ALMA MATER STUDIORUM
UNIVERSITÀ DEGLI STUDI DI BOLOGNA

Seconda Facoltà di Ingegneria
con Sede a Cesena
Corso di Laurea Specialistica in Ingegneria Informatica

UN APPROCCIO AD ATTORI PER LO SVILUPPO DI APPLICAZIONI WEB STRUTTURATE

Elaborata nel corso di: Sistemi concorrenti e di rete

Relazione di Progetto di:
LUCA DOMENICONI

Relatore:
Prof. ALESSANDRO RICCI

ANNO ACCADEMICO 2011–2012
SESSIONE III

PAROLE CHIAVE

Actor

Web

Dart

Javascript

Isolates

A te Antonietta,
Vorrei solo che il sole e la luna
splendessero per sempre
insieme nel cielo.
Ti amo.

Indice

Introduzione

La programmazione concorrente apre la strada ad interfacce nuove e ricche, che non si bloccano alla prima complessa operazione eseguita, robuste ed affidabili applicazioni che non devono essere riavviate se qualcosa va storto ed infine a performance nettamente migliorate che permettono di scrivere programmi che finora potevano solo essere immaginati.

Purtroppo ragionare in termini concorrenti è intrinsicamente difficile, i processi possono comunicare tra di loro ed agire su entità condivise, generando una serie non indifferente di problematiche. C'è la necessita dunque di astrazioni adatte alla descrizione di problemi concorrenti e che supportino entità di sincronizzazione idonee alla gestione di queste situazioni.

Le attuali astrazioni per la descrivere processi concorrenti come i *thread* sono considerate a basso livello perchè lasciano la gestione della sincronizzazione in mano al programmatore e quindi anche l'onere di evitare tutte le problematiche relative.

Un modello molto interessante, considerato ad alto livello, è quello cosiddetto ad *attori*. Esso permette al programmatore di progettare il proprio sistema senza doversi preoccupare di aspetti di sincronizzazione, inoltre può garantire, se ben implementato, di scrivere codice come se fosse sequenziale (programmazione implicita) unendo anche tutti i vantaggi della programmazione concorrente (programmazione esplicita).

La concorrenza è e sarà un tema caldo non solo per lo sviluppo di applicazioni in generale, ma anche per quanto riguarda l'ambito *web*. Ne è prova il fatto che negli ultimi tempi siano nati diversi linguaggi ed astrazioni per permettere di gestire la concorrenza in questo contesto,

come ad esempio i *web* worker in *HTML5* e *Dart* con i suoi *isolate*, entrambi ispirati al modello ad attori.

E' interessante notare quindi come sia in generale, sia nello scenario web, questo modello stia emergendo e diventando uno di quelli di riferimento, grazie alle sue caratteristiche e peculiarità.

In questa tesi si studierà e si analizzerà a fondo il *modello ad attori* ed in particolare se ne considererà l'implementazione e uso nel caso del nuovo linguaggio *Dart* per il web. Si cercherà di capire se effettivamente *Dart* implementa correttamente questo modello e se tutte le sue proprietà vengono rispettate. Inoltre, grazie all'esecuzione di diversi test di performance, si proverà a vedere se *Dart* e i suoi isolate permettono prestazioni migliori di *Javascript* con *web worker*.

Il lavoro si sviluppa in sei capitoli: nel primo viene descritta la rivoluzione della concorrenza, cioè quali saranno i vantaggi di un approccio di questo tipo e i possibili problemi che si dovranno affrontare. Nel secondo capitolo parleremo del modello ad attori, un'astrazione ad alto livello per descrivere efficacemente problemi concorrenti. Verranno descritte a fondo le proprietà e le caratteristiche che un implementazione di un linguaggio e di un framework ad attori devono garantire, per permettere la produzione di software di qualità. Nel terzo capitolo prenderemo in considerazione Dart, cioè un linguaggio, tutt'ora in fase di sviluppo, per la creazione di applicazioni in ambito web, che si basa sul modello ad attori. Mentre nel quarto capitolo tratteremo quelli che sono gli isolate, cioè l'astrazione che dovrebbe mappare l'entità degli attori in Dart. Nel quinto capitolo, vedremo il contributo di questa tesi, cioè una serie di analisi per determinare se Dart effettivamente implementa correttamente il modello ad attori e se le proprietà più importanti, cioè l'encapsulation e la fairness vengono rispettate. Inoltre si descriveranno i risultati di vari test di performance per confrontare Dart e gli isolate a Javascript con e senza l'utilizzo di web worker. Nell'ultimo capitolo infine, saranno trattate le conclusioni e i possibili sviluppi futuri.

Capitolo 1

La rivoluzione della concorrenza

1.1 Introduzione

La concorrenza si riferisce all' esecuzione in parallelo (quindi nello stesso istante) delle parti che compongono una computazione. Essa è sempre stata propagandata come la via del futuro e la grande idea, ma negli ultimi 30 anni, gli sviluppatori software sono stati capaci di ignorarla. Oggi il nostro futuro parallelo è finalmente arrivato, infatti le macchine odierne sono definitivamente parallele e quindi occorre che il software si adatti per far fronte a questo importante cambiamento. Nelle ultime tre decadi, i miglioramenti nella fabbricazione dei semiconduttori e nelle implementazioni dei processori, hanno aumentato in continuazione la velocità alla quale i computer eseguivano programmi sequenziali. I cambiamenti nelle architetture multicore però, permettono benefici solo ad applicazioni concorrenti e quindi hanno un piccolo valore nell'attuale software mainstream. Infatti nelle nuove configurazioni, i core sono diventati più semplici rispetto ai processori tradizionali, inoltre eseguono le computazioni con una velocità di clock minore per ridurre il consumo di potenza in architetture multicore dense. Per questo motivo, le attuali applcazioni (desktop) non verrebbero eseguite più velocemente ma anzi, ci sarebbe addirittura un calo delle prestazioni. Tutto ciò introduce un punto fondamentale di svolta nello sviluppo del software, almeno per quanto riguarda

quello mainstream. I computer diverranno sempre più performanti, ma i programmi non potranno più cavalcare l'onda del miglioramento hardware a meno che non diventino altamente concorrenti. **Sebbene le performance multicore siano la forza principale per questo cambiamento, ci sono altre ragioni per volere la concorrenza, ad esempio la prima potrebbe essere quella di migliorare la *responsiveness*[1] facendo eseguire i lavori in maniera asincrona piuttosto che sincrona.** D'altro canto la concorrenza è un tema difficile, in primo luogo perchè i linguaggi ed i tool presenti sul mercato sono inadeguati per trasformare le attuali applicazioni in programmi paralleli, poi perchè è difficile trovare il parallelismo nelle applicazioni mainstream e per ultimo, ma sicuramente non come ordine di importanza, perchè la concorrenza richiede ai programmatori di pensare in un modo che gli umani trovano difficile. I linguaggi ed i tool per programmi concorrenti di oggi sono paragonabili, come livello raggiunto, all'inizio della programmazione strutturata[2] nei linguaggi sequenziali. Tutto quello che ci offre la programmazione concorrente sono **strutture come semafori, *coroutine*,*lock* oppure *thread* che sono ancora ad un livello di astrazione tale per cui la costruzione di programmi concorrenti risulta ancora complessa, dato che il programmatore deve tenere in conto esplicitamente delle possibili interazioni con le varie entità condivise**. Quello che si cerca quindi, sono strutture ad alto livello per la costruzione di grandi applicazioni che supportino fortemente la concorrenza e che permettano al programmatore di realizzare questi sistemi in maniera più semplice.

1.2 Un piccolo cenno storico

La concorrenza è stata la conseguenza di tre trend significativi:

[1] La *responsiveness* è la velocità di risposta di un sistema ad una o più richieste di un utente.

[2] La programmazione strutturata è un paradigma di programmazione emerso fra gli anni sessanta e gli anni settanta. Questi linguaggi offrivano un insieme di strutture di controllo del codice che doveva essere eseguito, ovvero forme di sequenza, di alternativa (*if* ad esempio) e di iterazione (*while/do* ad esempio).

1. L'accresciuto uso di processi che interagivano tra di loro (all'epoca, ad esempio, nei programmi eseguiti su X-window[3].

