1	No	No
JScript .NET extensions	No	No	No	No
ActionScript extensions	No	No	No	No

Tabella 3.2.: Comparazione degli interpreti JavaScript (tratto da [1])

3.2. JavaScript nei dispositivi mobili

Con il repentino sviluppo tecnologico degli ultimi anni, si è assistito ad una miniaturizzazione delle componenti hardware che ha permesso la creazione di palmari e cellulari sempre più piccoli e potenti, tanto da diventare paragonabili quanto a potenza di calcolo e funzionalità ai sistemi desktop.

Il browser, che prima esisteva nei dispositivi mobili in forma del tutto inadeguata ad un utilizzo efficiente, è stato più volte reinventato per consentire la più vasta possibile fruizione di risorse del web. Come si vede dalla tabella 3.1 di comparazione dei motori di rendering, lo stesso motore è utilizzato contemporaneamente da vari browser, talvolta anche mobili. Ad esempio il motore WebKit, sviluppato da Apple, è utilizzato sia dai browser "desktop" Safari e Chrome, che dai browser mobili per iOS (iPhone e iPad), Android, Palm e Symbian.

È stato quindi necessario dotare i browser mobili anche di un interprete JavaScript per consentire la visualizzazione di siti web dinamici anche nei dispositivi mobili.

3.3. Concetti di JavaScript

3.3.1. Letterali oggetto

In JavaScript è possibile istanziare molto facilmente un nuovo oggetto, semplicemente dichiarando tra parantesi graffe una lista di chiavi e valori. Per accedere ad una proprietà di un oggetto si può usare sia l'operatore "`.`" che l'operatore "`[]`".

```
var myObj = {
  key1: value1,
  key2: value2
  ...
};
myObj['key1'] = myObj.key2;
```

In molti altri linguaggi esiste la possibilità di creare oggetti in modo analogo, di solito tramite quella che è chiamata Map, Dizionario o Array associativo, ai quali possono essere aggiunte o rimosse a tempo di esecuzione coppie chiave-valore. Ma in JavaScript tale semplice struttura offre anche altre potenzialità:

metodi poiché le funzioni sono oggetti di prima classe, ad una proprietà di un oggetto può essere associata anche una funzione, rendendolo così non più una semplice Map ma un oggetto personalizzabile;

prototipo un oggetto così creato può essere utilizzato come prototipo (vedi sezione 3.3.2), ovvero come stampo per istanziare altri oggetti suoi cloni.

I letterali oggetto sono così pratici che hanno ispirato la creazione di JSON (vedi sezione 4).

3.3.2. Costruttori e Prototipi

Seppure in JavaScript non esistano le classi, è comunque possibile simularle definendo un oggetto e replicandolo con il costrutto `new`.

```
function MyObject (formalParameter) {
  this.instanceVar = formalParameter;
}
//...
var myObj = new MyObject(actualParamater);
```

La funzione `MyObject` rappresenta quindi il costruttore della "classe" omonima, da invocare mediante il costrutto `new`.

Per arricchire di metodi e variabili membro tutte le istanze di una stessa classe si può usare il prototipo dell'oggetto, reperibile mediante il metodo `getPrototypeOf(object)`, che è un abbreviazione di `object.constructor.prototype`.

```
myPrototype.increase = function () {
  this.instanceVar += 1;
}
//...
myObj.increase();
```

La modifica del prototipo si propaga a tutte le istanze della classe ed è valida sia per gli oggetti creati prima della definizione del metodo che per quelli che ancora devono essere creati.

Grazie all'uso dei prototipi è quindi possibile riprodurre molte delle caratteristiche del paradigma object oriented come il polimorfismo, l'ereditarietà (singola o multipla), i membri privati di una classe (tramite selettori e modificatori locali al costruttore), la tipizzazione forte (mediante il metodo `watch`, che monitora le modifiche ad una proprietà) e altri ancora.

3.3.2.1. Un approccio alternativo all'istanziazione di oggetti

Invocare direttamente (senza `new`) la funzione `MyObject` è lecito ma semanticamente inutile. Questo perché il costrutto `new` è stato introdotto appositamente per venire incontro alle richieste dei programmatori abituati ai linguaggi orientati agli oggetti, ma in realtà è solo fonte di confusione. Douglas Crockford propone in [11] un approccio per la creazione di oggetti basato sulla definizione di una funzione `Object.create(object)` che restituisca il clone dell'oggetto passato come parametro.

```
Object.create = function (o) {
  var F = function () {};
  F.prototype = o;
  return new F();
};
//...
var myObjetPrototype = {
  instanceVar: undefined,
  increase: function () {
    this.instanceVar += 1;
  }
}
//...
```

```
var myObject = Object.create(myObjectPrototype);
```

Listato 3.1: Creazione di oggetti da un prototipo

Si noti che `myObjetPrototype` è stato scritto con l'iniziale minuscola per evidenziare il fatto che non è un costruttore invocabile con il costrutto `new`.

Questo approccio, che è meno caotico del precedente, ha però una controindicazione: ogni istanza di una stessa classe avrà un costruttore diverso, poiché ne viene creato uno ad ogni invocazione di `Object.create`, quindi il loro prototipo non sarà più raggiungibile tramite `object.construcor.prototype`, ma solo mediante `getPrototypeOf(object)`. Ovviamente è anche possibile utilizzare direttamente il riferimento al prototipo contenuto nella variabile `myObjetPrototype` dell'esempio.

3.3.2.2. Catene di prototipi

Anche i prototipi sono dotati della proprietà `prototype`, che può quindi essere vista come un riferimento alla classe da cui un'altra classe *eredita* tutti i membri e metodi.

Quando si accede in lettura alla proprietà di un oggetto, questa viene prima cercata fra le proprietà dell'oggetto, poi tra le proprietà del suo prototipo, poi nel prototipo di quest'ultimo e così via, finché non si arriva alla base della catena, che sarebbe `Object.prototype`, il cui prototipo è indefinito.

Quando invece si accede in scrittura ad una proprietà, questa viene immediatamente assegnata all'oggetto. Se esisteva già una proprietà con lo stesso nome nella catena di prototipi, questa viene mantenuta ma non è più raggiungibile dall'oggetto che l'ha *sovrascritta*[2].

La catena di prototipi è contenuta nella proprietà interna (non accessibile) `[[prototype]]` della classe interna `Object`.

3.3.3. Funzioni

Le funzioni possono essere create tramite *dichiarazione*:

```
function f1 () {
  // si invoca con f1()
  // viene interpretata contestualmente al contesto che la contiene
}
```

assegnamento:

[2] Ma può ancora essere raggiunta dal suo prototipo!

```
var f2 = function () {
  // si invoca con f2()

  // viene interpretata solo al momento in cui (e se)
  // viene eseguita l'operazione di assegnamento
}
```

o *istanziazione* mediante costruttore `Function`:

```
var f3 = new Function(params, functionBody);
// si invoca con f3()
```

Le funzioni istanziate si comportano in modo leggermente diverso dalle altre due. Ad esempio il loro corpo viene rivalutato ad ogni loro esecuzione e sono pertanto più le lente delle loro controparti dichiarate od assegnate. Inoltre la loro catena di Scope (vedi sezione 3.3.3.3) è sempre composta dal solo oggetto globale. Il loro uso è pertanto di solito sconsigliato.

Come abbiamo già detto le funzioni sono oggetti di prima classe. Quindi possono essere passate come valori ad altre funzioni o associate a proprietà di oggetti. Inoltre possono avere esse stesse variabili membro e metodi.

3.3.3.1. arguments

A partire da JavaScript 1.4 ogni funzione ha, oltre alle sue variabili locali, la variabile `arguments`, un oggetto array-like[3] contenente i parametri passati alla funzione. Nelle versioni precedenti `arguments` era una proprietà degli oggetti di tipo `Function`.

`arguments`ha tre proprietà principali:

callee riferimento alla funzione attualmente in esecuzione;

caller (*deprecato*) riferimento alla funzione che ha invocato la funzione attualmente in esecuzione;

length numero di parametri attuali.

3.3.3.2. this

La parola chiave `this` si riferisce all'oggetto di contesto (detto anche oggetto corrente). Ad esempio, in una chiamata al metodo `obj.method()`, nel corpo del metodo `this` è un riferimento a `obj`.

Ci sono 4 modi con cui `this` può essere passato:

[3] Si usa come un array, ad esempio accedendo ad i suoi valori mediante `[]`, ed ha una proprietà `length`, ma non ha gli altri metodi di array (`split`, `push`, `pop`...) e non è neanche modificabile.

Modo	Invocato da	This
Implicitamente con una chiamata a metodo	`object.method(...)`	`object`
Esplicitamente attraverso `Function.prototype.apply`	`function.call(object, ...)`	`object`
Esplicitamente attraverso `Function.prototype.call`	`function.apply(object, [...])`	`object`
Implicitamente con `new`	`new constructor (...)`	Nuovo contesto anonimo

Tabella 3.3.: Significato di `this` nelle funzioni

Si deve usare con cautela la parola chiave `this` in quanto essa può non riferire sempre all'oggetto che ci si aspetta.

Ad esempio:

```
function Car(brand) {
  this.brand = brand;
}
Car.prototype.getBrand = function() {
  return this.brand;
}
var foo = new Car("toyota");
alert(foo.getBrand()); // "toyota"
var brand = "not a car";
var bar = foo.getBrand;
alert (bar()); // "not a car"
```

Listato 3.2: Esempio d'uso di `this`

Nel secondo caso, l'oggetto che invoca `bar()` è l'oggetto globale `window` che ha una proprietà che si chiama `brand` e che viene "catturata" dal metodo `getBrand()`.

3.3.3.3. Contesti di esecuzione e istanziazione delle variabili

Tutto il codice JavaScript viene eseguito in un *Contesto di Esecuzione*, che può essere globale, come spesso accade nel codice eseguito inline in file JavaScript o HTML, o locale a una funzione. Ogni volta che si invoca una funzione viene infatti creato un nuovo Contesto di Esecuzione in cui la funzione "entra". Al Contesto di Esecuzione sono associati uno *Scope* e varie proprietà, come ad esempio `arguments`. Quando la funzione conclude la sua esecuzione si torna al Contesto di Esecuzione precedente. In pratica l'esecuzione di codice JavaScript forma una *catena di Contesti di Esecuzione*. Il

Contesto di Esecuzione è definito nella specifica EcmaScript come concetto astratto e viene implementato da un oggetto concreto di tipo `Activation`.

Lo Scope consiste invece in una catena di oggetti di tipo Activation che viene mantenuta nella proprietà interna `[[scope]]`. In testa alla catena c'è sempre l'oggetto `Activation` associato alla funzione.

3.3.3.4. Istanziazione di variabili

Dopo la creazione del Contesto di Esecuzione inizia il processo di *istanziazione delle variabili* locali, mantenute in un oggetto `Variable`. Seppure per specifica `Variable` sia un oggetto distinto da `Activation`, di fatto è lo stesso `Activation` ad assolvere ai compiti di `Variable`. Viene creata una *proprietà nominata* (named property) dell'oggetto `Variable` per ogni parametro formale della funzione, a cui vengono assegnati valori in base ai parametri attuali, se presenti, altrimenti restano `undefined`. Per ogni funzione interna viene creata una proprietà nominata di `Variable` che ha per nome il nome della funzione e per valore l'oggetto-funzione.

L'ultimo stadio dell'istanziazione delle variabili consiste nella creazione di una proprietà nominata per ogni variabile locale. Inizialmente a queste proprietà viene assegnato il valore `undefined`. Dal fatto che l'oggetto `Activation`, con la sua proprietà `arguments`, e l'oggetto `Variable`, con le proprietà nominate corrispondenti alle variabili locali della funzione, siano lo stesso oggetto, ne deriva che l'identificatore `arguments` possa essere trattato come se fosse una variabile locale della funzione.

Infine viene assegnato un valore alla parola chiave `this`. Se il valore è un oggetto, attraverso `this` si potrà accedere alle proprietà dell'oggetto, se è `null`, `this` sarà l'oggetto globale.

3.3.3.5. Contesto di Esecuzione Globale

Il *Contesto di Esecuzione Globale* è leggermente diverso in quanto non ha associato un oggetto `Activation` e non necessita di uno Scope in quanto la sua Scope chain è composta dal solo dall'oggetto globale. Le sue funzioni sono le funzioni *top level* e il suo oggetto `Variable` è l'oggetto globale: questo è il motivo per cui le funzioni dichiarate globalmente sono proprietà dell'oggetto globale, così come anche le variabili globali.

3.3.3.6. Scope Chain

La *Scope Chain* di un Contesto di Esecuzione per una chiamata a funzione è costruita aggiungendo l'oggetto `Activation`/`Variable` del Contesto di Esecuzione in testa alla

catena mantenuta nella proprietà `[[scope]]` dell'oggetto-funzione, perciò è importante capire come la proprietà interna `[[scope]]` sia definita.

Le funzioni create con il costruttore `Function` hanno nella proprietà `[[scope]]` una scope chain che contiene il solo oggetto globale, mentre le funzioni create con dichiarazione o assegnamento (vedi sezione 3.3.3) hanno nella proprietà `[[scope]]` la scope chain dell'ambiente in cui sono state create.

Nel caso più semplice, una funzione dichiarata od assegnata nel Contesto Globale di Esecuzione, avrà una proprietà `[[scope]]` corrispondente ad una catena che contiene il solo oggetto globale. Se invece dichiariamo due funzioni contenute l'una nell'altra ed eseguiamo la funzione esterna (valutando così il suo codice) la funzione esterna avrà uno Scope Chain composto dal solo oggetto globale, mentre quella interna avrò uno scope chain composto dall'oggetto Activation della funzione esterna e dall'oggetto globale, e così via.

Lo Scope Chain può essere temporaneamente modificato dal costrutto `with(expr)-stmts` che aggiunge l'oggetto valutato in `expr` allo Scope Chain durante l'esecuzione del blocco `stmts`, e lo rimuove alla fine della sua esecuzione.

3.3.3.7. Risoluzione di identificatori

La risoluzione degli identificatori avviene cercando l'identificatore tra le proprietà dell'oggetto in testa alla Scope Chain corrente, poi viene cercato tra gli oggetti della sua Prototype Chain, quindi viene cercato nel secondo oggetto della Scope Chain e così via, fino a risalire al Contesto di Esecuzione Globale, che è sempre in testa alla Scope Chain.

Quindi, quando ci si trova all'interno di una funzione, l'identificatore viene prima ricercato tra le variabili e funzioni locali alla funzioni, che sono proprietà del Contesto di Esecuzione in testa alla Scope Chain.