2. Le reti di workstation che stavano diventando un alternativa a costi non proibitivi per la condivisione delle risorse (anche computazionali) e la risoluzione di problemi distribuiti, come ad esempio quelli loosely coupled[4].

3. La tecnologia a multiprocessore che divenne economicamente sostenibile.

E' anche interessante notare come Simula, il primo linguaggio ad oggetti, nato negli anni 60 al Norwegian Computing Center (Oslo) da Ole-Johan Dahl e Kristen Nygaard, fosse anche in grado di implementare un modello di concorrenza, attraverso l' utilizzo delle cosiddette *coroutine*[5], questo fa capire come, già all'epoca questo tema fosse molto sensibile.

1.3 Benefici della concorrenza

Veniamo ora ai benefici della concorenza, cioè ai motivi che dovrebbero spingerci a realizzare applicazioni concorrenti, abbandonando la programmazione tradizionale. Di seguito i principali:

- Sfruttamento dell'architettura hardware disponibile e performance migliori.

[3]X Window System, noto in gergo come X Window o X11 (o, ancora più semplicemente, come X), è di fatto il gestore grafico standard per tutti i sistemi Unix. X-window fornisce l'ambiente e i componenti di base per le interfacce grafiche, ovvero il disegno e lo spostamento delle finestre sullo schermo e l'interazione con il mouse e la tastiera).

[4]I problemi loosely coupled sono quelli che possono essere partizionati in molti piccoli sottoproblemi, tali che l'interazione tra questi ultimi sia limitata.

[5]Le coroutine sono componenti di linguaggi che generalizzano le subroutine permettendo punti di ingresso multipli. Quando viene invocata una subroutine, l' esecuzione comincia dall' inizio del suo codice e una volta che si arriva ad un' uscita (return), essa finisce. Un' istanza di una subroutine ritorna (fa return) una volta sola. Le coroutine sono simili alle subroutine, eccetto per il fatto che si può uscire lasciando il controllo o chiamando un' altra coroutine, permettendo però di rientrare nella coroutine di partenza proprio nel punto in cui essa aveva lasciato il controllo, mantenendo anche il suo stato nel momento in cui l'aveva lasciato.

- *Responsiveness* migliorata.
- Robustezza e affidabilità.
- Possibilità di distribuire la computazione.

Nell'implementazione di un linguaggio di programmazione, si è sempre cercato di astrarre il più possibile l'architettura del sistema computazionale agli occhi del programmatore, in modo che esso potesse concentrarsi sul modello del problema da risolvere piuttosto che sui dettagli architetturali di un sistema specifico. Questo vale anche per la programmi concorrenti ed è per questo che si può parlare di esecuzione *potenziale* in parallelo, poichè è possibile che le componenti di una computazione vengano eseguite sequenzialmente, ad esempio su un' architettura a singolo processore, piuttosto che in parallelo, su un' architettura multi processore, senza che questo possa modificare le scelte progettuali del programmatore. Quindi un sistema concorrente può sfruttare entrambe le architetture, mentre un programma squisitamente sequenziale, sfrutterebbe solo architetture a singolo processore oppure un solo core, sebbene ne siano presenti diversi (questo potrebbe essere problematico per la programmazione sequenziale in futuro, vedi 1.4). Il primo punto fa riferimento quindi alla possibilità di eseguire più istruzioni contemporaneamente. In maniera semplice se si può eseguire più computazione nello stesso intervallo di tempo, migliori saranno le performance. Anche se vedremo che esistono delle problematiche in alcuni tipi di parallelismo (vedi 1.6.3) che potrebbero farle peggiorare sensibilmente.

Come visto nell'introduzione, con la programmazione concorrente è possibile migliorare la *responsiveness* dei sistemi. Ad esempio nelle applicazioni odierne troviamo spesso un interfaccia lato utente, che alla pressione di un tasto deve eseguire delle operazioni. Se le richieste avvengono in maniera sincrona nel codice e le operazioni sono computazionalmente onerose (se non addirittura bloccanti) si rischia che l'interfaccia non venga riaggiornata nel frattempo, con la perdita totale della possibilità di interagire con l'utente.

Il terzo punto si riferisce al problema che può avere un sistema (anche composto da più entità) dove il flusso di controllo è unico. Se una delle entità si blocca, si ferma anche tutto il programma. Con

un sistema concorrente composto da più entità indipendenti che interagiscono tramite messaggistica asincrona, se una di esse si blocca, il sistema può tranquillamente continuare a funzionare e prendere anche le relative contromisure per riportare la situazione alla normalità.

Il quarto punto fa riferimento alla possibilità di avere le entità che costituiscono il nostro sistema su diversi calcolatori in esecuzione nello stesso istante, per migliorare le performance che possono addiruttura crescere se il numero di calcolatori aumenta (scalabilità). Inoltre è possibile, tramite apposite infrastrutture, spostare la computazione da un nodo ad un altro (mobilità), con tutta una serie di benefici come ad esempio il *load-balancing* e tutte le conseguenze positive che ne derivano (questo punto è spiegato ancor più dettagliatamente nel paragrafo 2.9.4).

1.4 Conseguenze e svantaggi della concorrenza

La rivoluzione della concorrenza sarà probabilmente più distruttiva di quella avvenuta per il passaggio all'astrazione Object Oriented, questo fondamentalmente per quattro motivi differenti:

1. Forte dipendenza delle performance ai nuovi sistemi hardware (problema per i linguaggi sequenziali).
2. Difficoltà nel ragionare in termini concorrenti.
3. Complessità nell'analisi dei difetti, debug, ricerca di colli di bottiglia e test dei programmi concorrenti.
4. Se mal gestita: problematiche legate all'utilizzo di risorse condivise.

Il primo punto descrive il problema relativo alla dipendenza delle performance dei programmi dai nuovi supporti hardware, a seconda che essi siano concorrenti oppure no. Alcuni linguaggi come il C hanno ignorato l'OO e sono rimasti utilizzabili, inoltre in certe situazioni si sono dimostrati ancora più performanti dei loro simili scritti con quest'ultima astrazione, questo perchè non c'era dipendenza con le

architetture hardware. Con lo sviluppo di strutture multi-core e l'abbandono di quelle monoprocessore, le uniche applicazioni che potranno avere ottime performance saranno quelle concorrenti, mentre le altre si dovranno adattare al nuovo corso, oppure gradualmente moriranno o saranno utilizzate in ambiti dove l'utilizzo di moderne architetture hardware non è importante.

Il secondo punto si riferisce alla difficoltà degli esseri umani di ragionare in termini di codice concorrente piuttosto che sequenziale. Ad esempio anche persone particolarmente accorte potrebbero mancare possibili interleaving tra semplici collezioni di operazioni parzialmente ordinate. Occorre quindi avere a disposizione delle astrazioni che permettano al programmatore di gestire situazioni anche complesse in maniera semplice (vedi 1.7.1).

Il terzo punto è strettamente correlato al precedente, infatti il fatto che per le persone sia difficile ragionare in termini concorrenti fa si che debbano essere disponibili dei tool che aiutino il programmatore durante tutto il ciclo di vita del nostro software. E' dimostrabile come questi tool sviluppati per la programmazione concorrente debbano essere notevolmente più complessi di quelli utilizzati nella programmazione sequenziale.

L'ultimo punto indica la possibilità che l'utilizzo nel nostro sistema delle risorse condivise (memoria, oggetti...) non adeguatamente progettate, porti ad incappare in problematiche distruttive come ad esempio *corse critiche* (*Race Conditions*), *stalli* (*Deadlock*) o *morte di stenti* (*Starvation*). Nel paragrafo 1.6.4 si tratterà a fondo questo tema.

1.5 Differenze di concorrenza tra applicazioni client e server

La concorrenza è un tema sfidante per le applicazioni client-side mentre per molti programmi server-side è un problema risolto. Infatti in questo tipo di contesto esistono già delle soluzioni che funzionano bene e e delle architetture quasi di routine, anche se programmarle e scalarle può richiedere ancora un grosso sforzo. Queste applicazioni tipicamente hanno un'abbondanza di parallelismo, dato che devono

maneggiare simultaneamente una grande quantità di stream, cioè flussi di dati relativi a diverse richieste. Ad esempio un web server esegue indipendentemente migliaia di copie dello stesso codice in maniera isolata, codice che agisce su dati per la maggior parte non sovrapposti, in quanto ogni richiesta avrà il suo contesto e fondamentalmente non necessita di condividere informazioni con altre richieste diverse. L'unico punto in comune può essere un' entità di memorizzazione astratta come ad esempio un database, che però per come è strutturato, permette un accesso fortemente concorrente, nonostante ci sia comunque la necessità di sincronizzazione tra i processi. In questa maniera la rete può dare la sensazione di vivere ancora in un universo single-thread.