3.3.4. Chiusure lessicali

Le funzioni JavaScript sono anche dette *chiusure* in quanto conservano lo "stato" del contesto in cui sono state dichiarate. Una funzione può infatti accedere alle variabili locali dichiarate al suo interno e alle variabili locali contenute nello stesso scope in cui è dichiarata la funzione e nella catena di Scope che contengono la funzione, persino se queste variabili non sono più accessibili. Diversamente da quanto avviene nei linguaggi C-like, il valore di queste variabili viene mantenuto (per riferimento) "all'interno della funzione". Un'altra istanza dello stesso oggetto-funzione potrebbe avere valori diversi (ad esempio parametri diversi).

```
1  /**
2   * Ritorna una funzione che ha un parametro implicito
3   * e due impliciti
4  **/
5  function exampleClosureForm(arg1, arg2){
6    var localVar = 8;
7    function exampleReturned(innerArg){
8      return ((arg1 + arg2)/(innerArg + localVar));
9    }
10   return exampleReturned;
11 }
12 var firstGlobalVar = exampleClosureForm(2, 4);
13 var secondGlobalVar = exampleClosureForm(12, 3);
14 alert(firstGlobalVar(5)); // 0.75: (2+4)/(5+8)
15 alert(secondGlobalVar(5)); // 1.15: (12+3)/(5+8)
```

Listato 3.3: Esempio di chiusura lessicale

Più precisamente, una chiusura è formata ritornando un oggetto-funzione che è stato creato all'interno di un Contesto di Esecuzione di una chiamata a funzione e assegnando un riferimento a tale funzione interna a una proprietà di un altro oggetto. Oppure assegnando direttamente un riferimento a tale oggetto-funzione a, per esempio, una funzione globale o una proprietà di un oggetto accessibile globalmente o un oggetto passato per riferimento come argomento della chiamata alla funzione esterna.

Nell'esempio del listato 3.3 `firstGlobalVar` e `secondGlobalVar` sono chiusure lessicali in quanto hanno accesso a variabili (`localVar`, `arg1` e `arg2`) che non sono più accessibili, perché dichiarate in un contesto che è stato chiuso. I valori di tali variabili non possono essere rimossi dal Garbage Collector perché figurano come proprietà dell'oggetto `Activation` legato al Contesto di Esecuzione che sta in testa alla Scope Chain della funzione ritornata da `exampleClosureForm`.

Gli ambienti di esecuzione dei due oggetti-funzione contenuti nelle due variabili globali e ritornati da `exampleClosureForm` sono quindi distinti e infatti, quando vengono invocati, ritornano valori diversi anche se viene utilizzato lo stesso parametro.

3.3.4.1. Esempio di chiusura lessicale

Passaggio di parametri all'handler di setTimeout

Un esempio classico di chiusura lessicale è quello in cui si crea una funzione con un parametro preimpostato, in modo da poterla invocare in un secondo momento senza doverle passare esplicitamente il parametro. Questo meccanismo è necessario nel caso si voglia ad esempio impostare come handler di `setTimeout` una funzione con parametro.

```
function createHandler (param) {
  var myHandler = function () {
    console.log(param*2);
  }
  return myHandler;
}
var handler = createHandler (param);
var timerId = setTimeout (handler, time);
```

Listato 3.4: Utilizzo di handler con parametri in `setTimeout`

`setTimeout` può invocare una funzione con un certo ritardo, secondo la seguente sintassi:

```
var timerId = setTimeout (handler, time);
```

che non consente quindi di passare parametri alla funzione `handler`.

Per aggirare la limitazione si può dichiarare una funzione esterna a cui passare il parametro, la quale conterrà una funzione interna che usa il parametro locale alla funzione esterna. La funzione esterna fornisce come valore di ritorno un riferimento alla funzione interna, producendo così una chiusura lessicale. Si veda ad esempio il codice del listato 3.4.

4. JSON

JSON nasce da un'idea di Douglas Crockford come concorrente di XML, ovvero come linguaggio di interscambio di dati, ma con l'obiettivo di essere più veloce e leggibile. JSON è l'acronimo di JavaScript Object Notation, in quanto la sua sintassi è un sottoinsieme di quella di JavaScript. Essa prevede un set di tipi di dato sufficientemente espressivo e comune a tutti i linguaggi di programmazione.

Questo è un esempio di file JSON valido:

```
{
  books : [
    {
      author : "D. Crockford",
      title : "JavaScript: The Good Parts",
      publisher : "O'Reilly",
      year : 2008,
      lendable: true
    },
    {
      author : "P. Wilton and J. McPeak",
      title : "Beginning JavaScript",
      publisher : "Wiley Publishing",
      year : 2007,
      lendable : false
    }
  ],
  magazines: [
    ...
  ]
}
```

Listato 4.1: Esempio di file JSON

Come si vede è possibile definire dati primitivi come undefined, null, Stringhe, Numeri e e dati strutturati sotto forma di Array e Mappe.

Non è prevista la definizione di funzioni perché dipendenti da un particolare linguaggio, mentre un file JSON deve poter essere fruito da qualsiasi linguaggio.

Per poter utilizzare JSON ci vuole un *serializzatore* per convertire un valore in stringa JSON, e un *parser* per effettuare il processo inverso. In rete esistono moltissimi parser e serializzatori per molti linguaggi di programmazione.[1]

Poiché un file JSON è anche un file JavaScript, non è necessario un parser per caricare da un file JSON un oggetto JavaScript, ma è sufficiente utilizzare la funzione `eval`:

```
var myJSObject = eval('('+myJsonString+')');
```

Usare la funzione `eval` al posto di un parser è comunque sconsigliato in quanto non fa alcun test per verificare che il codice sia JSON-strict, ovvero che non contenga codice potenzialmente malevolo.

4.1. JSON nativo

JSON è stato inserito nella specifica di ECMAScript 5 ed è pertanto supportato nativamente dai browser più recenti (Firefox 3.5, Explorer 8, Google Chrome e Apple Safari 4). Il supporto di Opera era previsto per la versione 10.5 ma nella versione 10.10 non è ancora presente.

È quindi possibile evitare l'uso della funzione `eval` appoggiandoci al parser e al serializzatore nativi del browser che dovrebbero essere più veloci rispetto a librerie di terzi.

```
var myJsonString = JSON.stringify(myObject);
var myJavascriptObject = JSON.parse(myJsonString);
```

[1] http://www.json.org/

Parte II.

Migrazione

Ora che abbiamo fatto un po' di chiarezza su alcuni concetti chiave di JavaScript possiamo procedere con l'obiettivo principale: migrare lo stato JavaScript di un'applicazione web da un client all'altro in modo del tutto trasparente, senza cioè richiedere l'intervento dell'utente.

Sono stati considerati vari approcci per raggiungere tale scopo, tutti con il medesimo schema di base, che consiste nel salvare lo stato del client sorgente in modo da inviarlo ad un server che a sua volta lo inoltri al client destinatario, il quale si dovrà occupare di ripristinare lo stato dell'applicazione.

Ciò che varia da un approccio all'altro è il modo in cui si salva (e ripristina) lo stato (trattato nel capitolo 5) e il formato di salvataggio (trattato nel capitolo 6).

Nel capitolo 7 verrà fatta una breve panoramica sui problemi affrontati e su come sono stati risolti. Dopodiché il capitolo 8 analizzerà in dettaglio i problemi, mentre nel capitolo 9 verranno proposte soluzioni per ciascun problema.

Infine nei capitoli 10 e 11 verranno mostrati due casi d'uso della piattaforma di migrazione.

5. Modo di salvataggio dello stato

Salvataggio della stato dell'interprete JavaScript

Il codice JavaScript eseguito all'interno di una pagina web accede in lettura e scrittura a delle zone di memoria riservate all'interprete JavaScript (detto anche JavaScript engine). Qualsiasi informazione relativa all'esecuzione del programma (variabili locali e globali, scope, stack delle chiamate, etc...) viene salvata in questa zona di memoria, che sarà probabilmente localizzata in uno o più file. Pertanto se potessimo prelevare il contenuto di questa memoria da un dispositivo e caricarlo in un altro dispositivo, avremmo di fatto implementato una migrazione.

Nel caso i due dispositivi utilizzino lo stesso interprete JavaScript non dovrebbe essere difficile salvare e ripristinare lo stato con questo approccio ma, nel caso gli interpreti siano diversi, potrebbe essere necessario un lavoro di conversione dello stato in modo da appianare le differenze tra i due interpreti.

In generale è difficile rendere questo approccio multi-piattaforma, dato che ogni browser avrà potenzialmente un diverso interprete JavaScript e una diversa strutturazione della memoria. Si deve tener conto poi che le differenze si acuiscono passando da desktop a mobile.

Monitoraggio dei cambiamenti

Nell'ambito delle ricerche che hanno a che vedere con JavaScript, la *strumentazione* dell'engine JavaScript è una pratica comune per effettuare benchmark, analizzare comportamenti a tempo di esecuzione [21] o semplicemente per creare dei log [23].

Tale tecnica consiste nel modificare direttamente il codice dell'interprete JavaScript, ovvero il codice relativo al runtime JavaScript[1], in modo da creare comportamenti ad hoc o da inserire chiamate a funzioni per implementare funzionalità aggiuntive.

Si potrebbe applicare al nostro problema, implementando un sistema di monitoraggio che registri ogni assegnamento di variabile e chiamata di funzione dell'applicazione web. Per effettuare il ripristino sul dispositivo destinatario della migrazione si dovrebbe

[1] Ad esempio in Internet Explorer 8, il codice relativo al runtime JavaScript risiede nella libreria dinamica jscript.dll, pertanto in [21] è stato sufficiente ricompilare la libreria dovo aver modificato il codice sorgente relativo al runtime.

quindi partire da una condizione iniziale dello stato e ripetere sistematicamente tutte le operazioni effettuate nel dispositivo sorgente.

Appare subito chiaro come tale meccanismo non possa che scalare per applicazioni di modesta dimensione, che non hanno niente a che vedere con le applicazioni comunemente usate nel web che invece richiedono l'esecuzione continua di codice innescando quindi centinaia di operazioni al secondo.

Plugin per browser

In "Browser State Repository Service" [24] è stato affrontato un problema molto simile al nostro[2], appoggiandosi però all'utilizzo di un *plugin per browser*. In pratica l'utente deve installare un plug-in per ogni dispositivo che voglia utilizzare nella migrazione. Il plug-in consente all'utente di fare una "fotografia" della sua sessione di lavoro e di inviarla ad un server, che la conserverà affinché l'utente possa utilizzarla per riprendere il suo lavoro su un altro dispositivo.

Questo approccio limita l'uso della migrazione ai soli browser per cui è stato sviluppato il plug-in e non garantisce la possibilità di estensione a tutti i browser, in quanto ciascuno di essi fornisce le proprie API per i plug-in, che possono variare anche notevolmente da un browser all'altro.

Librerie JavaScript

L'approccio che abbiamo adottato è invece basato sull'utilizzo di librerie JavaScript. Ovvero tutto il codice che gestisce la migrazione sul client è costituito da codice JavaScript, cablato direttamente nella pagina HTML inviata dal Migration Proxy al browser del client. In questo modo, a patto di utilizzare un sottoinsieme di JavaScript comune ai browser più utilizzati, si può ottenere una piattaforma di migrazione che sia indipendente dal browser. Inoltre il Migration Client, ovvero l'applicazione utilizzata dall'utente per richiedere la migrazione, è un'applicazione web indipendente dal browser e che non richiede alcuna installazione di software da parte dell'utente.

[2] Nell'articolo si parla di salvataggio dello stato degli script (JavaScript o VBScript) ma non viene spiegato come viene affrontato, se non attraverso un generico salvataggio delle variabili globali.

Tra l'altro c'è una grossolana imprecisione che dice che "non c'è bisogno di salvare le funzioni degli script, in quanto le funzioni non cambiano e sono ricaricate nel browser quando lo snapshot [lo stato salvato] è ripristinato", ovvero che le funzioni sono ripristinate al caricamento della pagina. In realtà alcune variabili contenenti riferimenti a funzioni potrebbero essere inizializzate durante l'esecuzione e non a tempo di caricamento. Inoltre non tutte le funzioni sono "senza stato".

Si veda la sezione 8.6 per un'analisi più dettagliata dei problemi relativi alle funzioni in JavaScript.

5.1. Enumerazione delle variabili globali

Poiché JavaScript è un linguaggio basato su variabili globali, per poter ispezionare lo stato di un applicazione è sufficiente accedere ai valori di ciascuna variabile globale. Tali valori sono accessibili come proprietà dell'oggetto globale `window`, enumerabili quindi tramite un ciclo `for..in`:

```
for (propertyName in window) {
  var propertyValue = window[prop];
  saveValueToState(propertyName, propertyValue);
}
```

Fra le proprietà dell'oggetto `window` ve ne sono molte che non hanno niente a che fare con il codice JavaScript caricato nella pagina web, ma che fanno parte del cosiddetto BOM, *Browser Object Model*, un'interfaccia fornita dal browser per mettere a disposizione funzioni e valori di varia utilità, come ad esempio:

location l'indirizzo del file attualmente caricato nel browser;

document il riferimento alla radice del DOM, l'albero rappresentante il contenuto (X)HTML del documento visualizzato;

history contenente le informazioni sulla storia della navigazione, utilizzata ad esempio per i tasti "indietro" e "avanti" del browser;

e tanti altri...

Tali proprietà devono essere escluse dallo stato degli script JavaScript, in quanto essendo dipendenti dal browser e più in generale anche dal dispositivo, potrebbero causare inconsistenze quando si cerca di ripristinarle. Inoltre alcune di queste proprietà, come `document`, sono già trattate dalla piattaforma di migrazione.

5.2. Esclusione delle variabili del BOM dallo stato

Quando si carica nel browser una pagina web priva di codice JavaScript, l'oggetto globale `window` conterrà tutte le variabili del BOM che vogliamo escludere dallo stato JavaScript.

Un meccanismo semplice per escludere tali variabili è quello di creare una lista di esclusione da utilizzare come filtro per lo stato ad ogni migrazione. Ovviamente ogni browser ha il suo BOM ed è pertanto necessario creare una lista per ogni browser.

Per creare una lista di esclusione si può aprire una nuova finestra del browser tramite la funzione `window.open` e inserire tutte le proprietà del riferimento alla nuova finestra in una lista di nomi.

```
var newWindow = window.open();
var exclusionList = [];
for (propertyName in newWindow) {
  exclusionList.push(propertyName);
}
```

Al momento della migrazione si filtreranno le variabili da inserire nello stato escludendo quelle presenti nella lista relativa al browser in uso. Se il browser non è fra quelli supportati allora si filtreranno le proprietà presenti in tutte le liste.

Il riconoscimento del browser non è sempre banale, in quanto non c'è uno standard di identificazione utilizzato da tutti i browser e quindi, per avere una valutazione accurata, si devono invece fare varie valutazioni incrociate. Esistono però varie librerie a cui poter delegare questo compito. In questa tesi ad esempio è stata usata la libreria "JavaScript Browser Sniffer" [7].

6. Formato di salvataggio dello stato

La scelta di un formato di salvataggio piuttosto che un altro è pressoché ininfluente per quanto riguarda la risoluzione del problema che abbiamo affrontato. Tale scelta è stata quindi guidata dalla velocità di elaborazione, dalla facilità di realizzazione e dalla presenza di librerie di terzi che potessero essere sfruttate per alleggerire il carico di lavoro.

Questi sono i formati valutati:

- XML
- JSON
- YAML
- JavaScript puro

6.1. XML vs JSON

JSON, di cui abbiamo parlato nell'introduzione (capitolo 4), è il più semplice tra i linguaggi presi in considerazione. È di facile lettura, veloce e leggero. Certamente più leggero di XML, che richiede dei parser in grado di navigare l'albero rappresentante il documento XML. Per contro XML, "*eXtensible Markup Language*", è più espressivo e versatile.

Nel mondo dello sviluppo web basato su librerie e frameworks, i due linguaggi sono talmente supportati da essere praticamente interscambiabili. Ma non è sempre sempre così, tanto che si sono venute a creare due scuole di pensiero, ciascuna delle quali cerca di far emergere come linguaggio universalmente adottato l'uno piuttosto che l'altro.

Come spesso accade in lotte di queste tipo si è venuta a formare una scuola di pensiero intermedia, secondo la quale si deve cercare di sfruttare il più possibile la velocità e semplicità di JSON quando questo sia sufficiente, ripiegando invece su XML nel caso ci siano esigenze di maggiore espressività e flessibilità.

6.2. YAML

YAML, la cui pronuncia fa rima con "camel", è un acronimo ricorsivo che sta per "YAML Ain't a Markup Language" (YAML non è un linguaggio di markup).

YAML è un linguaggio human-friendly, unicode-based, cross-language per la serializzazione di dati, pensato in particolare per le strutture dati dei moderni linguaggi di programmazione agili. Serve a molteplici scopi che spaziano dai file di configurazione ai messaggi su Internet, dalla object persistence alla revisione dati (data auditing).

YAML è in effetti un soprainsieme di JSON che sacrifica un po' della sua facilità di codifica e decodifica a favore di una maggiore leggibilità e flessibilità. Ogni documento JSON è infatti un documento valido YAML, il che rende facile la migrazione da JSON a YAML quando è richiesta una maggiore complessità.

Oltre alle differenze già citate, YAML ha alcuni vantaggi rispetto a JSON:

- gestisce correttamente oggetti di tipo `Date`;
- è in grado di definire tipi di dato ad hoc;
- è in grado di gestire riferimenti tra oggetti.

6.2.1. Implementazioni per JavaScript

Il sito ufficiale[1] ospita una specifica estremamente chiara e dettagliata [9], ma sfortunatamente non esistono librerie JavaScript che la implementino completamente. Anzi, l'unica libreria sviluppata finora per la serializzazione è YAML JavaScript. Mentre per il parsing esiste la libreria js-yaml.

Entrambe le librerie sono incomplete, quasi prive di commenti e sprovviste di documentazione. In particolare la libreria per il dump non implementa il referencing, né la gestione di oggetti di tipo `Date`.

Per questo motivo l'uso di YAML è stato scartato in questa tesi, ma potrebbe essere una scelta interessante per sviluppi futuri di questo lavoro.

6.3. JavaScript puro

Non è detto che si debba per forza usare un linguaggio di interscambio se si sa a priori che sia il fornitore che il consumatore del messaggio da scambiare sono in grado di interpretare il JavaScript.

[1] http://www.yaml.org/

Si potrebbe invece inviare direttamente codice JavaScript, che è in pratica lo stesso principio di JSON. Solo che JSON si limita a definire un sottoinsieme di tipi di dato di JavaScript che sia comune a tutti i linguaggi di programmazione, sottoinsieme che non è sufficiente ai nostri scopi.

Utilizzando JavaScript puro si potrebbero risolvere i problemi legati alle carenze di JSON, ma se ne dovrebbero risolvere altri che sono automaticamente risolti dai serializzatori JSON. In particolare si dovrebbe risolvere il problema dei riferimenti circolari, che possono causare una ricorsione infinita durante la serializzazione, ma bisognerebbe anche, più in generale, trovare il modo di gestire i riferimenti tra oggetti.

6.4. Formato scelto: JSON

Poiché XML e JSON sono linguaggi di interscambio molto comuni nel web, la scelta è stata ristretta ad uno dei due. YAML manca infatti di un'implementazione completa per JavaScript, mentre l'approccio con JavaScript puro avrebbe richiesto troppi sforzi per risolvere problemi che sono già implicitamente risolti dai serializzatori di XML e JSON.

JSON è sembrato subito il formato migliore per salvare lo stato JavaScript di un'applicazione, proprio in virtù della sua intrinseca affinità con JavaScript, e delle sue velocità e semplicità. Ovviamente è stato necessario estenderlo per ovviare alle sue limitazioni (vedi sezione 8.1). Inizialmente le estensioni da fare erano marginali perciò JSON sembrava la scelta più ovvia. A posteriori ci si è però resi conti che per conservare uno stato il più possibile integrale, era necessario salvare molte informazioni aggiuntive, per cui un linguaggio più flessibile come XML, dotato anche del linguaggio di descrizione XSD, sarebbe forse stata una scelta migliore. L'utilizzo di XML o di un altro formato più espressivo di JSON (come YAML) potrebbe quindi essere un valido sviluppo futuro del progetto.

7. Breve panoramica sui problemi affrontati

In questo capitolo presenteremo brevemente i problemi che abbiamo affrontato per l'esportazione dello stato JavaScript di un'applicazione web. Nel successivo analizzeremo in dettaglio ciascun problema, proponendo poi una soluzione per ciascuno di essi nel capitolo 9.

7.1. JSON

Di base viene inserito nello stato tutto ciò che è supportato da JSON:

tipi primitivi: Number, String, Boolean, null

array: `[value1, value2, ...]`

array associativi (o Map): `{key1: value1, key2: value2, ...}`

7.2. Riferimenti a oggetti

```
var x = "someValue";
var y = x;
```

Il valore `someValue` non viene serializzato due volte nello stato (come avverrebbe in JSON standard), ma al posto del valore di `y` viene inserito nello stato un riferimento alla variabile `x`, in modo che, oltre a risparmiare memoria, la variabile `y` dopo la migrazione continui a fare riferimento alla variabile `x`.

7.2.1. Riferimenti circolari

```
var bartSimpson = {
  name: "Bart",
  father: {
    name: "Homer",
```

```
    son: bartSimpson
  }
};
```

Listato 7.1: Esempio di array associativo con riferimenti circolari

Un oggetto di questo tipo non sarebbe serializzabile con JSON standard. Con la libreria dojox.json.ref, al posto del valore della proprietà `son` viene messo un riferimento alla radice dell'oggetto (*path referencing*).

7.3. Timer

```
var timerId = setTimeout (handler, milliseconds);
clearTimeout (timerId);
```

Le API di EcmaScript non prevedono funzioni che consentano di accedere allo stato di un Timer attivo, né di poter enumerare la lista dei Timer attivi. Per consentire il ripristino di un Timer attivo dopo la migrazione sono state sovrascritte le funzioni standard `setTimeout` e `clearTimeout`.

7.4. Date

```
var date = new Date (2010, 07, 22);
```

JSON standard non prevede la possibilità di serializzare oggetti di tipo `Date`. La libreria dojox.json.ref fornisce un'implementazione parziale di questa serializzazione mediante stringa in formato ISO-UTC. È stata quindi estesa manualmente affinché gli oggetti di tipo `Date` venissero serializzati correttamente.

7.5. Proprietà dinamiche degli oggetti

```
var array = [value1, value2];
array.dynProp = someValue;
```

In JavaScript tutti i tipi di dato non primitivi sono oggetti e, come tali, possono avere proprietà assegnate dinamicamente. JSON serializza tali proprietà solo se l'oggetto è un "oggetto puro", ovvero un array associativo. Si è fatto quindi in modo che le proprietà dinamiche venissero conservate nello stato anche per altri tipi di oggetto. Per adesso è stato fatto solo per gli array, ma si può estendere tale caratteristica ad altri tipi di oggetto.

7.6. Riferimenti a nodi del DOM

```
var element = getElementById ("myHtmlElement");
var image = new Image ("imageSource");
```

Nel caso un oggetto di tipo `HtmlElement` sia effettivamente un riferimento ad un nodo del DOM, si controlla se ha un `id` assegnato. Se ce l'ha, si salva nello stato il suo `id`, se non ce l'ha gliene assegniamo uno.

Se l'oggetto non è un riferimento del DOM (come la variabile `image` dell'esempio), viene salvato per valore accedendo a tutte le sue proprietà pubbliche. Attualmente vengono gestiti in questo modo solo oggetti di tipo `Image` e `Div`.

7.7. Funzioni

```
function foo (param) {
  return param * externValue;
}
```

JSON non serializza oggetti di tipo `functionv`, ma la libreria dojox.json.ref prevede questa possibilità. Senza opportune estensioni si rischia però che lo "stato" della funzione, in particolare se è una chiusura lessicale, venga perduto durante la migrazione (nell'esempio il valore `externValue` potrebbe non essere più accessibile se non attraverso la funzione).

Per risolvere il problema, il codice JavaScript viene modificato in modo che le chiusure lessicali non accedano più ad uno stato interno nascosto, ma ad uno stato globale, quindi accessibile durante la migrazione.

8. Problemi da risolvere

Nei capitoli precedenti abbiamo selezionato come approccio per salvare lo stato l'utilizzo di librerie JavaScript e come formato di salvataggio il linguaggio di interscambio JSON, da estendere opportunamente per ovviare ai limiti imposti dall'interoperabilità tra linguaggi di programmazione.

In questo capitolo esamineremo i problemi dovuti ai limiti di JSON e quelli che sono più in generale legati alla migrazione della sessione di lavoro di un utente che utilizzi un'applicazione web basata su JavaScript. Il metodo adottato per risolverli verrà invece trattato nel capitolo successivo.

8.1. Limiti di JSON

JSON non è stato pensato per effettuare il dumping dello stato di un'applicazione, ovvero per salvare sistematicamente tutti i valori utilizzati da un'applicazione, ma come linguaggio di interscambio di dati tra server e client, ovvero per il trasferimento di modeste quantità di dati o di cui comunque si conosca a priori la struttura.

Per quanto concerne i nostri scopi, JSON ha quindi alcune forti limitazioni:

1. JSON non gestisce i *riferimenti tra oggetti* o, per meglio dire, JSON serializza qualsiasi tipo di oggetto per valore e non per riferimento. Quindi due riferimenti che puntino allo stesso oggetto verranno serializzati in due oggetti distinti ma con valori identici. Nella maggior parte dei casi questa è la soluzione desiderata, ma nel nostro caso, per mantenere uno stato il più possibile fedele all'originale, è necessario serializzare il riferimento ad un oggetto anziché il valore;

2. Per lo stesso motivo di cui sopra, JSON non è in grado di serializzare oggetti con *riferimenti ciclici*, sollevando un eccezione con messaggio "too much recursion";

3. il serializzatore JSON esclude valori che non hanno una rappresentazione in JSON, come `undefined` e soprattutto `function`, in quanto non sono indipendenti dal linguaggio di programmazione. Per aggiungere dei metodi ad un oggetto si può

però utilizzare una funzione `reviver` passata come parametro alla funzione di parsing;

4. il serializzatore JSON esclude le proprietà contenute nel prototipo. (Vedi sezione 3.3.2 per un richiamo sui prototipi.) In pratica include solo le proprietà dell'oggetto per cui `obj.hasOwnProperty(prop)` sia vero;

8.2. Proprietà dinamiche degli oggetti

A oggetti di qualsiasi tipo (array associativi, array, stringhe, funzioni...) possono essere assegnate dinamicamente delle proprietà. Tali proprietà sono conservate nello stato JSON solo se l'oggetto che le contiene è un letterale oggetto (o un array associativo) mentre vengono completamente ignorate negli altri tipi di oggetto, in quanto non possono essere espresse senza contravvenire alla notazione standard di JSON.

Analizziamo ad esempio cosa succede per gli array.

8.2.1. Array e Map

Seppure gli Array e gli Array associativi (talvolta detti Map) si dichiarino con una notazione diversa, si usano in pratica allo stesso modo.

Un array si dichiara di solito con una delle due notazioni:

```
array = [element1, element2, ...];
array = new Array();
```

Mentre un array associativo si dichiara con una lista di chiavi-valori, eventualmente vuota:

```
associativeArray = { key1: value1, key2: value2, ... }
```

Di solito si accede ad un elemento di un array attraverso un identificatore numerico che rappresenta anche la sua posizione all'interno dell'array. Ma, come per un qualsiasi oggetto, è possibile assegnargli a runtime nuove proprietà rendendolo di fatto anche un array associativo.

```
myKey = "anyKey";
array.mykey = "anyValue";
array[myKey] = "anyValue"; // equivalente alla precedente
```

Questo può causare problemi in quanto, durante la serializzazione, JSON esclude tutte le proprietà con chiave non numerica assegnate a runtime dalla stringa risultante.

Esempio di serializzazione di array:

```
array = [a, b, c];
array.prop = "a value";
array[5] = 'f';
jsonArray = JSON.stringify (array);
// jsonArray: [a, b, c, , , f]
```

8.3. Prototipi e Classi

Come abbiamo detto nell'introduzione (sezione 3.3.2), in JavaScript non esistono le classi, ma possono comunque essere simulate attraverso l'uso dei prototipi.

Serializzare con JSON un oggetto istanziato con `new` lo "appiattirà" fino a renderlo un semplice letterale oggetto in quanto verranno esclusi:

- i metodi (ovvero le proprietà di tipo funzione);
- le proprietà del prototipo;
- il riferimento al costruttore (e quindi anche al prototipo), che diventerà genericamente `Object`.

```
1  // costruttore MyObject
2  function MyObject (param) {
3    this.instanceVar = param;
4    this.getVar = function () {
5      return this.instanceVar
6    };
7  }
8  // Aggiunta di una proprietà al prototipo
9  MyObject.prototype.protoProperty = "pp";
10
11 // creazione di un oggetto di tipo MyObject
12 var myObj = new MyObject(5);
13 console.log (myObj.constructor); // function MyObject(param)
14 console.log (myObj.constructor.prototype); // {protoProperty="pp"}
15
16 // serializzazione e deserializzazione
17 var json = JSON.stringify(myObj);
18 var newObj = JSON.parse (json);
19
20 // In newObj mancano sia getVar che protoProperty
21 console.log (newObj); // { instanceVar=5}
22 // Il costruttore e prototipo originali si sono persi
```

```
23 console.log (newObj.constructor); // function Object()
24 console.log (newObj.constructor.prototype); // {}
```

Listato 8.1: Problemi nella serializzazione di oggetti istanziati con `new` e dotati di prototipo

8.4. Riferimenti a nodi del DOM

Il DOM, *Document Object Model*, è uno standard W3C pensato per rappresentare e navigare documenti HTML e XML. All'interno dei browser sono i cosiddetti *layout engine* (Gecko, Webkit, Presto...) ad occuparsi di fare il parsing della pagina (X)HTML in un DOM.

Attraverso le API fornite dal browser è possibile effettuare molte operazioni sull'albero (ad esempio aggiungere o rimuovere nodi), ma l'operazione più comunemente usata, introdotta con il DOM di livello 2, è probabilmente la `getElementById` che consente di ottenere un riferimento al nodo del DOM contrassegnato da un identificatore univoco.

Non è necessario che un browser fornisca il supporto al DOM per poter visualizzare pagine web, ma è necessario per poter modificare la pagina a tempo di esecuzione, in quanto il DOM è l'interfaccia attraverso la quale JavaScript vede la pagina HTML che lo contiene e lo stato del browser.

Se l'oggetto che si vuole serializzare contiene riferimenti a nodi al DOM, questi verranno esclusi dalla serializzazione. Il problema trattato in questa sezione è in effetti l'unione di due problemi di cui abbiamo parlato in precedenza:

- la perdita di riferimenti ad oggetti;
- la perdita di informazioni su oggetti complessi.

8.5. eval

La funzione `eval` consente di valutare a tempo di esecuzione una stringa di codice arbitrario. Il codice eseguito all'interno di una clausola `eval` ha un proprio contesto di esecuzione e funziona pertanto in modo diverso dal codice tradizionale.

Il suo uso è sconsigliato dalle norme di buone programmazione, in quanto rende il codice meno leggibile e mantenibile, nonché più difficile da debuggare. Inoltre la funzione `eval` è maggiormente prona ai problemi di sicurezza, in quanto non esegue alcuna validazione sul codice da interpretare.