Il mondo delle applicazioni client non è invece per nulla così ben strutturato e regolare. Una tipica applicazione client esegue una computazione relativamente breve per conto di un singolo utente e la concorrenza è ricercata dividendo la computazione in parti più piccole. Queste parti possono interagire e condividere dati in una miriade di modi. Quindi, quello che rende questo tipo di programmazione difficile risulta essere:

- Il codice concorrente non omogeneo e di piccola dimensione (mentre nel server si esegue quasi sempre lo stesso tipo di codice ed esso è di dimensioni maggiori).
- Le interazioni sono complicate (nel server si interagisce praticamente solo con il database).
- Le strutture dati sono non omogenee e non adatte alla concorrenza (non come un database).

1.6 Problemi e modelli di programmazione

Oggi si può esprimere il parallelismo tramite un numero diverso di modi, ognuno dei quali è applicabile solo ad un sottoinsieme di programmi. In molti casi è difficile, senza una progettazione ed un'analisi accurata, capire in anticipo che modello è appropriato per un

dato problema ed è sempre intricato combinare diversi tipi di modelli quando una data applicazione non calza in un singolo paradigma. Questi modelli di programmazione differiscono significativamente in due dimensioni:

- Granularità delle operazioni parallele.
- Grado di accoppiamento tra i task.

Punti diversi in questo spazio bidimensionale favoriscono certi modelli di programmazione, è quindi importante esaminarli correttamente.

Le operazioni eseguite in parallelo possono spaziare da una singola istruzione, come un addizione o una moltiplicazione, a complessi programmi che possono richiedere giorni o ore per essere eseguiti. Ovviamente per piccole operazioni, i costi di overhead di una infrastruttura parallela sono significativi. Le architetture multicore riducono i costi di sincronizzazione e comunicazione rispetto alle convenzionali architetture multiprocessore, questo fa ridurre i costi di overhead su piccoli pezzi di codice.

L'atra dimensione presa in considerazione è il grado di accoppiamento nella comunicazione e sincronizzazione tra le operazioni. L'ideale sarebbe che non ci fossero per niente in modo tale che le operazioni vengano eseguite indipendentemente e producano distinti output. In questo caso, le operazioni possono essere eseguite in qualunque ordine, senza ovviamente incorrere in costi di comunicazione e sincronizzazione, inoltre sono facilmente programmabili, senza che si incorra in problemi di corse critiche[6]. Questa situazione è estremamente rara dato che la maggior parte dei programmi concorrenti condividono dei dati. La complessità di assicurare operazioni corrette ed efficienti aumenta con l'aumentare della diversità delle operazioni. Il caso più semplice è quello di eseguire lo stesso codice per ogni programma concorrente. Questo tipo di condivisione è così regolare che può essere analizzata prendendo in considerazione un solo task. Ovviamente il caso più difficile e sfidante è quello di eseguire delle operazioni diverse in modo tale che il pattern di condivisione sia più complesso da comprendere.

[6]Le race condition o corse critiche sono delle situazioni che si verificano quando due o più processi stanno leggendo o scrivendo un qualche dato condiviso e il risultato finale dipende dall'ordine in cui vengono schedulati i processi.

1.6.1 Parallelismo indipendente

Il modello più semplice e meglio funzionante è il cosiddetto *parallelismo indipendente* (chiamato anche modello dei *task imbarazzantemente paralleli*) nel quale una o più operazioni sono applicate indipendentemente ad ogni oggetto in una collezione di dati. Un esempio potrebbe essere quello di moltiplicare ciascun elemento di una matrice per se stesso e memorizzare il valore trovato nella stessa locazione:

Listing 1.1: Esempio di parallelismo indipendente a granultarità fine delle operazioni parallele

```
double A[100][100];
...
A = A*2;
```

Come si può facilmente notare, tutti i diecimila task risultano essere indipendenti e sono eseguibili senza coordinazione, inoltre la loro granularità è molto fine in quanto viene utilizzata solo una linea di codice! Questa concorrenza è probabilmente molta di più di quanto sia necessaria in un computer normale (vista la sua granularità fine), così un approccio più pratico potrebbe essere quello di partizionare la matrice in blocchi più grandi ed eseguire le operazioni su questi, concorrentemente. Non è detto quindi che il parallelismo indipendente debba essere per forza quello a grana fine. Un ulteriore esempio può essere quello dei motori di ricerca che condividono solo un enorme database read-only oppure quelle simulazioni che richiedono di eseguire delle computazioni con un grande spazio di dati di input.

1.6.2 Parallelismo regolare

Il passo successivo al parallelismo indipendente è quello di applicare le stesse operazioni ad una collezione di dati quando però le computazioni sono mutualmente dipendenti. Un operazione su una parte di dati è dipendente da un'altra operazione se c'è la necessità di una comunicazione o una sincronizzazione tra le due perchè queste avvengano correttamente. Per esempio, consideriamo una computazione su una matrice che rimpiazza ogni punto con la media dei suoi quattro vicini:

Listing 1.2: Esempio di operazione su una matrice che necessita di coordinazione (parallelismo regolare)

```
A[i,j] = ( A[i-1,j] + A[i,j-1] +
  A[i+1,j] + A[i,j+1]) / 4;
```

Questa computazione richiede un attenta coordinazione per assicurarsi che ogni vicino legga la nostra locazione, prima che venga rimpiazzata con la media. Se lo spazio non è un problema, allora le medie possono essere memorizzate in un nuovo array. Questa modalità risolutiva ci fa tornare al caso precedente di parallelismo indipendente, però non è sempre possibile metterla in atto. Come già detto in precedenza, il parallelismo regolare, richiede sincronizzazione o comunque delle strategie ottimamente orchestrate per produrre corretti risultati, inoltre è anche possibile, a differenza di un parallelismo generico, analizzare il codice per determinare come può avvenire l'esecuzione e quali dati vengono condivisi. Quest'ultimo vantaggio è solo teorico dato che l'analisi dei programmi è una disciplina imprecisa ed è praticamente impossibile per i compilatori capire e ristrutturare, laddove ce ne sia necessità, codici concorrenti sufficientemente complessi. In fondo all'asse della granularità troviamo, come già visto nel paragrafo 1.5, i server web, che sono formati da processi identici ed indipendenti tranne per l'accesso ad un database comune. In questo caso la computazione è eseguita in parallelo senza una grande necessità di coordinazione sulle transizioni del database.

1.6.3 Parallelismo non strutturato

La più generale e meno disciplinata forma di parallelismo, si ha quando la computazione differisce, in questo modo **gli accessi ai dati non sono predicibili e hanno bisogno di essere coordinati attraverso un qualche tipo di sincronizzazione**, se questo non accade i risultati dell'esecuzione saranno non deterministici. Questa è la forma di parallelismo più comune nei programmi che fanno uso di *thread* e di entità per la sincronizzazione esplicita come i *lock*.

1.6.4 Il problema dello stato condiviso

L'aspetto fortemente sfidante del parallelismo non strutturato è la condivisione di risorse. Un'applicazione client tipicamente manipola una memoria condivisa formata da oggetti interconnessi in maniera impredicibile. Quando due processi provano ad accedere allo stesso oggetto, cercando di modificarne lo stato, e non facciamo niente per coordinarne le azioni, andiamo incontro a corse critiche. Ovviamente queste ultime sono un grande problema per il nostro sistema dato che i processi potranno leggere e scrivere valori inconsistenti o corrotti. Per questo motivo esistono una ricca varietà di dispositivi di sincronizzazione che possono prevenire le corse critiche e il più semplice di tutti questi è il *lock* (vedi B.1). Come si può vedere dall'analisi fatta in B.1.2 i *lock* portano ad una serie di problemi non indifferenti che non solo degradano le prestazioni ma che possono mettere in seria difficoltà l'esecuzione di un applicazione concorrente. Anche l'utilizzo di architetture *lock-free* (vedi B.1.3), nonostante possano dare dei vantaggi in alcune situazioni in termini prestazionali possono portare ad altri problemi non considerati in precedenza. Inoltre sebbene siano temi promettenti sono ancora soggetto di ricerche attive.