Fortunatamente, con il passare degli anni si sta consolidando un uso più strutturato e consapevole del JavaScript che esclude quindi l'utilizzo della funzione `eval`.

Per tutti questi motivi l'impatto della funzione `eval` non verrà valutato in questa tesi.

8.6. Funzioni e chiusure lessicali

Poiché le funzioni sono dipendenti dal linguaggio di programmazione, serializzando lo stato con un linguaggio di interscambio come JSON, le funzioni non verranno incluse nello stato.

Aggiungere le funzioni allo stato può sembrare semplice. Il corpo, comprensivo della signature, di una funzione può infatti essere facilmente ottenuto tramite la funzione `toString`.[1] Mentre per ripristinarlo è sufficiente utilizzare la funzione `eval`.

```
function foo (params) {...};
var serializedFoo = foo.toString();
var restoredFoo = eval(serializedFoo);
```

Poiché le funzioni sono interpretate solo al momento in cui si esegue il codice del contesto in cui sono dichiarate e poiché le funzioni possono fare uso di valori esterni al loro contesto (ad esempio del contesto globale o della catena di contesti in cui sono dichiarate - vedi sezione 3.3.4), si può verificare la condizione, per altro molto comune, in cui la funzione diventa una *chiusura lessicale* e il semplice salvataggio e ripristino del suo corpo porterebbe alla perdita dei riferimenti a valori contenuti in altri contesti.

8.6.1. Accedere ai valori racchiusi nella chiusura lessicale

È impossibile migrare una chiusura lessicale con l'approccio che abbiamo scelto per la migrazione (`for..in` sulle proprietà dell'oggetto globale). Per mantenere questo approccio bisognerebbe poter enumerare i valori racchiusi nella chiusura, accedendo allo scope chain della funzione che vogliamo migrare, la quale è contenuta nella proprietà interna `[[scope]]`.

[1] In realtà non tutte le funzioni consentono di accedere al proprio corpo mediante `toString`. Le API dei browser forniscono infatti alcune funzioni che non sono scritte in JavaScript ma che sono cablate nativamente all'interno dell'implementazione del browser.

Ad esempio, invocando in FireFox il metodo `toString` sulla funzione nativa `setTimeout` si ottiene il seguente risultato:

```
function setTimeout()  [native code]
```

Si deve pertanto evitare di salvare nello stato una funzione nativa per evitare che in fase di ripristino il riferimento alla funzione originale vada perduto.

Per specifica del linguaggio, le proprietà interne non sono esposte dal linguaggio stesso. Questa può sembrare una limitazione ma è in realtà voluta dai progettisti del linguaggio per mettere a disposizione le potenzialità delle chiusure lessicali. Grazie a tale meccanismo è infatti possibile, mantenere dei valori *privati* incapsulandoli all'interno di funzioni (chiusure lessicali) che avranno accesso esclusivo a tali valori, simulando di fatto i modificatori di visibilità dei linguaggi di alto livello.

Se ci fosse il modo di infrangere tale caratteristica, alcuni codici che vi fanno affidamento non potrebbero più garantire la sicurezza di un stato privato.

8.7. Chiamate AJAX pendenti

Le chiamate AJAX si appoggiano all'oggetto `XmlHttpRequest` che non fa parte dello standard W3C, ma che è ormai presente in tutti i browser moderni. Tramite questo oggetto è possibile aprire una connessione AJAX, sincrona o asincrona, con un server web (ad esempio una servlet), inviare una richiesta e ricevere una risposta.

```
var req = httpRequest.open (httpMethod, url, isAsynchronous);
req.send (<request>);
// ...
var xml = req.xmlResponse;
```

La richiesta asincrona consente di effettuare richieste al server mentre l'utente continua a navigare la pagina web. In un momento impredicibile arriverà la risposta del server, che sarà gestita da un handler appositamente definito.

Poiché anche la migrazione innescata dall'utente è un evento asincrono, al momento della sua attivazione potrebbero esserci ancora delle richieste AJAX pendenti. In tal caso le risposte arriverebbero dopo che la migrazione è già avvenuta, non raggiungendo quindi il dispositivo destinatario della migrazione.

Purtroppo le API del browser non forniscono delle funzioni per sapere, in un determinato momento, se ci sono delle richieste AJAX pendenti. Ma, anche nel caso fosse possibile ottenere la lista delle richieste pendenti, sarebbe comunque impossibile inoltrare le risposte arrivate dopo la migrazione al dispositivo destinatario (ad esempio sovrascrivendo il comportamento di `XmlHttpRequest`), in quanto quest'ultimo attende una specifica risposta da un determinato server.

Si potrebbe anche impedire che vengano effettuate altre richieste e ritardare la migrazione fino all'arrivo delle risposte alle richieste pendenti, prevedendo ovviamente un timeout per evitare un'attesa indefinita.

8.7.1. Ignorare le richieste pendenti

In alternativa si potrebbero anche ignorare del tutto le richieste pendenti, in quanto si presuppone che l'utente non attivi la migrazione in un momento cruciale. In particolare se la modifica dello stato apportata da AJAX è indipendente da eventuali modifiche apportate dall'utente, la mancata ricezione della risposta (e del conseguente aggiornamento di stato) comporterebbe al più, dal punto di vista dell'utente, un ritardo di aggiornamento dell'applicazione, analogo a quello che si potrebbe verificare in caso di un rallentamento della rete.

Questa è ovviamente un'assunzione molto forte: basti pensare ad un'applicazione che incrementa un contatore ogni volta che il server risponde ad una sua richiesta. Se durante la migrazione qualche risposta andasse perduta, tale contatore avrebbe un valore minore rispetto a quello che avrebbe se non avvenisse la migrazione, causando potenzialmente un malfunzionamento dell'applicazione.

In questa tesi si è scelto di affrontare problemi più stringenti, lasciando il problema delle richieste pendenti a sviluppi futuri.

8.8. Timers

Esiste un problema simile a quello delle richieste AJAX pendenti, che è quello dei timers in esecuzione.

I Timers non sono inclusi nella specifica EcmaScript. Essi sono oggetti non standard ma di uso molto comune, che sono quindi divenuti standard de facto ed inseriti nel cosiddetto DOM di livello 0. Fra le specifiche di W3C, compaiono in quella di Window Object 1.0 [12] e nella proposta di standard di HTML5 [14] sotto forma di funzioni dell'oggetto globale `window` e con i nomi `setTimeout`, `resetTimeout`, `setInterval` e `resetInterval`. Le prime due funzioni servono rispettivamente ad impostare e a fermare un timer singolo, mentre le seconde servono ad impostare un timer che si ripete ad intervalli regolari.[2]

[2] JavaScript è un linguaggio a thread singolo, pertanto l'esecuzione di handler asincroni avviene sempre in modo sequenziale (vedi figura 8.1).

Quando viene attivato un handler, ad esempio a seguito di un click del mouse, una callback AJAX o di un timer scaduto, questo viene messo in una coda. Non appena il codice attualmente in esecuzione termina, viene eseguito il primo handler in coda. Quindi, mediamente, gli handler dei timer non vengono eseguiti al loro scadere, ma dopo un ritardo che varia in base al carico di lavoro dell'applicazione JavaScript.

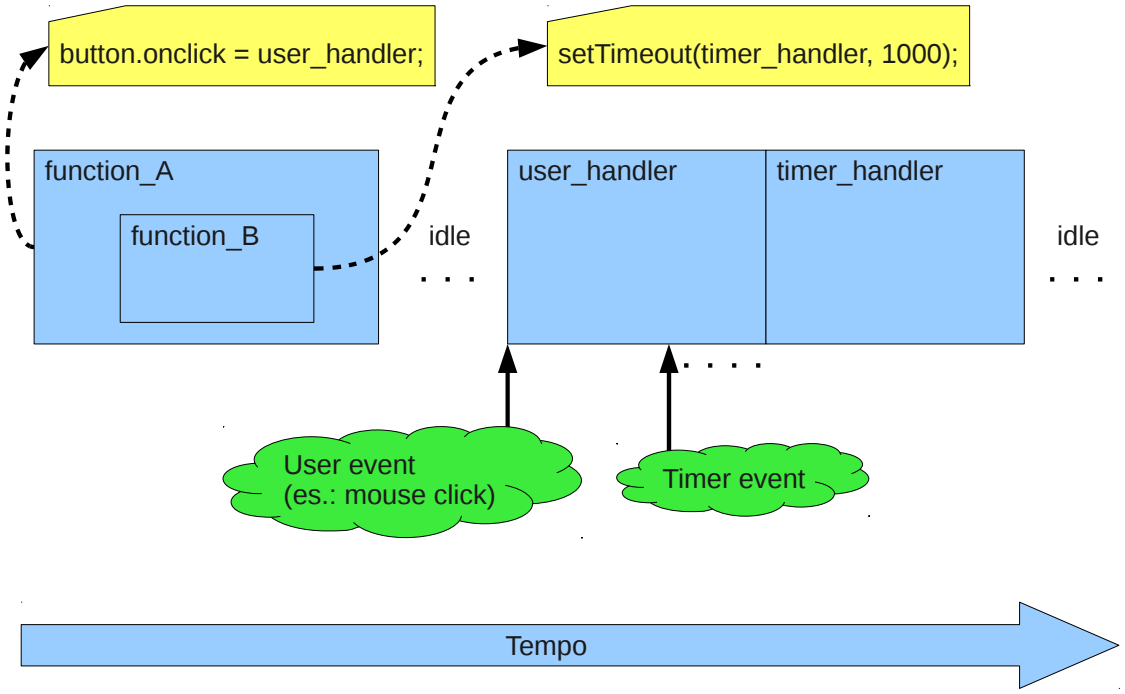

Figura 8.1.: Esecuzione sequenziale degli handler

Entrambe le funzioni "set" prendono come parametri un riferimento ad un handler ed un tempo in millisecondi[3] e ritornano un identificatore numerico del timer in esecuzione, che può essere passato come parametro alle funzioni "reset" per interrompere il timer.

```
var timerId = setTimeout (msec, handler);
// ...
resetTimeout (timerId);
```

La migrazione dello stato richiede la risoluzione di due problemi relativi ai timer:

1. la presenza di timer pendenti al momento della migrazione;
2. la presenza di variabili contenenti riferimenti a timer.

In realtà il riferimento numerico al timer viene incluso nello stato perché è un semplice intero (long integer), ma risulterà inconsistente una volta ripristinato nello stato del

[3] Secondo la recente specifica di HTML5 [14], le funzioni `setTimeout` e `setInterval` possono prendere in ingresso anche un terzo parametro opzionale, `args`, che consente di passare degli argomenti alla funzione di callback, evitando così di dover necessariamente ricorrere alla creazione di una chiusura lessicale come invece succedeva fino ad ora. Tale specifica è già supportata dalla maggior parte dei browser, ma non da Internet Explorer [8].

dispositivo destinatario della migrazione, a meno che non si trovi il modo di migrare anche i timers in esecuzione.

9. Soluzioni adottate

Come abbiamo visto nei capitoli precedenti, JSON di per sé non è molto potente e flessibile, ma esistono librerie che lo estendono consentendo di arginare alcune sue limitazioni, come la perdita dei riferimenti tra gli oggetti durante la serializzazione.

Dato che la maggior parte delle librerie che estendono JSON si concentrano su una o comunque poche funzionalità, sarà probabilmente necessario usarne più d'una.

Per risolvere il problema generale, ovvero la migrazione dello stato di un'applicazione web, lo si è scomposto in sotto-problemi, discussi nel capitolo 8 e risolti separatamente nel resto di questo capitolo.

Alla fine del capitolo (sezione 9.7) tutte le soluzioni adottate verranno finalmente messe insieme per risolvere il problema globale.

9.1. Riferimenti ad oggetti

Alcuni linguaggi dinamici forniscono la possibilità di ottenere un identificatore univoco per ogni istanza di un oggetto, come ad esempio la funzione `id()` di Python[1]. La specifica EcmaScript non prevede questa possibilità. Per poter conservare i riferimenti tra gli oggetti serializzati, si deve quindi trovare un modo di aggiungere al linguaggio l'identificazione di istanze.

L'estensione di JSON per il supporto dei riferimenti ad oggetti è un argomento di ricerca relativamente recente. Le prime tracce che ho trovato in rete risalgono al 2007, in due articoli di Kris Zyp, che ne aveva parlato nel suo blog JSON.com, non più online, ma reperibile tramite Internet Archive.

Nel primo articolo [27] l'autore elenca i vari approcci possibili al problema dell'object referencing, mostrando come verrebbe serializzato il seguente oggetto di esempio:

```
obj = {
  name:"foo",
  child: {
    "name" : "bar"
```

[1] http://docs.python.org/library/functions.html#id

```
  }
};
obj.child.parent = obj;
```

Possibili approcci per implementare l'object referencing in JSON

fixup schemes il cui utilizzo è stato proposto in JSON-RPC[2] 1.1 – si aggiunge all'oggetto serializzato una property `fixUp` che rappresenti, attraverso una particolare sintassi, i riferimenti ad altri oggetti.
L'oggetto di esempio diventerebbe il seguente messaggio JSON-RPC:

```
{
  "result" : {
    "name" : "foo",
    "child": {"name" : "bar"}
  },
  "fixups": [[
    ["child","parent"],
    []
  ]]
}
```

id referencing utilizzato in JSPON[3] – ad ogni oggetto che può essere referenziato si aggiunge un id numerico univoco. Questo tipo di referencing è utile nel caso vi siano *riferimenti inter-message*, ovvero tra oggetti JSON distinti. Il progetto è stato sviluppato dallo stesso Kris Zyp quando lavorava alla Xucia, compagnia che sembra scomparsa, così come il progetto JSPON sembra essere stato abbandonato.
L'oggetto di esempio serializzato con JSPON:

```
{
  "name":"foo",
  "id":"1",
  "child":
  {
    "name" : "bar",
    "parent": {"id":"1"}
  }
}
```

[2] JSON Remote Procedure Call: http://json-rpc.org/
[3] JavaScript Persistent Object Notation: http://www.jspon.org/

path referencing da utilizzare con la sintassi di JSONPath[4] – utilizza path relativi per riferire altri oggetti. È più semplice da implementare rispetto all'id referencing in quanto non richiede che ogni oggetto sia dotato di id, ma è inutilizzabile in caso di riferimenti inter-message.
L'oggetto di esempio serializzato con la sintassi di JSONPath ($ indica la root):

```
{
  "name":"foo",
  "child":
  {
    "name" : "bar",
    "parent": {"id":"$"}
  }
}
```

Nel secondo articolo [26] invece Zyp propone una vera e propria libreria basata sulle API della libreria JSON di Crockford [10], che cerca di prendere il meglio dai due approcci di *id* e *path referencing*. La libreria utilizza l'id referencing se trova degli id negli oggetti da serializzare, altrimenti utilizza il path referencing. Seppure il progetto sembrasse decaduto, Zyp ha continuato a coltivarlo presso l'azienda SitePen, per la quale sviluppa tecnologie web di ultima generazione.

9.1.1. JSON referencing in Dojo Toolkit

Recentemente anche il framework JavaScript *Dojo Toolkit*[5] ha esteso JSON per supportare il referencing [28]. In particolare il modulo sperimentale dojox.json.ref[6] fornisce il supporto a varie forme di riferimento come quello circolare, multiplo, inter-message e ritardato (lazy). Non sorprende scoprire che dietro a tale lavoro c'è la mano di Krys Zyp.