1.7 Metodologie di programmazione concorrente

Esistono fondamentalmente tre diverse possibilità per introdurre la concorrenza in un linguaggio di programmazione, cioè in maniera:

- Esplicita (tramite astrazioni ad alto o basso livello).
- Implicita (tramite librerie o API).
- Automatica.

La programmazione *esplicita* si verifica quando il programmatore deve dichiarare esattamente quando la concorrenza si verificherà nel programma. Il vantaggio maggiore si trova nel fatto che i programmatori possono esprimere il pieno potenziale della concorrenza nelle

applicazioni, lo svantaggio invece è che per funzionare in maniera produttiva, essa richiede l'utilizzo di astrazioni ad alto livello. Se infatti le astrazioni fossero di basso livello, sarebbero i programmatori a doversi sobbarcare in prima persona tutto il lavoro di gestione delle interazioni con le entità concorrenti (con particolare attenzione a quelle che condividono il loro stato).

La programmazione *implicita* nasconde la concorrenza dietro librerie o API in modo tale che il chiamante possa lavorare come se fosse all'oscuro della esecuzione in parallelo della chiamata. Questo approccio permette ai programmatori meno specializzati di usare la concorrenza in maniera sicura mentre gli svantaggi si trovano fondamentalmente nel fatto che non è possibile sfruttare a pieno tutto il potenziale della concorrenza che potrebbe avere il tipo di problema affrontato ed inoltre non è sempre possibile evitare di esporre la concorrenza in tutte le circostanze. Il peggiore sfruttamento delle potenzialità è dovuto essenzialmente al fatto che astrazioni esplicite possono permette a tutto il codice di essere eseguito direttamente su processori o core differenti, mentre se richiediamo della concorrenza on-demand tramite l'utilizzo di API o librerie solamente il codice chiamato potrà essere concorrente.

La programmazione *automatica* è un approccio largamente studiato, dove il compilatore prova a trovare il parallelismo da solo in maniera automatica (da qui il nome). Questo approccio pur essendo molto attraente si è rivelato poco efficiente nella pratica, dato che è necessaria un'accurata analisi per capire il potenziale comportamento del programma. Fondamentalmente esso risulta più adatto per semplici linguaggi di programmazione come il *Fortran* piuttosto che linguaggi come il *C* che risultano più complessi anche per l'utilizzo dei puntatori. Inoltre i linguaggi sequenziali spesso utilizzano algoritmi posti in sequenza e che quindi complessivamente non contengono molta concorrenza.

1.7.1 Quali metodologie utilizzare?

Dato che la programmazione *automatica* come visto nel paragrafo 1.7 si è rivelata poco efficiente, occorrerebbe combinare i vantaggi della programmazione *esplicita* con quelli dell'*implicita*. Quindi da una par-

te il vantaggio di avere astrazioni ad alto livello che sfruttino a pieno il potenziale della concorrenza e dall'altro la semplicità di scrivere il codice come se fosse eseguito sequenzialmente.

La maggior parte dei linguaggi di programmazione odierni offrono una programmazione *esplicita* tramite l'uso di astrazioni di basso livello come i *thread* e i *lock*. Esse possono portare a tutta una serie di problematiche come quelle dovute allo stato condiviso (vedi 1.6.4) e da codice non rientrante[7] ed inoltre fanno si che il programmatore debba gestire le interazioni tra le varie entità, per questo motivo non si può parlare assolutamente di scrittura di codice come fosse sequenziale.

Le astrazioni ad alto livello che vedremo, permettono invece di concentrarsi sul problema da risolvere e non sulle questioni riguardanti le interazioni tra le entità concorrenti. Infatti in questo caso, nella scrittura di un task, il programmatore dovrebbe avere l'illusione di scrivere codice sequenziale, con tutti i vantaggi del caso. Tre esempi di questo tipo di astrazione sono:

- *Chiamate asincrone.*
- *Future.*
- *Active object.*

Una *chiamata asincrona* è una chiamata ad una funzione o metodo che è eseguita in maniera non bloccante, cioè in modo che il chiamante non si blocchi in attesa dell'esecuzione e della risposta conseguente alla sua esecuzione. Il chiamante continuerà nel suo lavoro e concettualmente la chiamata sarà effettuata tramite un messaggio ad un altro task diverso o creato all'uopo che eseguirà l'operazione richiesta indipendentemente.

Una *future* è un meccanismo per permettere di ricevere un risultato da una chiamata asincrona. Esso è una sorta di segnaposto dove sarà inserito il risultato della chiamata, una volta che sarà disponibile.

[7] Un codice rientrante può essere interrotto in maniera safe (quindi senza creare problemi come ad esempio l'inconsistenza di variabili condivise) durante la sua esecuzione, rieseguito (*re-entered*) , prima che la prima esecuzione sia portata a termine.

L'ultimo esempio di astrazione ad alto livello sono i cosiddetti *active object*. Si tratta concettualmente di entità che possono essere eseguite indipendentemente, e che si comportano come dei veri e propri *monitor* dove solamente un metodo dell'oggetto è eseguito a run-time, ma senza richiedere l'utilizzo di lock (il tutto è fatto in maniera implicita, come espressamente richiesto in precedenza). Inoltre le chiamate esterne ai metodi sono eseguite tramite messaggi asincroni (*chiamate asincrone*) e gestiti tramite l'utilizzo di code. Gli *active object* possono essere implementati in diverse maniere, un esempio sono proprio gli attori che vedremo nei capitoli successivi.

E' interessante notare come il linguaggio *Dart* che verrà trattato nel capitolo 3 utilizzi proprio le tre astrazioni ad alto livello viste, come le *chiamate asincrone* (tramite l'uso di messaggi e porte),le *future* e gli *active object* (gli attori o meglio gli *isolate*) per la realizzazione di sistemi concorrenti.

Capitolo 2

Gli Attori

2.1 Descrizione

Il modello ad attori è un modello di computazione originato nel 1973 basato sull'idea che la computazione concorrente può essere descritta tramite un sistema di entità comunicanti, chiamate attori [9]. Gli attori sono componenti autosufficienti, interattivi ed indipendenti di un sistema che operano concorrentemente e possono comunicare attraverso il message-passing asincrono [1]. Grazie al loro utilizzo è possibile sviluppare sistemi paralleli, distribuiti e mobili [4].

2.2 Caratteristiche degli attori e di un sistema ad attori

Possiamo riassumere brevemente le caratteristiche salienti di un attore:

- Possono ricevere e rispondere ai messaggi di altri attori (interattività).
- Comunicano con il protocollo bufferizzato asincrono[1].

[1] Questo modella il ritardo e il nondeterminismo nell'ordine di arrivo nelle comunicazioni dei processi distribuiti. Infatti il percorso che prende un messaggio sulla rete che lo porta dal destinatario al mittente e il ritardo che esso accumula non è specificato, in questa maniera l'ordine di arrivo dei messaggi è indeterminato.

- Ogni attore ha il suo stato locale (non condiviso[2]) che è composto da delle variabili locali e da un cosiddetto *comportamento*. Esso può aggiornare il suo stato locale (indipendenza/incapsulamento dello stato) solo attraverso l'uso dei messaggi (cambiando lo stato è possibile anche cambiare il comportamento dell'attore stesso).

- Il loro comportamento è un insieme di azioni eseguite in maniera atomica che l'attore tenterà di portare a termine a seconda del messaggio che gli è giunto (esecuzione atomica dei metodi, detta anche *macro-step semantics*).

- Ogni attore effettua le sue azioni concorrentemente con gli altri attori.

- Possono creare nuovi attori.

- Hanno una vita indefinita.

- Un attore ha un nome che è globalmente unico specificato all'atto della sua creazione ed un comportamento che determina le sue azioni. Il nome inizialmente è conosciuto solo al creatore dell'attore.

Inoltre un framework ad attori dovrebbe presentare le seguenti proprietà:

- *Fairness* nello scheduling degli attori e nella consegna dei messaggi.

- *Location trasparency* (trasparenza nella località) permette di scrivere codice senza dover conoscere il luogo in cui è in esecuzione l'attore considerato o gli attori considerati.

- *Transparent migration* o *migration* (trasparenza nella migrazione) permette al sistema di spostare gli attori sulla rete computazionale dipendentemente dalle necessità.

[2]Questo significa che due attori, per come è strutturato il sistema, è impossibile che abbiano un campo condiviso. Essi avranno la loro percezione del mondo separata, ognuno dagli altri.

2.3 Funzionamento

All'atto di creazione di un attore, la mailbox (l'entità che colleziona i messaggi ricevuti da ogni attore) è vuota. Da questo momento, la vita di un attore può essere vista come un un loop in esecuzione, dove all'interno sono eseguiti in maniera sequenziale i seguenti passi:

1. Rimuove un messaggio dalla sua mailbox (spesso implementata come una coda).

2. Decodifica il messaggio e esegue il corrispondente metodo richiesto all'interno del messaggio stesso.

Se la mailbox di un attore è vuota, esso si blocca in attesa del prossimo messaggio (tali attori bloccati vengono definiti *idle actors*). Il processamento di un messaggio può causare l'aggiornamento dello stato locale di un attore, la creazione di nuovi attori e la spedizione di nuovi messaggi. E' importante notare come non ci siano interferenze tra messaggi che sono processati concorrentemente da attori diversi.

Un programma ad attori termina quando ogni attore creato è inattivo (idle) e gli attori non sono aperti all'ambiente (altrimenti quest'ultimo può mandare nuovi messaggi agli attori in futuro). Bisogna tenere in considerazione che un programma ad attori non ha la necessità di terminare, ad esempio infatti in certi sistemi interattivi oppure sistemi operativi, l'esecuzione può continuare indefinitivamente.

2.4 Le tre primitive fondamentali

Quando un attore è inattivo e ha un messaggio pendente, esso può accettarlo e mettere in atto la computazione che è stata precedentemente definita al suo interno (comportamento). Come risultato di tutto ciò, esso può mettere in atto tre tipi di azioni (primitive):

- **create**: crea un nuovo attore partendo da una descrizione di comportamento ed un set di parametri

- **send to**: spedisce un messaggio ad un attore

- **become**: rimpiazza il comportamento dell'attore con un nuovo comportamento[3]

Queste primitive formano un semplice ma potente insieme sopra quale costruire un ampio range di astrazioni di alto livello e paradigmi per la programmazione concorrente.

La primitiva *create* è per la programmazione concorrente quello che la definizione di lambda abstraction[4] è per la programmazione sequenziale. Essa estende la creazione dinamica di risorse alla programmazione concorrente.

Send è la primitiva di base che permette all'attore che la utilizza, di far si che un messaggio sia posto nella mailbox di un altro attore (coda di messaggi). E' importante sottolineare che il nome dell'attore destinatario del messaggio debba essere necessariamente specificato all'atto di invio del messaggio stesso.

E' interessante notare come in [1] il cambiamento di stato può essere effettuato usando il cosiddetto *rimpiazzamento del comportamento*. Ogni qual volta un attore processa una comunicazione, esso computa anche il suo comportamento in risposta alla prossima comunicazione che riceverà. Il comportamento di rimpiazzamento per un attore puramente funzionale[5] è identico al comportamento originale, mentre in tutti gli altri casi può cambiare. Il cambiamento nel comportamento più rappresentare un semplice cambiamento di una variabile di stato come ad esempio il saldo di un conto bancario, o può essere un cambiamento nelle operazioni che sono eseguite in risposta al messaggio. Quindi parlando in termini puramente object-oriented, si possono modificare sia i campi che metodi di quel particolare oggetto.

[3]Bisogna considerare che su [1] il rimpiazzamento del comportamento può indicare la modifica sia dello stato dell'attore che proprio dei metodi che vengono eseguiti in risposta ad un messaggio. In [4] si parla di *update* dello stato che può modificare il comportamento dell'attore.

[4]La lambda abstraction è un termine usato nel Lambda Calculus per descrivere una funziona anonima ad una variabile.

[5]Si tratta di un attore che non ha uno stato interno ed esegue solo delle funzioni che sono statiche nel tempo.

2.5 Descrizione di sistemi ad attori: Event-diagram

La computazione concorrente può essere visualizzata in termini di event-diagrams (diagrammi ad eventi). Questi diagrammi sono stati sviluppati per modellare il comportamento di un sistema ad attori. In esso possono essere presenti delle linee verticali, chiamate *lifeline* (punto 1 nella figura 2.1) che rappresentano la vita di un attore nel tempo (che scorre dall'alto verso il basso) e lungo le *lifeline* possono accadere dei cosiddetti *eventi*. Se due *lifeline* sono connesse con una linea (tratteggiata o meno) significa che c'è una connessione causale tra gli *eventi*, cioè il primo evento in ordine temporale (quello più in alto sulla *lifeline*) ha causato il secondo (quello più in basso). Gli *eventi* negli *Event-diagram* sono:

- Invio di un messaggio, descritto con un punto sulla *lifeline*.
- Ricezione di un messaggio, descritto con un punto sulla *lifeline* e con la descrizione del messaggio posta dentro a delle parentesi quadre (se sono più argomenti deono essere separati da una virgola) (punto 2 nella figura 2.1).
- Creazione di un attore (lato creante), descritto con un punto sulla *lifeline*.
- Creazione di un attore (lato creato), la *lifeline* inizia con un arco aperto (punto 5 nella figura 2.1).

La comunicazione nell'event diagram viene rappresentata con una linea continua che va dall'evento invio di un messaggio, all'evento ricezione (punto 4 nella figura 2.1), mentre per la creazione di un attore si utilizza la linea tratteggiata (punto 3 nella figura 2.1). Le comunicazioni così dette *pendenti*, rappresentano comunicazioni che sono state spedite ma non sono ancora state ricevute. Esse sono descritte con frecce che non arrivano ad un attore ma ad una nota che specifica il messaggio e il destinatario (punto 6 nella figura 2.1).

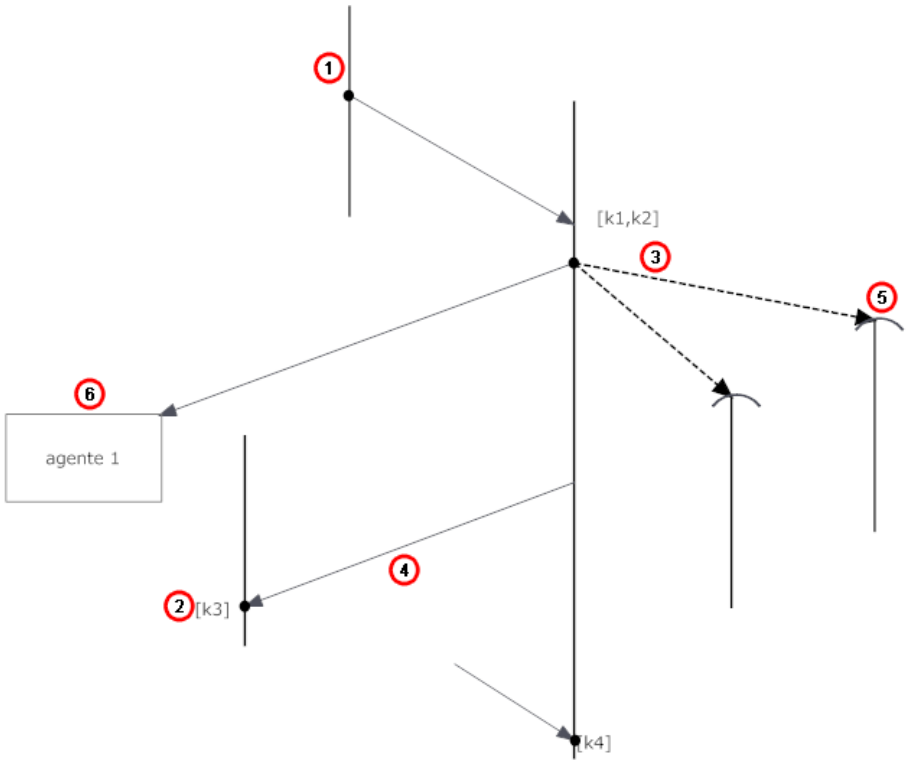

Figura 2.1: Un esempio di Event Diagram

2.6 Vantaggi del modello ad attori

2.6.1 Vantaggi rispetto alla programmazione sequenziale

Nella paradigma della programmazione object-oriented, un oggetto incapsula[6] dati e comportamento. Questo permette di separare l'interfaccia di un oggetto (il cosa) dalla sua rappresentazione/funzionamento (il come). Ciò fa in modo che si possa pensare ai sistemi attraverso un ragionamento modulare che ne permette così il loro sviluppo e la loro evoluzione. Gli attori estendono i vantaggi degli oggetti alla computazione concorrente, dato che sarà sempre possibile distinguere cosa fa un attore da come lo fa. Inoltre essi permettono di **separare il controllo (dove e quando è in esecuzione un attore) dalla logica della computazione**.