Purtroppo questo modulo non implementa le API della libreria di Crockford [10], quindi non è possibile ad esempio passare come parametro al serializzatore una funzione `replacer`. Nel modulo è però presente un'opzione `serializeFunctions` che consente di includere nel messaggio JSON anche le funzioni passate come stringhe. Il parser sarà poi in grado di ripristinarle senza operazioni aggiuntive.

[4] http://goessner.net/articles/JsonPath/
[5] http://www.dojotoolkit.org
[6] http://api.dojotoolkit.org/jsdoc/1.3/dojox.json.ref

9.2. Date

Gli oggetti di tipo `Date` vengono serializzati come oggetto vuoto - - in JSON.

Fra le caratteristiche della libreria dojox.json.ref figura anche la possibilità di salvare e ripristinare tale tipo d'oggetto, di uso estremamente comune negli script JavaScript.

Purtroppo però la sua implementazione è solo parziale. La libreria infatti serializza correttamente gli oggetti di tipo `Date` come stringhe ISO-formattate[7] [3] ma non è in grado di riconvertirle correttamente in oggetti di tipo `Date` al momento della deserializzazione.

Ad esempio un oggetto di tipo `Date` viene serializzato dalla libreria dojox.json.ref in una stringa come `"2010-03-27T16:27:16Z"`. Quando si deserializza il messaggio JSON rappresentante la stringa non si ottiene, come ci si aspetterebbe, un oggetto di tipo `Date`, ma di nuovo una stringa identica al messaggio JSON.

9.2.1. Discrepanze di orario

Si noti infine che migrando un oggetto di tipo `Date` da un macchina all'altra potrebbero comunque verificarsi delle asincronie dovute ai diversi orari interni delle due macchine. L'unico modo per limitare il problema, è che i dispositivi partecipanti alla migrazione sincronizzino l'orario presso lo stesso server.

9.2.2. Gestire le date con dojox.json.ref

In realtà la libreria dojox.json.ref è effettivamente in grado di deserializzare correttamente oggetti di tipo di `Date`, ma in modo un po' macchinoso. Bisogna infatti istruire il parser sull'esistenza di oggetti di tipo `Date` all'interno del messaggio JSON, mediante un secondo messaggio JSON, chiamato *Schema*, passato come parametro al parser.

In una pagina dei bugs di Dojo Toolkit ho trovato ad esempio il frammento di codice riportato nel listato 9.1.

Come si vede, oltre che dover costruire uno Schema, è necessario anche inserire gli oggetti serializzati all'interno di un *path*, nell'esempio "`/dog/`" e "`/cat/`", e assegnare loro un id, altrimenti il deserializzatore non sarà in grado di identificare le proprietà da trattare come date.

Non è necessario inoltre specificare nello schema la profondità a cui si trova la proprietà contenente l'oggetto di tipo `Date`, in quanto il deserializzatore si limiterà a verifi-

[7] Il formato ISO tronca la data al secondo, perdendo quindi parte dell'informazione. La perdita di precisione potrebbe causare problemi in applicazioni il cui funzionamento dipenda dall'uso di timer precisi al millisecondo, come ad esempio i giochi.

```
var testStr =
  '{id:"/dog/1",eats:{$ref:"/cat/2"},aTime:"2008-11-07T20:26:17-07:00"}';
var schemas = {
  "/dog/" : {
    prototype : {barks:true},
    properties : {aTime:{format:'date-time'}}
  },
  "/cat/" : {
    prototype : {meows:true}
  }
}
var deserializedDateUsingSchema =
  (dojox.json.ref.fromJson(testStr, {schemas:schemas})).aTime;
```

Listato 9.1: Esempio di deserializzazione di proprietà di tipo `Date`

carne la presenza nella lista. Questa semplificazione ha l'ovvio svantaggio che, nel caso l'oggetto abbia proprietà di tipo `Date` con lo stesso nome a profondità diverse, solo una di queste verrà recuperata.

schemas
Il parametro schemas è legato a un altro lavoro di Kris Zyp, come spiegato in [29]: la proposta di ufficializzazione presso l'IETF dell'internet-draft JSON schema[a], ovvero un formato di documento per definire la struttura di un oggetto JSON un po' come XSD lo è per XML.

[a] http://tools.ietf.org/html/draft-zyp-json-schema-02

9.2.2.1. Serializzazione delle date

Se per deserializzare gli oggetti di tipo `Date` sono riuscito a trovare una tecnica per sfruttare la libreria dojox.json.ref, purtroppo non ci sono riuscito per quanto riguarda la serializzazione.

Per farlo ho dovuto modificare a mano la libreria, in modo che ogni volta che viene trovato un oggetto di tipo `Date` venga aggiunta una voce alla proprietà `properties` dell'oggetto `schemas`.

9.3. Timers

I timer interferiscono con la migrazione dello stato JavaScript nei seguenti modi:

1. presenza di timer pendenti al momento della migrazione;

```
window._mig_setTimeout = window.setTimeout;
window.setTimeout = function(handler, time) {
  var timerID = addTimer(handler, time);
  startCentralTimer();
  return timerID;
};
```

Listato 9.2: Sovrascrittura della funzione `setTimeout`

2. presenza di variabili contenenti riferimenti a timer.

Purtroppo non esistono API che mettano a disposizione la lista dei timer attivi, né un modo per ottenere un riferimento al timer a partire dal suo identificatore numerico. Tutto ciò che è consentito fare con un timer è crearlo ed innescarlo con la funzione `setTimeout` [`setInterval`] o fermarlo con la funzione `clearTimeout` [`clearInterval`].

Data la semplicità di tali funzioni, in questa tesi si è adottata la strategia della *sovrascrittura del codice*. Ovvero ad ogni inizializzazione del codice JavaScript le funzioni originali `set/clear Timeout/Interval` sono salvate in variabili con un nome diverso, mentre alle proprietà originali vengono assegnate nuove funzioni, che aggiungano le funzionalità che ci interessano ed in particolare il mantenimento dei timer attivi in una struttura centralizzata.

L'importante è che le nuove funzioni forniscano la stessa interfaccia delle funzioni originali, in modo che possano funzionare in modo intercambiabile.

9.3.1. Implementazione

Per implementare la strategia della sezione precedente ci siamo serviti di una Map `timers` pubblica (in modo che venga automaticamente inserita nello stato) e di una Map `_timers` privata (che invece non sarà accessibile dal serializzatore). La funzione `addTimer`, innescata ad ogni invocazione di `setTimeout`, aggiorna entrambe le Map. Nella prima Map viene semplicemente aggiunto un letterale oggetto `{"handler": handler, "time": time}` all'indice `timerId`. Mentre nella seconda viene aggiunto un oggetto `Timer`, che fornisce una vera simulazione dei timer originali. L'oggetto `Timer` ha anch'esso i campi `handler` e `time`. Inoltre ha anche un campo `lastTick`, contenente il tempo dell'ultimo aggiornamento e il metodo `update` per aggiornare il Timer. Il campo `time` è utilizzato come contatore alla rovescia che viene aggiornato a intervalli regolari, fino ad innescare l'invocazione di `handler` quando raggiunge il valore zero.

Tutti i timer attivi sono aggiornati ad intervalli regolari da un unico *timer centralizzato*, l'unico ad essere gestito dalla funzione originale `setTimeout`. Il timer centrale

```
/** class Timer **/
var Timer = function Timer(handler, time) {
  this.handler = handler;
  this.time = time;
  this.lastTick = new Date().getTime();
};
Timer.prototype.update = function () {
  var newTime = new Date().getTime();
  this.time -= newTime - this.lastTick;
  this.lastTick = newTime;
};
```

Listato 9.3: Classe `Timer`

invoca ad ogni "rintocco" la funzione `update` di ogni timer attivo. Poiché non è prevedibile l'intervallo esatto tra una chiamata e l'altra dell'handler del timer centrale, il metodo `update` dei timer non scala dal tempo residuo una quantità fissa di millisecondi, ma scala la differenza tra il tempo attuale e il tempo dell'ultimo aggiornamento (`lastTick`). Contemporaneamente viene aggiornata anche la lista pubblica dei timer, `timers`.

In pratica la lista pubblica è solo una replica di alcune delle informazioni della lista privata. Nella fase di ripristino a seguito della migrazione, a partire dalla lista pubblica timers, si ricostruirà ex-novo una lista privata `_timers` e si farà ripartire il timer centralizzato, ripristinando così lo stato dei timer attivi del dispositivo sorgente.

9.4. Proprietà dinamiche

Ad istanze di oggetti di qualsiasi tipo possono essere assegnate proprietà dinamicamente, ma solo i letterali oggetto conserveranno tali proprietà quando verranno serializzati con JSON.

Può capitare che in uno script JavaScript vengano assegnate proprietà anche ad oggetti che non siano Map. Se tali proprietà non vengono correttamente conservate nello stato, si potrebbe avere una perdita di informazioni.

Ad esempio (listato 9.4), in una delle applicazioni JavaScript che abbiamo preso come riferimento (PacMan, capitolo 10), a degli Array vengono assegnate proprietà non numeriche determinate a runtime, utilizzandoli di fatto come se fossero degli Array Associativi.

Il serializzatore JSON ignorerà tutte le proprietà non numeriche dell'array, in questo esempio tutte, dando come risultato un array vuoto.

```
var divNames = [
  'g|r1','g|r2','g|r3','g|r4',
  'pac|r','gameOver',
  'bonus|r1','bonus|r2','bonus|r3','bonus|r4'
];
divisions = new Array();

for (var i=0; i<divNames.length; i++)
  divisions[divNames[i]] = ...
```

Listato 9.4: Esempio di uso di proprietà non numeriche negli Array

In questa tesi la libreria dojox.json.ref è stata estesa per poter serializzare anche le proprietà non numeriche, ma solo per quanto riguarda gli Array. Ulteriori sviluppi potrebbero prevedere la serializzazione di proprietà dinamiche per qualsiasi tipo di oggetto.

9.4.1. Aggiungere le proprietà non numeriche degli array a JSON

Per poter serializzare anche le proprietà non numeriche degli Array, abbiamo modificato direttamente il codice del serializzatore della libreria dojox.json.ref.

In particolare c'è una parte di codice della funzione `serialize` in cui viene effettivamente confezionata la stringa rappresentante l'array, "`[ el-1, el-2, ... ]`", dove per ogni elemento `el-i` viene invocato ricorsivamente il serializzatore.

La lista degli elementi (con chiave numerica) è ottenuta mediante un semplice ciclo `for`. Per creare invece la lista degli elementi con chiave non numerica è sufficiente utilizzare un ciclo `for..in` che escluda le proprietà con chiave numerica, grazie alla funzione JavaScript `isNaN` (*is "Not a Number"*) e che aggiorni la Map `_mig_additionalProperties` contenente tutti i valori relativi a chiavi non numeriche (vedi listato 9.5).

Al termine della funzione `serialize`, prima che vengano eliminati gli id temporanei dagli oggetti (inseriti dalla libreria per la gestione dei riferimenti incrociati), si invoca `serialize` sulla Map e si aggiunge la stringa risultante al messaggio JSON precedentemente costruito (vedi listato 9.6). Così facendo si riesce a sfruttare la gestione dei riferimenti della libreria.

9.4.2. Ripristinare le proprietà non numeriche degli array

Per ripristinare le proprietà non numeriche degli array si deve copiare il valore/riferimento di ogni proprietà della Map `_mig_additionalProperties`, presa dallo stato salvato, nei rispettivi array. Ogni oggetto contenuto nella Map è identificato da una

```
for (var prop in array) {
  if (array.hasOwnProperty(prop)) {
    var keyStr;
    // skip non-string keys
    if (typeof prop == "string") {
      // skip numeric keys
      if (isNaN (prop))
        keyStr = prop;
      else
        continue;
    } else
      continue;

    // 'path' is the path from the root (window) to the leaf 'prop'
    _mig_additionalProperties[path][keyStr] = array[prop];
  }
}
```

Listato 9.5: Aggiunta di proprietà non numeriche degli array

```
var json = serialize(it,'#','');
// adding additional dynamic properties of arrays to Json state
if (typeof _mig_additionalProperties == "object")
{
  var additionalProperties = _mig_additionalProperties;
  delete _mig_additionalProperties;
  json = json.substr (0, json.length-1)+", ";
  var json2 =
    "_mig_additionalProperties : " +
    serialize (additionalProperties, '#_mig_additionalProperties', '');
  json = json.concat (json2);
  json += "\n}";
}
```

Listato 9.6: Aggiunta di proprietà non numeriche nel messaggio JSON

```
// Find the pointer to the array we want to restore its properties
var pointer = path2Pointer (path);

// Restoring properties
var properties = _mig_additionalProperties [path];
for (p in properties) {
  if (properties.hasOwnProperty(p) && properties[p]) {
    try {
      // Restore property p
      pointer[p] = properties[p];
    }
    catch (error) {
      JSStateMigrator.log (
        "property '"+p+"' of '"+path+"' cannot be restored. " +
        "Error: \n"+error.toString()
      );
    }
  }
}
```

Listato 9.7: Ripristinare le proprietà non numeriche degli array

chiave che rappresenta il path della proprietà, ovvero un percorso del tipo "`/object/property/subproperty/...`" che illustri come la proprietà discenda dall'oggetto root. Il path viene passato come parametro alla funzione `path2Pointer`, che provvederà a convertirlo in un puntatore all'oggetto contenente la proprietà da ripristinare. (Vedi listato 9.7.)

9.5. Riferimenti al DOM

JavaScript consente di creare al volo elementi del DOM e di appenderli ad altri elementi in modo che il motore di rendering del browser (detto layout engine) li visualizzi direttamente nella pagina.

Nell'esempio seguente si crea al volo un link (anchor) e si appende ad un div già presente nel DOM:

```
var link = document.createElement("a");
link.setAttribute('href', 'mypage.htm');
var parent = document.getElementById("mydiv");
parent.appendChild(link);
```

Per poter serializzare correttamente nodi del DOM abbiamo modificato direttamente il codice del serializzatore, inserendo una procedura che viene attivata nel caso l'oggetto da serializzare sia un'istanza della classe `Element`.

A questo punto ci troviamo di fronte a tre possibilità:

1. l'elemento è appeso ad un nodo del DOM;
2. l'elemento è appeso ad un nodo del DOM ma non è dotato di un id;
3. l'elemento *non* è appeso ad un nodo del DOM;

Poiché la piattaforma di migrazione serializza l'intero albero del DOM del dispositivo sorgente e lo ricostruisce nel nodo destinatario, nel caso 1 ci basterà inserire nello stato JavaScript l'id dell'elemento. In fase di ripristino sarà invece sufficiente copiare il riferimento del nodo del DOM destinatario nella variabile da ripristinare tramite:

```
anObject.aProperty = document.getElementById("myID");
```

Il caso 2 si riduce facilmente al caso 1 assegnando un id al nodo del DOM. La scelta dell'id è arbitraria, l'importante è mantenere l'univocità ed evitare id che potrebbero essere già stati usati.

```
myElement.id = getUniqueId();
```

Ma come facciamo a sapere se l'elemento fa parte del DOM senza poter utilizzare la funzione `getElementById`?

Un modo sicuro potrebbe essere navigare esaustivamente l'intero albero del DOM confrontando l'elemento in questione con ogni nodo dell'albero. Per evitare una ricerca così onerosa per ogni elemento da serializzare, si può utilizzare un'euristica basata sul tipo di elemento. Ad esempio un elemento di tipo `Image` può essere cercato nell'array `document.images`, contenente tutte le immagini della pagina. Allo stesso modo si possono cercare anche i form (`document.forms`). Per la generalità degli elementi si può invece usare la funzione `getElementByTagName`, che restituisce una lista, eventualmente vuota, di elementi aventi un determinato tag.