In un linguaggio sequenziale, una struttura ricorsiva di controllo (del flusso) è implementata utilizzando uno stack che contiene i vari record di attivazione[7]. In questa maniera non è possibile distribuire il lavoro di computazione oppure processare più di una richiesta, dato che il tutto avviene in sequenza. Utilizzando invece un attore che si preoccupa di portare a termine la funzionalità richiesta, sarà possibile sia **suddividere la sua computazione in modo che le parti eseguibili in parallelo, vengano portate a termine concorrentemente**, inoltre si potranno anche **effettuare all'attore stesso più richieste contemporaneamente**. L'attore che riceve la richiesta di una funzionalità, delega la maggior parte del processamento richiesto

[6]Si definisce incapsulamento (o encapsulation) la tecnica di nascondere il funzionamento interno - deciso in fase di progetto - di una parte di un programma, in modo da proteggere le altre parti del programma dai cambiamenti che si produrrebbero in esse nel caso che questo funzionamento fosse difettoso, oppure si decidesse di implementarlo in modo diverso. Per avere una protezione completa è necessario disporre di una robusta interfaccia che protegga il resto del programma dalla modifica delle funzionalità soggette a più frequenti cambiamenti.

[7]Ogni volta che viene chiamata una funzione, per essa si crea un nuovo record di attivazione nello stack (cioè uno spazio ove sono contenute le variabili locali di quella funzione e i parametri passati), che viene posto immediatamente sopra a quelli già presenti. Quando la funzione termina, il record di attivazione viene cancellato dallo stack stesso, liberando quindi la memoria per i successivi record di attivazione.

a un certo numero di attori, ognuno dei quali è creato dinamicamente. Inoltre il numero di attori creati è direttamente proporzionale alla magnitudo della computazione richiesta. Questo crea un vantaggio molto elevato in termini di tempo risparmiato, se si ha disponibilità di una rete di processori, poichè parte della computazione sarà eseguita in parallelo, distribuendo gli attori sui diversi processori. E' importante sottolineare come problemi descritti attraverso algoritmi ricorsivi, iterativi, risolti tramite il pattern divide and conquer (vedi A.2), etc., possono essere astratti in modo tale da essere eseguiti in computazioni indipendenti.

Un esempio significativo di suddivisione della computazione in un sistema ad attori, può essere quello del calcolo del fattoriale di un numero. Come visto in precedenza, una possibile implementazione potrebbe essere quella di utilizzare un attore capo (chiamato *Fattoriale*, punto 3 nella figura 2.2) che si preoccupa da un lato, di ricevere le richieste di calcolo del fattoriale da diversi consumatori (chiamato *Consumer*, punto 1 nella figura 2.2), e dall'altro di suddividere la computazione (Divide and conquer A.2) delegando le parti ai nuovi attori (m1,m2,m3, m1 è il punto 2 nella figura 2.2) da lui creati, per portarla a termine. In primis, il *Consumer* che necessita del calcolo del fattoriale di un numero n (pari a 3 nell'esempio), si rivolgerà all'attore che svolge questa funzionalità, cioè quello chiamato *Fattoriale*, tramite un messaggio ([3,c] nell'esempio) che ha come argomenti il numero in questione ed un riferimento al *Consumer* stesso in modo tale che il *Fattoriale* possa restituirgli il risultato. Il *Fattoriale* crea immediatamente un nuovo attore-lavoratore che si chiama m1 e che ha come comportamento quello di moltiplicare il numero che gli viene passato all'atto di creazione (nell'esempio pari a 3) per il numero che gli perverrà tramite un messaggio futuro da un altro attore (nell'esempio [2], quindi 3x2). Tutto ciò è descritto nella figura, tramite la notazione *(m1,f(3,c))* (punto 4 nella figura 2.2). Subito dopo aver creato il primo attore-lavoratore, il *Fattoriale*, si manda un auto-messaggio (punto 5 nella figura 2.2) dello stesso tipo di quello ricevuto in precedenza (ma ovviamente con il numero decrementato e il destinatario per la risposta differente) dal *Consumer* per rieseguire quanto scritto sopra (creazione di un nuovo attore e così via). Questi auto-messaggi terminano quando si è raggiunto il valore zero (ultimo punto della lifeline del

Fattoriale). Tutto ciò permette di calcolare in maniera concorrente il fattoriale di uno o più numeri. Naturalmente questa funzione non è molto complessa per cui anche gli attori-lavoratori creati saranno molto semplici, tanto che probabilmente un implementazione sequenziale sarebbe più performante. In generale però la complessità di questi attori-lavoratori, è arbitrariamente complessa.

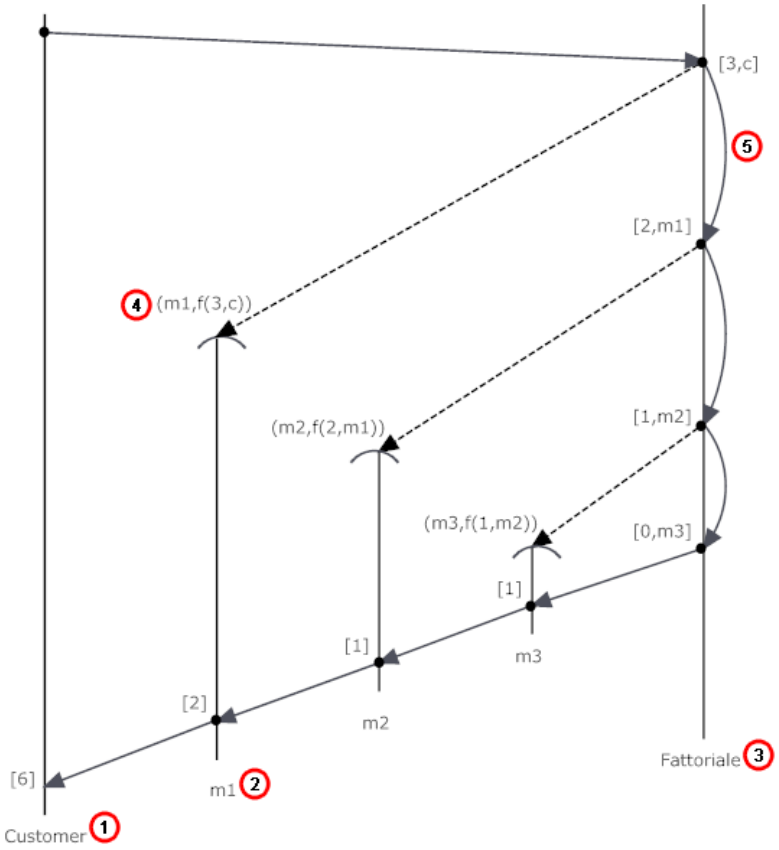

Figura 2.2: Un esempio di suddivisione dei compiti: il calcolo del fattoriale

2.6.2 Vantaggi rispetto ad altri modelli di programmazione concorrente

Un implementazione corretta del modello ad attori garantisce una serie di caratteristiche (vedi 2.9) che facilitano tutto il percorso di creazione di un software di qualità. Queste caratteristiche permettono:

- Accesso safe allo stato tramite interfaccia conosciuta.
- Riduzione del nondeterminismo per facilitare l'analisi di proprietà dell'applicazione.
- Nessuno stato condiviso (quindi nessun problema relativo) neanche durante le comunicazioni.
- La decomposizione dei sistemi in componenti auto-contenuti, autonomi, interattivi e operanti asincronicamente per la modellazione semplice del nondeterminismo presente in sistemi distribuiti, reattivi, mobili e forme di computazioni interattive.
- Scalabilità migliorata.

Tutto ciò è chiaramente in contrasto con il modello a memoria condivisa utilizzato dai *thread* che:

- Non hanno interfaccia.
- Portano ad un nondeterminismo massimo che può essere ridotto solo dal programmatore tramite lock.
- Possono avere stato condiviso e utilizzano risorse condivise.
- Occupano risorse utilizzando lock.
- Hanno Scalabilità problematica (dovuta ai punti precedenti).

2.7 Un primo esempio

Per mostrare come lavora un semplice sistema ad attori, vediamo un piccolo esempio di un framework chiamato *ActorFoundry*.

Listing 2.1: Esempio del linguaggio utilizzato nel framework ActorFoundry

```
public class CiaoActor extends Actor {
        @message
        public void saluta() throws
        RemoteCodeException{
                ActorName other = null;
                send( stdout, "print", "Ciao");
                other = create(
                AiutoActor.class,"Mondo");
                send(other , "scrivi");
        }
}

public class AiutoActor extends Actor {

        // costruttore che inizializza la stringa
        // Parola = secondo argomento passato

        @message
        public void scrivi() throw
        RemoteCodeException{
                send(stdout, "print", Parola);
        }
}
```

Si può notare come siano presenti due classi che estendono la classe Actor, cioè quella che descrive il comportamento generico di un attore. Come già detto in precedenza, i messaggi vengono gestiti dai metodi della classe, nel caso specifico rispettivamente *saluta()* e *scrivi()*, infatti essi sono annotati con l'etichetta *@message*.

Inoltre possiamo osservare la presenza delle primitive viste in precedenza (vedi 2.4), in primis del metodo *create(HelpActor.class,argomento)* che permette ad un' istanza di un attore di tipo *CiaoActor* di creare un' istanza di un attore di tipo *AiutoActor*, passandogli la parola che poi quest'ultimo dovrà scrivere (opzionale). Vediamo anche come la primitiva *create()* restituisca il nome dell'attore creato (vedi 2.2), in modo tale che sia poi possibile in seguito inviargli dei messaggi. In secondo luogo notiamo che è presente anche la primitiva *send*, ad esempio con *send(other, scrivi)*. Essa permette di spedire alla mailbox dell'attore posto nel primo argomento, il messaggio posto nel secondo. Nell'esempio specifico, il secondo argomento sarà proprio il metodo (l'azione) che l'attore ricevente deve eseguire all'arrivo del messaggio (e i successivi argomenti di *send* gli argomenti del metodo da attivare). Occorre ricordare che secondo le specifiche del nostro sistema ad attori, questo tipo di comunicazione è asincrona, infatti una volta chiamato questo metodo, il controllo viene subito ritornato all'attore che non aspetta il suo arrivo a destinazione. Inoltre la rete potrà avere un ritardo indeterminato e l'ordine di arrivo dei messaggi sarà non deterministico, assumendo comunque che il messaggio sarà prima o poi consegnato, attuando una certa forma di *fairness*.

Sempre dall'esempio, osserviamo come un'istanza di *CiaoActor* possa solo ricevere un tipo di messaggio, cioè saluta, che farà scattare il metodo saluta. Questo metodo serve anche come *entry point* (punto di accesso) in luogo del *main* tradizionale.

Quando l'attore *CiaoActor*, riceve il messaggio saluta, esso come prima cosa, spedirà il messaggio print con l'argomento Ciao all'attore *stdout* (un attore built in che rappresenta lo standard output stream). Come risultato di tutto ciò, la stringa Ciao potrà prima o poi essere stampata a standard output. Successivamente viene creata dall'attore *CiaoActor* un'istanza della classe *AiutoActor*. Il primo attore invierà poi un messaggio scrivi all'attore appena creato, in modo tale da delegare un parte della scrittura a quest'ultimo. E' interessante notare come, dato l'asincronismo della comunicazione in questo tipo di sistemi, sia possibile che la parola Mondo venga stampata a standard output prima di Ciao.

2.8 Sincronizzazione

Gli attori come abbiamo già visto in precedenza, fanno uso della messaggistica asincrona che per definizione non permette nessun tipo di sincronizzazione tra mittente e ricevente. Un messaggio viene inviato, e il mittente continua a fare ciò per cui era stato programmato. In seguito, se riceverà una risposta, sarà valutata quando essa sarà la prima della mailbox. Ci sono alcuni casi particolari però, dove questo non è il modello migliore per descrivere le interazioni che avvengono tra attori. Ad esempio nel caso in cui:

- Il mittente debba valutare la risposta al suo messaggio prima di effettuare altre operazioni.
- Occorra processare un messaggio solo se l'attore si trova in un certo stato.
- Un attore non debba eseguire ulteriori operazioni, prima di aver ricevuto un certo numero di messaggi specificati.

Queste sono situazioni descritte da pattern di comunicazione, che rispettivamente ai tre esempi precedenti, prendono il nome di:

- Messaggistica tipo Remote Procedure Call (RPC-like messaging).
- Vincoli di sincronizzazione locali (local synchronization constraints).
- Join continuation.

Tutte queste situazioni non possono essere descritte con il nostro precedente modello ad attori, poichè dopo aver elaborato un messaggio si passa al successivo della mailbox, qualunque esso sia. Occorrerebbe perciò bufferizzare i messaggi momentaneamente non importanti per elaborarli successivamente, lasciando spazio a quelli che in quel momento ci interessano. Per fare ciò si utilizzano particolari primitive od oggetti di prima classe che permettono di realizzare questi pattern, nonostante il modello comunicativo degli attori sia asincrono. Vedremo singolarmente nei tre sottoparagrafi successivi la descrizione accurata di questi pattern e le primitive per realizzarli.

2.8.1 RPC-like Messaging

La comunicazione RPC-like è un pattern comune di message-passing nei programmi ad attori. **Nelle comunicazioni RPC-like, il mittente di un messaggio (richiesta) aspetta che la risposta arrivi dal destinatario, prima di eseguire altre azioni.** Per esempio si consideri l'esempio di un attore cliente che richiede un preventivo per un viaggio. Il cliente dovrebbe aspettare che il preventivo arrivi prima di decidere se comprare il biglietto oppure se richiedere un preventivo ad un' altra agenzia. Come visto in precedenza, l'attuale modello ad attori dovrebbe essere modificato per permettere di bufferizzare i messaggi che non corrispondono alla risposta cercata. In particolare il nuovo comportamento dell'attore sarebbe:

1. Il cliente manda una richiesta.
2. Il cliente controlla i messaggi in arrivo.
3. Se il messaggio arrivato corrisponde alla risposta della sua particolare richiesta, esso porta a termine le azioni appopriate, come ad esempio accettare l'offerta o continuare a cercare.
4. Se il messaggio arrivato non corrisponde alla risposta della sua particolare richiesta, allora il messaggio deve essere manipolato, ad esempio bufferizzandolo per un essere processato più avanti. Il cliente deve continuare a controllare la mailbox per verificare la presenza della risposta che cerca.

Per effettuare questo pattern in *ActorFoundry* si usa la primitiva *SendRPC* che mette in pratica quanto visto sopra.

Listing 2.2: Primitiva SendRPC per effettuare il pattern RPC-like Messaging in ActorFoundry

```
sendRPC(actor,msg)
```

La messaggistica RPC-like è universalmente supportata nei linguaggi ad attori e nelle librerie. Essa è particolarmente utile in due tipi di scenari comuni. Il primo si verifica quando vogliamo mandare

una sequenza ordinata di messaggi a un particolare recipiente e ci si voglia assicurare che il destinatario abbia ricevuto un messaggio prima di mandare il successivo. Una variante del primo scenario si ha quando il mittente manda un messaggio a più destinatari e nel nostro caso si voglia assicurare che un destinatario abbia ricevuto il suo messaggio prima di inviarlo ad un altro. Nel secondo scenario invece, lo stato del mittente dipende dalla risposta del messaggio che ha inviato. Per questo non dovrebbe processare altri tipi di messaggi prima della ricezione della risposta, poichè ad esempio potrebbe dare delle risposte non significative. In questo caso rientra anche quello per cui le successive domande siano dipendenti dalla risposta ricevuta. Un esempio potrebbe essere la comunicazione tra tre attori che vogliono andare al mare a divertirsi: senza RPC-like Messaging ci saranno ben 3 operazioni (due risposte ed una domanda) inutili, mentre questo non accade se si utilizza correttamente la primitiva.