Una volta verificata l'appartenenza dell'elemento al DOM, in caso positivo gli si assegna un id, e ci si riduce al caso 1; in caso negativo, cioè se l'elemento non fa parte del DOM, ci si riduce al caso 3.

Nel caso 3, ovvero in cui l'elemento non faccia parte del DOM, si è costretti a serializzarlo per valore. Ci sono alcune librerie come JSONML[8], che forniscono funzioni per serializzare e deserializzare elementi del DOM. Nel nostro caso ci siamo limitati, come primo approccio, a gestire solo oggetti di tipo `Image` e `DIV`, i due tipi di elemento utilizzati nell'applicazione di riferimento del PacMan (capitolo 10).

[8] http://jsonml.org/

9.6. Chiusure lessicali

Come abbiamo spiegato nella sezione 8.6, per specifica del linguaggio non è possibile ispezionare lo stato di una chiusura lessicale. Non si può pertanto salvare e ripristinare lo stato di una chiusura a meno che non sia la chiusura stessa a fornire tali funzionalità. Questa è infatti l'idea alla base della soluzione che abbiamo adottato per risolvere il problema, ovvero fare in modo che le chiusure lessicali espongano il proprio stato tramite una funzione `getState`.

Ovviamente non è possibile farlo a tempo di esecuzione per i motivi che sono stati esposti nella sezione 8.6, quindi abbiamo scelto di far modificare al Migration Proxy il codice JavaScript da inviare al client.

9.6.1. Salvare lo stato di una chiusura lessicale

Per capire cosa modificare del codice di una funzione vediamo un esempio:

```
var foo = function (param) {
  var local = 2 * param;
  return local + externValue;
}
```

Mentre i nomi `param` e `local` si riferiscono rispettivamente a un parametro locale e ad una variabile locale e sono pertanto entrambi locali al contesto della funzione, `externValue` si riferisce ad un valore esterno. Nel caso il riferimento alla funzione `foo` sia passato all'esterno del contesto in cui è stata dichiarata, `externValue` continuerà a riferire allo stesso valore, avendo cioè formato una chiusura lessicale.

Al momento della migrazione si presentano quindi due problemi:

1. `externValue`potrebbe non essere incluso nello stato perché non più presente nell'albero delle proprietà che ha come radice l'oggetto globale;
2. la funzione dopo il ripristino potrebbe perdere il riferimento al valore di `externValue`.

Metodo `getState`

Pertanto il primo passo da compiere è fornire alla funzione un modo per esporre i valori della chiusura lessicale, ad esempio tramite una funzione `getState`:

```
var foo = function (param) {
  arguments.callee.getState = function () {
    return {
      "externValue": externValue
    };
```

```
  }
  var local = 2*param;
  return local + externValue;
}
```

Listato 9.8: Aggiunta del metodo `getState alle chiusure lessicali`

`arguments.callee` consente di ottenere un riferimento alla funzione stessa.

Incapsulamento in una chiusura lessicale

In questo modo però viene ricreata la funzione `getState` ad ogni esecuzione della funzione `finnerFoo`. Per evitare che ciò accada si può sfruttare il meccanismo che è alla base del problema che vogliamo risolvere: le chiusure lessicali. Ovvero possiamo incapsulare la funzione `foo` in una funzione anonima che ne ritorni il riferimento:

```
var foo = (function () {
  var innerFoo = function (param) {
    var local = 2*param;
    return local + externValue;
  };
  innerFoo.getState = function () {
    return {
      "externValue": externValue
    };
  }
  return innerFoo;
})();
```

Listato 9.9: Incapsulamento di una chiusura lessicale

La firma e il corpo della funzione originaria sono rimasti intatti, ma è stato aggiunto un assegnamento alla variabile `innerFoo`, la creazione del metodo `getState` e il ritorno della funzione `innerFoo`. Il tutto è diventato il corpo di una funzione anonima che viene eseguita non appena è interpretata, ottenendo che alla variabile `foo` sia assegnata proprio la funzione `innerFoo`.

Dichiarazione e assegnamento

Nell'esempio precedente abbiamo preso in considerazione una funzione creata tramite un'espressione e assegnata ad una variabile. In JavaScript però si possono creare funzioni anche semplicemente dichiarandole:

```
function foo (param) {
  //...
}
```

Per poter sfruttare la soluzione che abbiamo scelto dobbiamo ricondurre le dichiarazioni di funzioni a espressioni-funzione:

```
var foo = function foo (param) {
  //...
}
```

In questo caso lasciamo anche l'identificatore "`foo`" dopo la parola chiave `function` perché potrebbe servire ad invocare la funzione ricorsivamente.

9.6.2. Migrazione di chiusure lessicali

Se al momento della migrazione, il serializzatore incontra una funzione, deve quindi verificare se questa è dotata di metodo `getState`. Se non lo è, allora vuol dire che non è una chiusura lessicale e che è sufficiente salvare il corpo della funzione (se accessibile); altrimenti bisogna anche salvare lo stato della chiusura lessicale, che per l'appunto può essere facilmente ottenuto invocando il metodo `getState` ad essa associata.

In modo analogo alle proprietà non numeriche degli array (sezione 9.4), gli stati delle chiusure lessicali verranno salvati in una Map `_mig_functionStates` indicizzata mediante i path relativi all'oggetto globale `window`[9].

9.6.3. Ripristinare lo stato di una chiusura lessicale

Se si ripristina una funzione su un dispositivo destinatario della migrazione creando una nuova funzione a partire dal corpo salvato nello stato e assegnandola ad una variabile, tale funzione perderà potenzialmente i riferimenti a valori esterni al suo contesto.

Simulare la chiusura lessicale

Per ripristinare le chiusure lessicali sul dispositivo destinatario possiamo ricorrere ad una simulazione che consiste nel dichiarare una variabile locale per ogni riferimento a valore esterno e assegnando a tali variabili il valore contenuto nello stato salvato della funzione:

```
var foo = (function () {
  var externValue = getValueFromFunctionState(functPath,"externValue");
  var innerFoo = function foo (param) {
    var local = 2 * param;
    return local + externValue;
```

[9] Si noti che la modifica del codice delle funzioni è fatta a tempo di compilazione e non a runtime, quindi non si conoscono i path relativi delle variabili cui saranno assegnate le chiusure lessicali. Si dovrà quindi prevedere un meccanismo per collegare a runtime il metodo `getState` della chiusura lessicale ai path relativi delle variabili che lo conterranno.

```
  };
  innerFoo.getState = ...
  return innerFoo;
})();
```

Listato 9.10: Simulazione di chiusura lessicale

La funzione `getValueFromFunctionState`, la cui implementazione per adesso rimane indefinita, va a pescare nello stato salvato il valore riferito al nome `externValue` all'interno della chiusura lessicale identificata dal suo `functPath`. Se tale valore è un riferimento ad un oggetto allora si preserverà la consistenza con altri contesti che utilizzino quel valore; se invece il valore è un tipo di dato primitivo, allora verrà copiato nella nuova variabile per valore, perdendo l'eventuale condivisione con altri contesti.

Sincronizzazione della chiusura simulata con lo stato salvato

Ad un costo di un maggior carico di lavoro si potrebbe però ripristinare il valore dallo stato ad ogni esecuzione della funzione e salvare di nuovo il valore nello stato al termine dell'esecuzione, così da mantenere lo stato sincronizzato[10] anche in caso di tipi di dato primitivi, con tutti i possibili utilizzatori. Per assicurarci che il valore venga salvato nello stato indipendentemente dalle possibili interruzioni del flusso di esecuzione (`return` ed errori), si può inserire il suo salvataggio all'interno di una clausola `finally`:

```
var foo = (function () {
  var innerFoo = function foo (param) {
    var externValue = getValueFromFunctionState (functPath, "externValue")
        ;
    try {
      var local = 2 * param;
      return local + externValue;
    } finally {
      setFunctionStateValue (functPath, "externValue", externValue);
    }
  };
  // innerFoo.getState = ...
  return innerFoo;
})();
```

Listato 9.11: Sincronizzazione della chiusura simulata con lo stato salvato

In questo modo si potrebbe fare a meno anche della funzione `getState`, dato che lo stato delle chiusure lessicali è salvato ad ogni loro invocazione.

[10] Non c'è bisogno di utilizzare semafori per la sincronizzazione in quanto JavaScript è single threaded.

9.6.4. Riassumendo

In pratica con questo approccio, si modifica il comportamento delle chiusure lessicali in modo che, invece di accedere a valori contenuti in uno stato "nascosto", accedano ad uno stato contenuto in un oggetto globale.

Il procedimento è schematizzato nella figura 9.1, dove sono evidenziate le modifiche effettuate ad ogni passo.

Tale approccio romperebbe pertanto le funzionalità relative all'information hiding e alla sicurezza fornite dalle chiusure lessicali. Ma è accettabile se utilizzato all'interno di una piattaforma per la quale l'utente abbia espresso fiducia, come è effettivamente verificato nel caso della piattaforma che abbiamo utilizzato per questa tesi.

9.6.5. Implementazione

La modifica del codice sorgente, detta talvolta *strumentazione* (instrumentation), è un argomento classico della teoria dei compilatori. Avendo a disposizione un parser in grado di interpretare la grammatica di EcmaScript, si può costruire un compilatore che, preso in ingresso l'albero di sintassi astratto generato dal parser, modifichi l'albero e lo ritraduca nel linguaggio sorgente.

Effettuare la compilazione è un processo complesso e che rischia di modificare pesantemente il modo in cui il codice appare, ad esempio eliminando i commenti e gli spazi bianchi superflui che, non essendo informazione utile, non vengono inseriti nell'albero di sintassi astratta.

Più che un generatore di codice, quello che serve a noi è un *riscrittore* (rewriter), che sia in grado di sfruttare il parser per riconoscere i punti sensibili del codice e di modificarlo in loco senza generarlo ex-novo.

9.6.5.1. Generatori di compilatori

Esistono numerosi strumenti che consentono, a partire da una grammatica annotata, di costruire lexer, parser e compilatori. Il generatore di parser più noto è probabilmente YACC - Yet Another Compiler Compiler - nato per scrivere parser in C a partire da grammatiche scritte in una notazione simile alla BNF - Backus-Naur Form. Il parser generato da YACC richiede un analizzatore sintattico, che storicamente è stato Lex.

A partire da questi due strumenti sono nati vari derivati che ne hanno ampliato le funzionalità, in particolare per generare lexer e parser scritti in vari linguaggi.

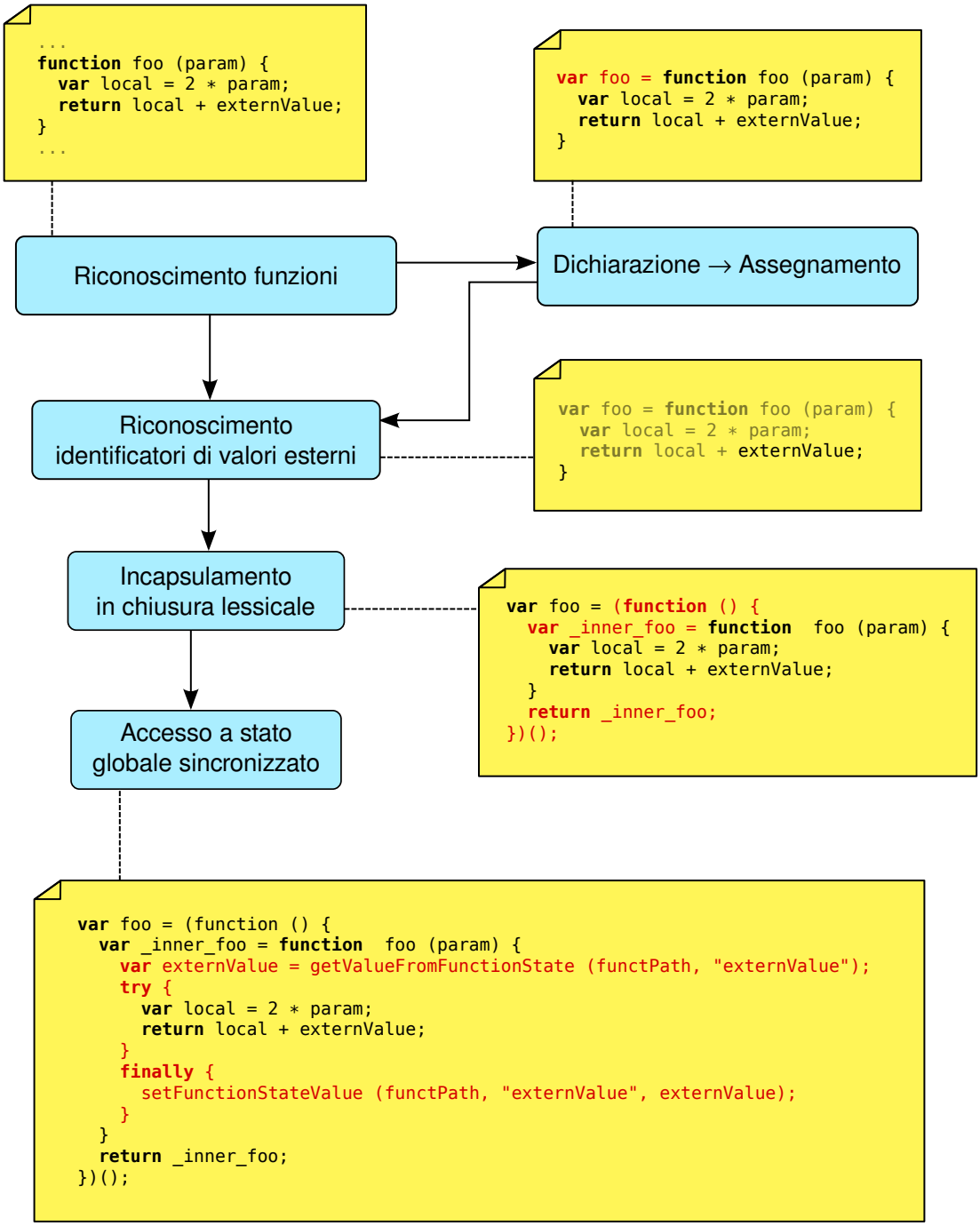

Figura 9.1.: Modifica del codice delle chiusure lessicali

ANTLR

Lo strumento che abbiamo scelto di utilizzare in questa tesi è ANTLR[11] - ANother Tool for Language Recognition - che fornisce un framework per la costruzione di riconoscitori, interpreti, compilatori e traduttori a partire da descrizioni grammaticali contenenti azioni in vari di linguaggi target (attualmente C, Java, Python, C# e Objective-C). ANTLR fornisce anche un eccellente supporto per la costruzione e l'attraversamento di alberi, traduzione, recupero e segnalazione di errori.

Rispetto ai discendenti di YACC, ANTLR produce codice facilmente leggibile sfruttando un più intuitivo parser LL[12] a discesa ricorsiva. Inoltre consente di definire lexer, parser e tree parser con un'unica notazione consistente, facilitando la scrittura di strumenti che utilizzino più componenti.

I lexer generati da ANTLR sono in grado anche di preservare i commenti e gli spazi bianchi in un "canale nascosto" che non verrà preso in considerazione dal parser, ma che sarà comunque disponibile ed utilizzabile per altri scopi (ad esempio la generazione di documentazione).

ANTLR facilita inoltre la generazione di Riscrittori, una categoria di compilatori che non produce un output radicalmente diverso dall'input, ma che effettua solo alcune modifiche minori, che è proprio il caso del lavoro svolto in questa tesi per la modifica delle chiusure lessicali.

9.7. Strategia adottata per il salvataggio e ripristino dello stato

Ora che abbiamo affrontato tutti i sotto-problemi inerenti la migrazione dello stato JavaScript di un'applicazione web, possiamo finalmente mettere insieme i pezzi per risolvere il problema nella sua globalità.

Il meccanismo di salvataggio e ripristino è stato in parte dettato dall'utilizzo della libreria dojox.json.ref, che ha imposto alcuni vincoli per poter sfruttare le sue potenzialità.

Il cuore della gestione dello stato è la libreria JavaScript JSStateMigrator, scritta appositamente per la tesi. Le sue funzioni principali sono:

[11] http://www.antlr.org/

[12] Un parser **LL** interpreta l'input da destra a sinistra (**L**eft to Right) e produce derivazioni sinistre (**L**eftmost derivations) delle frasi. La classe di grammatica interpretabile da questi parser è nota con il nome di grammatiche **LL**.

saveState che fa una "fotografia" dello stato attuale dell'applicazione e restituisce un messaggio JSON che rappresenta una Map contenente:

state lo stato dell'applicazione serializzato in un messaggio JSON;

schemas l'oggetto Schema che consente al deserializzatore di individuare gli oggetti di tipo `Date` (vedi sezione 9.2);

loadState prende come parametro un messaggio rappresentante lo stato dell'applicazione migrata e lo ripristina nell'applicazione del dispositivo destinatario.

9.7.1. Salvataggio dello stato

Abbiamo detto che il meccanismo di base per salvare lo stato di un'applicazione JavaScript consiste nel salvare il valore di tutte le variabili globali, ovvero le proprietà dell'oggetto `window`.

Poiché l'id referencing, ovvero la tecnica di conservazione dei riferimenti tra oggetti basata sull'utilizzo di id univoci (vedi sezione 9.1), è applicabile solo a letterali oggetto cui può essere assegnata una proprietà id, per poter gestire facilmente i riferimenti si sfrutta invece il path referencing. Quest'ultimo gestisce i riferimenti esplicitando il percorso che va dalla radice dell'albero rappresentante l'oggetto alla foglia rappresentate la proprietà riferita.

Lo stato serializzato si ottiene quindi serializzando un letterale oggetto che ha come chiavi i nomi delle proprietà dell'oggetto globale `window` e come valori, i valori (o i riferimenti) delle proprietà di `window`.

```
var globalMap = {};
for (var prop in window) {
// testing if a property is to be excluded
  if ( JSStateMigrator.isExcluded (prop) )
    continue;
  globalMap[prop] = window [prop];
}
```

Listato 9.12: Selezione delle proprietà di `window` da serializzare

Al ciclo `for..in` è stata aggiunta anche una condizione (`isExcluded`) per controllare se la proprietà sia tra quelle da escludere, ovvero tra le proprietà dell'oggetto globale che non hanno a che vedere con lo stato JavaScript dell'applicazione ma che fanno invece parte del BOM (vedi sezione 5.1).

Dopodiché si prepara lo scheletro dello Schema che servirà al parser per identificare le proprietà di tipo `Date`.

```
// JSON Schema skeleton
var schemas = {
  "/mig/" : {
    properties : {}
  },
  "/other/" : {}
};
```

Infine si produce il letterale oggetto da restituire, che contiene lo stato serializzato e lo schema.

```
// CREATING JSON MESSAGE FROM 'globalMap'
var JSONstate = dojox.json.ref.toJson (globalMap, !niceIndent);
// adding Date property names to JSON Schema
schemas["/mig/"].properties = _mig_dateProperties;
delete _mig_dateProperties;
// Creating final JSON message with 'state' and 'schemas'
var JSONmessage = dojox.json.ref.toJson({
  state : JSONstate,
  schemas : schemas
}, niceIndent);
return JSONmessage;
```

La prima chiamata alla funzione `toJson`, oltre a creare un messaggio JSON contenente una chiave-valore per ogni proprietà dell'oggetto globale, inserirà anche chiavi-valori relative a:

additionalProperties le proprietà non numeriche degli array;

domElements i riferimenti a nodi presenti nel DOM;

images le immagini create a runtime e non legate al DOM.

E si occuperà di aggiungere allo Schema una proprietà per ogni oggetto di tipo `Date`.

9.7.2. Ripristino dello stato

Il ripristino dello stato JavaScript dell'applicazione web avviene nel client, dopo che la pagina HTML è stata caricata, invocando il metodo `loadState` della libreria JSStateMigrator.js.

`loadState` estrapola dal messaggio JSON inviato dal Migration Server l'oggetto Schema (che serve a identificare gli oggetto di tipo `Date`) e lo usa come parametro addizionale per deserializzare lo stato JSON.

Dopodiché vengono ripristinate tutte le variabili globali utilizzando un semplice ciclo `for..in`.

```
globalObjects = dojox.json.ref.fromJson (JSONstate, {schemas: schemas});
for (name in globalObjects) {
  try {
    window[name] = globalObjects[name];
  } catch (error)
  {
    JSStateMigrator.log ("poperty "+name+" cannot be restored.");
  }
}
```

Listato 9.13: Ripristino delle variabili globali

A questo punto vengono ripristinate le informazioni che non possono essere catturate da JSON e che sono state inserite in delle mappe aggiunte allo stato dopo la serializzazione delle variabili globali (proprietà non numeriche degli array ed elementi del DOM).

Infine viene invocata la funzione `initVarsAfterMigration` della libreria CentralTimer.js per ricreare i timer pendenti e far partire il timer centrale, come spiegato in sezione 9.3.

```
// Restoring DOM elements
if (globalObjects._mig_domElements)
{
  var domElements = _mig_domElements;
  restoreDomElements (domElements);
}
// Restoring Images not reachable by ID
if (globalObjects._mig_images) {
  var images = _mig_images;
  restoreImages (images);
}
// Restoring non-numeric properties to arrays
if (globalObjects._mig_additionalProperties)
{
  var additionalProperties = _mig_additionalProperties;
  restoreAdditionalProperties (additionalProperties);
}
// Restoring pending timers
_mig_centralTimer.initVarsAfterMigration();
```

Listato 9.14: Ripristino di elementi del DOM, immagini e proprietà non numeriche degli array

```
1  // RESTORING DOM ELEMENTS
2  function restoreDomElements (domElements) {
3    for (var path in domElements) {
4      // FIND THE POINTER to the array we want to restore its properties
5      var split = path.split(".");
6      // the element property name is the last word in the path
7      var prop = split.pop();
8      var pointer =
9        split.length>0
10         ? path2Pointer (split.join("."))
11         : window;
12     // RESTORING ELEMENTS
13     var elementId = domElements [path]["id"];
14     var element = document.getElementById(elementId);
15     if (element) {
16       try {
17         pointer[prop] = element;
18       } catch (error) {
19         console.log ("element with ID '"+elementId+"' at '"+path+"' cannot
               be restored.");
20       }
21     } else {
22       console.log ("element at "+path+" is not accessible by id ("+
             elementId+")");
23     }
24   }
25 }
```

Listato 9.15: Funzione `restoreDomElements`

Le tre funzioni `restoreDomElements`, `restoreImages` e `restoreAdditionalProperties` agiscono secondo una schema molto simile.

Tutte e tre prendono come argomento una Map (`domElements`, `images` o `additionalProperties`) indicizzata da path (percorsi che indicano la posizione della proprietà all'interno dell'albero radicato in `window`[13]) e avente come valori le proprietà serializzate. Per ogni path della Map ottengono il riferimento alla proprietà rappresentata dal path (utilizzando la funzione `pathToPointer`) e ripristinano il valore indicizzato da path.

Si veda a titolo d'esempio il codice della funzione `restoreDomElements` nel listato 9.15.

[13] vedi sezione 9.4.2 per una spiegazione più dettagliata su come vengono costruiti i path.

10. Esempio d'uso: JavaScript-PacMan^{plus}

Per rilassare alcuni vincoli del problema nella sua interezza, abbiamo selezionato un'applicazione JavaScript che richiedesse di risolvere solo un limitato sotto-insieme di problemi per poterne migrare lo stato tra due dispositivi.

Tale applicazione è la trasposizione JavaScript di un famosissimo videogioco arcade degli anni '80: PacMan.

10.1. Le regole

Il gioco consiste nel muovere il protagonista, il Pacman, attraverso un labirinto visto dall'alto. Il Pacman mangia, accumulando punti, dei gettoni-cibo sparsi nel labirinto ed accede al livello successivo non appena li ha consumati tutti. Muovendosi nel labirinto deve fare attenzione a non incappare in uno dei quattro "fantasmini", pena la perdita di una vita. La sua unica arma contro di essi consiste in una "pillola" che, se mangiata, inverte i ruoli, consentendo al Pacman di mangiare i fantasmi, ma solo per un breve intervallo di tempo.

10.2. L'applicazione

L'interfaccia, interamente in HTML, è composta da una tabella rappresentante il labirinto, da una barra con degli indicatori (vite, punti, bonus...) e da una serie di pulsanti per interagire con il gioco ("new game", "speed", "smoothness"...).

Il movimento del Pacman avviene solitamente attraverso l'utilizzo delle frecce della tastiera ma sono stati inseriti anche dei pulsanti per muovere il Pacman, consentendone così l'utilizzo anche in dispositivi privi di tastiera (ad esempio palmari con touch screen).

Figura 10.1.: Screenshot del Pacman

10.3. Codice

Tutta la logica del gioco è interamente implementata sul client, tramite script JavaScript.

Al caricamento della pagina, l'interfaccia viene interamente visualizzata, ad eccezione del labirinto, che viene generato solo al momento della pressione del pulsante "new game". Prima di creare il labirinto viene però invocata la funzione `setApi`, che effettua alcuni test per selezionare le funzioni da utilizzare in base al browser utilizzato (ad esempio la presenza della funzione `getElementById`). Vengono anche inizializzate una serie di variabili globali che verranno utilizzate durante l'esecuzione del gioco.

10.3.1. Ciclo di esecuzione

L'esecuzione continua del gioco è dettata da tre funzioni, che si susseguono in un ciclo infinito scandito da timers:

doMove muove il Pacman e i fantasmi, aggiorna il punteggio e testa la fine del livello (ovvero se è finito il cibo). Infine invoca direttamente la funzione `testCrash`.

testCrash testa le collisioni tra il Pacman e i fantasmi. Se c'è collisione e la "pillola" è attiva, il fantasma viene mangiato e ritorna alla posizione di partenza, altrimenti il Pacman perde una vita e tutti i personaggi tornano alle posizioni di partenza. Quindi viene effettuata la chiamata ritardata di 1 millisecondo a `doMoveFinal`[1]:

```
pacTimer = setTimeout ("doMoveFinal()", 1);
```

doMoveFinal Gestisce la durata della pillola, se attiva, e calcola l'intervallo di tempo `mTo` prima del prossimo frame, dopodiché invoca la funzione `doMove` con timer di `mTo` millisecondi facendo ripartendo da capo con il ciclo di esecuzione del programma.
`mTo` è calcolato nel seguente modo:

```
mTO = dateA.getTime() - dateB.getTime();
mTO += Math.round(aSpeed / aStep);
if (mTO < 5) mTO = 5;
```

[1] Ad un primo sguardo l'utilizzo di `setTimeout` sembra essere solo una soluzione sbrigativa dell'autore per invocare una funzione conoscendone il nome, che potrebbe essere fatto semplicemente con `eval('doMoveFinal()')`. In realtà è possibile che l'autore abbia scelto di usare `setTimeout` per consentire al gestore degli eventi di mandare in esecuzione altri handler di eventi asincroni che sono in attesa di essere eseguiti. Questa è in effetti una tecnica che viene usata talvolta per spezzare lunghe computazioni in modo da mantenere più responsiva l'interfaccia grafica.

Dove `dateA` è l'ora attuale calcolata nella funzione `doMove`, mentre `dateB` è calcolata nella funzione `doMoveFinal`. `aSpeed` e `aStep` sono valori numerici che rappresentano rispettivamente la velocità di gioco e la fluidità (smoothness), impostabili direttamente tramite i controlli dell'interfaccia grafica.

Nota: tutte le chiamate a `setTimeOut` usano la variabile globale `pacTimer` per conservare un rifermento al timer in esecuzione.

10.4. Impatto sulla migrazione

Come abbiamo detto, l'applicazione del Pacman è stata selezionata proprio per la semplicità della sua implementazione, in quanto il suo codice non utilizza:

- librerie di terze parti, come JQuery, Dojo, Prototype, etc...
- oggetti definiti dall'utente, quindi non ci sono `new`, prototipi o costruttori;
- chiusure lessicali;
- chiamate AJAX.

Le caratteristiche che hanno invece pilotato lo sviluppo degli script di migrazione, sono le seguenti:

- oggetti di tipo `Date`;
- timer;
- proprietà non numeriche degli array;
- riferimenti a nodi del DOM.

10.4.1. Riferimenti a oggetti

Il Pacman non fa un uso di riferimenti a oggetti tale da causare problemi durante la migrazione, ma la loro gestione è stata ritenuta necessaria per la maggior parte delle applicazioni web, portando a scegliere come cuore degli script di migrazione la libreria dojox.json.ref, estensione di JSON, che si occupa principalmente di gestire i riferimenti tra oggetti (vedi sezione 9.1).

10.4.2. Oggetti di tipo Date

La maggior parte dei valori utilizzati dal programma e utilizzati da più funzioni è contenuto in variabili globali. Due di queste variabili, `dateA` e `dateB`, contengono oggetti di tipo `Date`, che devono essere quindi aggiunte allo stato, come spiegato nella sezione 9.2.

10.4.3. Timer

A meno che l'utente non metta il gioco in pausa prima di effettuare la migrazione, senza una adeguata gestione dei timer, l'esecuzione del gioco non riprende correttamente dopo la migrazione. Per questo è stato necessaria creare un timer centralizzato e mantenere una traccia dei timer attivi, come spiegato nella sezione 9.3.

10.4.4. Proprietà non numeriche degli array

In Pacman due array, `divisions` e `images`, vengono dichiarati come array, ma sono di fatto utilizzati come Map, impedendo pertanto al serializzatore JSON di serializzare le proprietà con chiavi non numeriche, poiché per definizione (in JSON) gli array sono strutture contenenti valori indicizzati da chiavi numeriche intere. Si veda ad esempio il listato 9.4 a pagina 60.

Per poter serializzare anche le proprietà non numeriche abbiamo esteso il serializzatore come spiegato nella sezione 9.4.

10.4.5. Riferimenti a nodi del DOM

Come se evince dal nome, gli array `divisions` e `images`, contengono rispettivamente DIV e immagini.

Poiché i nodi del DOM non rientrano tra gli oggetti standard serializzati da JSON, è stato necessario estendere il serializzatore come spiegato in sezione 9.5.

Per serializzare i DIV è stato necessario mettere nello stato gli ID che li identificavano poiché sul nodo destinatario è sufficiente invocare la funzione `getElementById` per ottenerne di nuovo il riferimento.

Gli elementi dell'array `images` venivano invece usati solo per conservare i path delle immagini e quindi non erano effettivamente appesi a nodi del DOM. Per migrare i riferimenti alle immagini è stato quindi necessario serializzarli per valore e ricostruirli ex-novo sul dispositivo destinatario.

11. Esempio d'uso: JsTetris

Dopo un caso d'uso semplice come il Pacman (capitolo 10) abbiamo cercato un esempio di applicazione web il cui codice JavaScript fosse più rappresentativo delle applicazioni più comuni, ma anche sufficientemente chiaro da poter essere facilmente testato e debuggato.

In questa ricerca è emerso JsTetris[1], una trascrizione JavaScript del noto gioco arcade Tetris.

11.1. Le regole

Il gioco del Tetris è composto da una griglia 12x22, all'interno della quale calano dall'alto, uno dopo l'altro, dei pezzi geometrici. Il giocatore deve cercare di posizionare le forme l'una accanto (o sopra) all'altra in modo da formare delle righe orizzontali complete. Le righe complete vengono eliminate dal gioco ed incrementano il punteggio. Se i pezzi continuano ad accumularsi senza completare righe fino a raggiungere la sommità della griglia, il giocatore ha perso. L'obiettivo consiste nel completare sufficienti righe per passare al livello successivo, quando la griglia verrà azzerata ed i pezzi inizieranno a cadere più velocemente.

11.2. L'applicazione

Anche in questo caso la logica è interamente implementata sul client con un unico script JavaScript tetris.js. L'interfaccia è composta da un unico DIV (`tetris`), scomposto verticalmente nel DIV contenente il menu e in quello contenente il gioco (`tetrisArea`). Il menu è scomposto a sua volta in altri DIV ciascuno contenente un diverso elemento del menu (pulsanti, prossimo pezzo e statistiche). L'area di gioco è un DIV che di base contiene solo i 6 DIV rappresentanti le barre verticali che fanno da guida e, man mano che i pezzi entrano in gioco, conterrà i cubetti (ciascuno un DIV) che compongono i pezzi.

[1] http://www.gosu.pl/tetris/

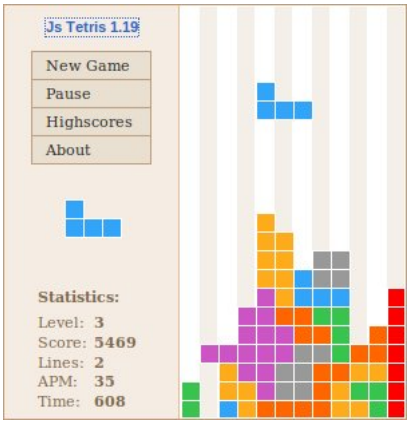

Figura 11.1.: Screenshot di JsTetris

Il modo di apparire e il posizionamento degli elementi del gioco sono interamente gestiti tramite CSS definito nel file tetris.css.

11.3. Codice

Tutto il codice di JsTetris è contenuto all'interno di una funzione-costruttore `Tetris`, una cui istanza è creata alla fine del caricamento della pagina tetris.html e messa in una variabile globale.

```
var tetris = new Tetris();
```

La classe `Tetris` definisce ed inizializza alcune variabili d'istanza (`stats`, `area`, `unit`...) tramite la sintassi:

```
this.instanceVar = value;
```

e alcuni metodi (`start`, `reset`, `pause`...), tramite la sintassi:

```
this.method = function (params) {};
```

Ma prima di tutto, definisce una variabile locale `self` inizializzata a `this`. Quindi, poiché la funzione `Tetris` è stata invocata mediante il costrutto `new`, `self` si riferirà alla nuova istanza appena creata.[2] `self` viene utilizzato in tutti i metodi, che incapsulano quindi il suo riferimento creando di fatto delle *chiusure lessicali*.

Dopodiché vengono dichiarate due variabili locali, `helpwindow` e `hihscores`, inizializzate con istanze di classi definite dall'utente:

[2] vedi tabella 3.3 a pagina 19 per una spiegazione del valore di `this`

```
// windows
var helpwindow = new Window("tetris-help");
var highscores = new Window("tetris-highscores");
```

I valori - o meglio, i riferimenti - contenuti in tali variabili sono poi utilizzati in handler assegnati direttamente a nodi nel DOM, come nel seguente stralcio di codice:

```
// game menu
document.getElementById("tetris-menu-start").onclick = function() {
  helpwindow.close();
  highscores.close();
  self.start();
  this.blur();
};
```

Tali handler formano quindi delle chiusure lessicali.

Come si vede, nel codice di questa funzione anonima di sole 4 righe, assegnata alla proprietà `onclick` del nodo del DOM, ci sono ben 3 istruzioni che fanno riferimento a valori contenuti nella chiusura lessicale.

Infine vengono definite le classi che sono state usate finora come `Window`, `Stats` o `Keyboard`. Queste classi sono costruite in modo analogo alla classe `Tetris` finora descritta. Purtroppo anche queste funzioni-costruttore entrano a far parte della chiusura lessicale di `Tetris`, in quanto sono usate nei suoi metodi.

11.4. Impatto sulla migrazione

JsTetris presenta le stesse difficoltà inerenti la migrazione che abbiamo affrontato con il Pacman (capitolo 10) ma in più presenta due nuove sfide:

- l'utilizzo di tipi definiti dall'utente e di istanze create con costrutto `new`
- le chiusure lessicali

11.4.1. Tipi di dato definiti dall'utente

L'utilizzo di tipi definiti dall'utente e di istanze create con costrutto `new` è un problema che non è stato affrontato in questa tesi nella sua interezza, ma che viene in parte risolto dalle soluzioni realizzate per altri problemi. Ad esempio le variabili d'istanza sono naturalmente incluse nello stato da JSON, mentre i metodi vengono inclusi nello stato grazie all'esportazione di funzioni e chiusure lessicali.

Ciò che viene perduto è un riferimento al costruttore e al prototipo, caratteristiche che in JsTetris non vengono utilizzate.

11.4.2. Chiusure lessicali

Il grande problema su cui ho lavorato nell'ultima parte della tesi è proprio la migrazione delle chiusure lessicali, di cui JsTetris fa un uso intensivo.

Il problema è stato ben definito ed è stata studiata una soluzione plausibile basata sulla costruzione di un compilatore in grado di modificare il codice JavaScript dell'applicazione web (vedi sezione 9.6). Purtroppo il tempo non è stato sufficiente a produrre una versione funzionante del compilatore, che verrà però realizzata nei prossimi mesi, e quindi non è ancora stato possibile verificarne la sua effettiva validità con JsTetris.

12. Conclusione

Quando mi è stata proposta questa tesi, conoscevo il linguaggio JavaScript solo ad un livello estremamente basilare e non avevo quindi una chiara idea su cosa sarebbe stato necessario fare per poter salvare lo stato JavaScript di un'applicazione web. Ma l'argomento *interfacce migratorie* alla base del progetto OPEN[1], in cui si inserisce la tesi, mi affascinava, così questo lavoro mi è sembrato una buona occasione per approfondire l'argomento e per imparare il linguaggio JavaScript.

Nel corso di questi mesi ho sviluppato una conoscenza molto dettagliata del linguaggio e dei meccanismi che lo caratterizzano, consentendomi di isolare una serie di sotto-problemi per i quali fosse relativamente semplice trovare una soluzione. In particolare ho ottenuto la migrazione totale dell'applicazione di riferimento JavaScript-PacMan^{plus} che, seppure molto semplice e poco rappresentativa dell'attuale panorama delle applicazioni web, è stata un ottimo punto di partenza. Le funzionalità sviluppate nella tesi sono:

- salvataggio e ripristino dello stato degli script JavaScript mediante linguaggio di interscambio JSON;
- conservazione dei riferimenti ad oggetti e a nodi del DOM;
- salvataggio e ripristino di oggetti di tipo `Date`, di proprietà non numeriche degli array e di elementi HTML non collegati al DOM;
- conservazione di timer in esecuzione;

Estendere gli script ad una più ampia gamma di applicazioni web ha reso necessario l'analisi di un problema molto complesso: la migrazione delle chiusure lessicali. Infatti la maggior parte dei siti web che si appoggi a JavaScript in modo non triviale, utilizza in qualche forma le chiusure lessicali. In particolare lo fanno i sempre più diffusi framework JavaScript per costruire *Rich Internet Applications* come JQuery, Dojo toolkit o Prototype.

Non essendo possibile accedere da JavaScript ai valori incapsulati nella chiusura lessicale, abbiamo scelto di adottare un approccio basato sulla modifica del codice, in modo

[1] Open Pervasive Environments for migratory iNteractive Services: http://www.ict-open.eu/

che le chiusure lessicali accedessero ad uno stato globale fittizio piuttosto che al loro stato privato. Per modificare il codice abbiamo scelto di utilizzare un tool per la generazione di un compilatore che modificasse automaticamente il codice degli script prima che vengano inviati al client. Purtroppo non c'è stato tempo sufficiente per implementare una versione funzionante del compilatore e non è stato quindi possibile verificare l'effettiva validità della soluzione proposta. Il lavoro verrà però ripreso dopo la tesi, sempre all'interno del laboratorio HIIS[2] del CNR di Pisa, con l'obiettivo di completare il compilatore e di estendere il supporto degli script di migrazione alla più ampia gamma possibile di applicazioni JavaScript.

Un altro problema che andrà affrontato assieme ai ricercatori del laboratorio è la perdita di riferimenti a nodi del DOM in caso di splitting della pagina web su più pagine, che avviene quando si migra un sito web in versione desktop su un dispositivo mobile dal display di dimensioni ridotte. Lo splitting viene effettuato in maniera automatica da un modulo della piattaforma, il semantic redesigner, che tenta euristicamente di mantenere in una stessa pagina gli elementi più inter-correlati. In questo modo ogni pagina risultante dallo split ha un DOM diverso, i cui elementi non sono più rintracciabili dagli script JavaScript. Una possibile modifica consiste nell'effettuare uno splitting logico anziché fisico, mantenendo cioè un'unica pagina in cui gli elementi vengono nascosti o mostrati a seconda della pagina logica che si sta visualizzando.

[2]Human Interfaces in Information Systems: http://giove.isti.cnr.it/

Ringraziamenti

Il primo ringraziamento va alla mia famiglia, per avermi sostenuto moralmente ed economicamente, senza la quale probabilmente non avrei raggiunto questo importante obiettivo.

Grazie anche al Professor Fabio Paternò, per avermi seguito costantamente nella realizzazione della tesi e per avermi concesso la possibilità di proseguire il lavoro anche dopo la Laurea.

Inoltre grazie a:

- gli amici di Siena, Giulio, Nicoletta, Niccolò, Giacomo, Pietro, Duccio e Mirco, che mi dimostrano ogni volta come la lontananza non sia mai distanza;
- i miei compagni di Università, per aver reso lo studio meno pesante e per gli indimenticabili momenti di svago, fra cui vorrei citare in particolare Claudio, Valerio, Alessandro, Emmanuele e Lorenzo;
- i miei coinqulini più "longevi", Andrea, Davide, Enrico e Pasquale, che mi hanno fatto sentire un po' più a casa;
- i numerosi amici che ho conosciuto a Pisa, compagni di giochi, di viaggi e avventure in genere: Daniele, Irene e i loro splendidi bimbi, Isabella, Marco, Millo, Francesca, Giorgio, Maria Giovanna e tanti altri.

Un ringraziamento speciale va infine a "babbo" Ivan, che mi ha adottato in questi anni a Pisa aiutandomi a superare i momenti più difficili.

Elenco delle tabelle

Elenco delle figure

Elenco dei listati

Bibliografia

[1] Comparison of layout engines (ECMAScript). http://en.wikipedia.org/wiki/Comparison_of_layout_engines_(ECMAScript).

[2] Document Object Model. http://en.wikipedia.org/wiki/Document_Object_Model.

[3] ISO 8601. http://en.wikipedia.org/wiki/ISO_8601.

[4] Javascript Closures. http://www.jibbering.com/faq/notes/closures/.

[5] Model View Controller. http://en.wikipedia.org/wiki/Model-View-Controller.

[6] Web browser engine. http://en.wikipedia.org/wiki/Layout_engine.

[7] JavaScript Browser Sniffer. http://jsbrwsniff.sourceforge.net/, 2007.

[8] Gecko DOM Reference. https://developer.mozilla.org/en/Gecko_DOM_Reference, 2010.

[9] Oren Ben-Kiki, Clark Evans, and Ingy dot Net. YAML Ain't Markup Language (YAML) Version 1.2. http://www.yaml.org/spec/1.2/spec.html, 2009.

[10] Douglas Crockford. JSON. http://www.json.org/.

[11] Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly, 2008.

[12] Ian Davis and Maciej Stachowiak. Window Object 1.0. http://www.w3.org/TR/Window/, 2006.

[13] Giuseppe Ghiani, Fabio Paternò, and Carmen Santoro. On-demand Cross-Device Interface Components Migration. In *MobileHCI 2010*. ACM New York, NY, USA.

[14] Ian Hickson. HTML5 - A vocabulary and associated APIs for HTML and XHTML. http://www.w3.org/TR/html5/, 2010.

[15] Michael Kozuch and M. Satyanarayanan. Internet suspend/resume. In IEEE Computer Society, editor, *Fourth IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'02)*, 2002.

[16] M. Mahemoff. *Ajax Design Patterns*. O'Reilly, 2006.

[17] Jeffrey Nichols, Zhigang Hua, and John Barton. Highlight: a System for Creating and Deploying Mobile Web Applications. In ACM, editor, *UIST'08*, Monterey, California, USA.

[18] Stephen Oney and Brad Myers. FireCrystal: Understanding Interactive Behaviors in Dynamic Web Pages. In F. Mattern and M. Naghshineh Berlin Heidelberg, editors, *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 105–108. IEEE, 2009.

[19] Terence Parr. *The Definitive ANTLR Reference*. The Pragmatic Bookshelf, Raleigh, North Carolina and Dallas, Texas, 2007.

[20] Fabio Paternò, Carmen Santoro, and Antonio Scorcia. Ambient Intelligence for Task Continuity across Multiple Devices and Languages. *Computer Journal, the British Computer Society*, 2009.

[21] Paruj Ratanaworabhan, Benjamin Livshits, David Simmons, and Benjamin Zorn. JSMeter: Measuring JavaScript Behavior in the Wild.

[22] John Resig. *Secrets of the JavaScript Ninja*. MEAP, 2008.

[23] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An Analysis of the Dynamic Behavior of JavaScript Programs. In ACM, editor, *PLDI'10*, Toronto, Ontario, Canada.

[24] H. Song, H Chu, N. Islam, S. Kurakake, and M. Katagiri. Browser State Repository Service. In F. Mattern and M. Naghshineh Berlin Heidelberg, editors, *Pervasive 2002*, pages 253–266. Springer-Verlag, 2002.

[25] P. Wilton and J. McPeak. *Beginning JavaScript*. Wiley Publishing, 3 edition, 2007.

[26] Kris Zyp. JSON referencing Proposal and library. http://web.archive.org/web/20071026190351/www.json.com/2007/10/19/json-referencing-proposal-and-library/, 2007.

[27] Kris Zyp. Json Referencing schemes. http://web.archive.org/web/20071026190633/json.com/2007/10/09/json-referencing-schemes/, 2007.

[28] Kris Zyp. JSON referencing in Dojo. http://www.sitepen.com/blog/2008/06/17/json-referencing-in-dojo/, 2008.

[29] Kris Zyp. JSON Schema in Dojo. http://www.sitepen.com/blog/2008/10/31/json-schema/, 2008.