Listing 2.3: Scambio di messaggi asincroni prima di andare al mare senza RPC-like messaging; Da 3 a 5 sono operazioni inutili

```
1        A–>B: è bel tempo?
2        C–>A: andiamo al mare?
3        A–>C: si
4        C–>A: porti tu i teli?
5        A–>C: si
6        B–>A: no sta piovendo
```

Listing 2.4: Scambio di messaggi asincroni prima di andare al mare con RPC-like messaging

```
A–>B: è bel tempo?
(A deve aspettare la risposta prima
di processare altri messaggi)
C–>A: andiamo al mare?
(bufferizzata da A)
B–>A: no sta piovendo
A: processa: andiamo al mare?
A–>C: no
```

Dato che la messaggistica RPC-like è simile alle *procedure call* nei linguaggi sequenziali, i programmatori hanno la tendenza a sovrauti-

lizzarla. Sfortunatamente l'uso inappropriato di questo tipo di pattern può introdurre dipendenze non necessarie nel codice, che possono rendere l'esecuzione della computazione molto più inefficiente di quanto debba essere ed inoltre essa può portare a *livelock*. Il *livelock* si presenta quando l'attore ignora o postpone il processamento di altri messaggi, aspettando una risposta che mai arriverà, (perchè magari anche l'altro attore è bloccato). Un esempio di *livelock*, nella figura 2.3, dove l'attore B aspetta r1 (risposta a m4) continuando a processare messaggi, ma A è bloccato per cui quel messaggio non arriverà mai. B risulterà ancora vivo (da qui *livelock*), ma comunque bloccato nel continuo processamento di messaggi:

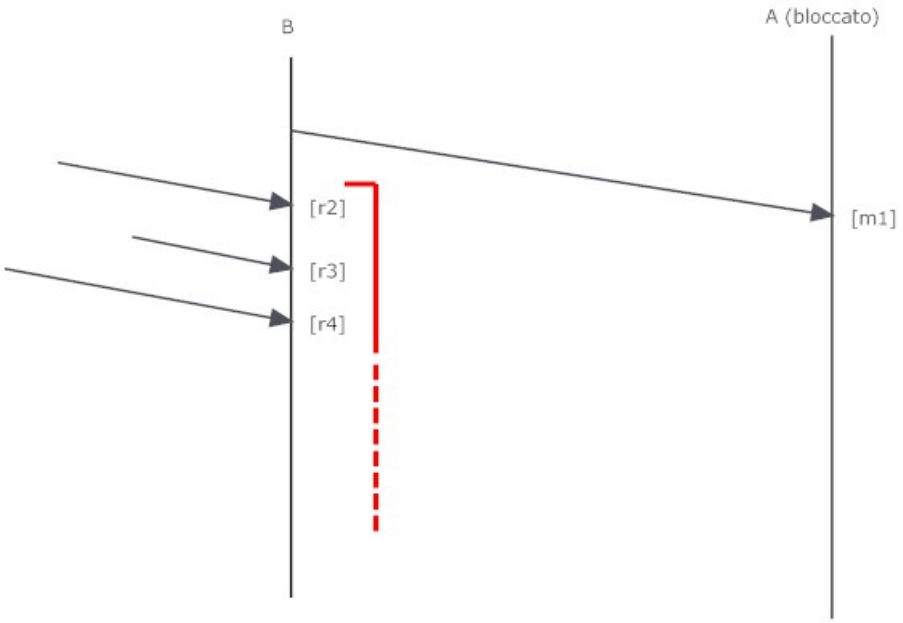

Figura 2.3: Un esempio di Livelock dovuto ad una chiamata RPC

2.8.2 Local synchronization contraints

Come già detto in precedenza, l'asincronismo è inerente ai sistemi distribuiti e mobili. Una implicazione dell'asincornismo è che il numero

dei possibili ordini in cui i messaggi possono arrivare è esponenziale al numero di messaggi pendenti (messaggi inviati ma non ancora ricevuti).**Inoltre in un sistema concorrente in generale, un mittente può essere ignaro dello stato in cui sarà il ricevente nel momento in cui riceverà il messaggio**. Ad esempio supponendo che un attore richieda un lavoro da eseguire ad un attore-capo, potrebbe accadere che quest'ultimo (si trova nello stato in cui) non abbia momentaneamente lavori da assegnargli. Un modo per risolvere questo impedimento sarebbe quello di far rispondere all'attore-capo che la sua richiesta è stata rifiutata. Ora l'attore-lavoratore potrebbe ripetitivamente riformulare la richiesta di lavoro fino a che la risposta non sia positiva. Questa tecnica è chiamata *busy waiting* poichè appunto si aspetta, continuando a generare richieste, fino a quando una non verrà soddisfatta. Purtroppo essa è alquanto dispendiosa, infatti **non permette all'attore lavoratore di eseguire altri lavori durante questa fase ed inoltre genera un traffico di messaggi assolutamente non necessario**.

Un modo per far fronte a questa problematica è quella, a livello del ricevente, di bufferizzare il messaggio che gli è arrivato e di processarlo solo nel momento in cui il suo stato gli permette di farlo. In questo modo il mittente aspetterà la risposta senza mandare in continuazione messaggi di richiesta. Ad esempio, nel caso del capo e del lavoratore visto in precedenza avremo:

Listing 2.5: Esempio di busy waiting; Capo (C) Lavoratore (L); si notano la serie di messaggi inutili infatti sarebbe bastata una sola domanda ed una sola risposta positiva

```
L->C: hai un lavoro per me?
C->L: No
L->C: hai un lavoro per me?
C->L: No
X->C: nuovo lavoro
L->C: hai un lavoro per me?
C->L: Si
```

Listing 2.6: Esempio di utilizzo di LSC; Capo (C) Lavoratore (L); nessun messaggio inutile presente

```
L->C: hai un lavoro per me?
Stato C : nessun lavoro -> bufferizza
la domanda precedente
...
...
(nuovo lavoro trovato dal C)
X->C: nuovo lavoro
C->L: si
```

Se i messaggi pendenti sono bufferizzati esplicitamente, come in questo caso, dentro al corpo di un attore, si mischia il modo in cui (il come) il messaggio sarà processato con il momento in cui (il quando) sarà processato. Questo va contro i principi del software sulla *separation of concerns*[8] Per questo motivo alcuni ricercatori hanno proposto diversi costrutti per permettere ai programmatori di specificare il corretto ordinamento dei messaggi in maniera modulare ed astratta, come ad esempio le formule logiche (predicati) sullo stato degli attori e sul tipo di messaggi. Molti linguaggi ad attori e diversi framework forniscono tali costrutti come ad esempio il pattern matching in *Erlang*, oppure la libreria *Actors* di *Scala*.

2.8.3 Join continuation

Se utilizziamo il pattern *Divide and conquer* (vedi A.2), ci potremmo trovare di fronte ad una situazione in cui, un attore-capo (punto 1 nella figura 2.4) debba collezionare, elaborare e restituire (punto 7 nella figura 2.4) i vari risultati dei lavori di tutti gli attori-operai (punto 2 nella figura 2.4) che dirige. Questo implica che prima di eseguire altri tipi di operazioni debba aver ricevuto tutti i messaggi in questione (punti 4,5,6 nella figura 2.4). Questo è possibile tramite l'utilizzo di una *Join continuation* (punto 3 nella figura 2.4) cioè una entità che porta a termine questo tipo di sicronizzazione. Si parla appunto di

[8]La SOC dichiara che gli elementi di un sistema devono essere esclusivi negli scopi che devono portare a termine. Questo significa che nessun elemento dovrebbe condividere responsabilità di un altro oppure accollarsi responsabilità di altri elementi. La SOC è raggiunta stabilendo dei limiti, cioè dei vincoli logici o fisici che delineano le responsabilità [11].

entità, poichè a seconda del linguaggio ad attori una *Join continuation* può essere realizzata tramite un attore oppure tramite una primitiva.

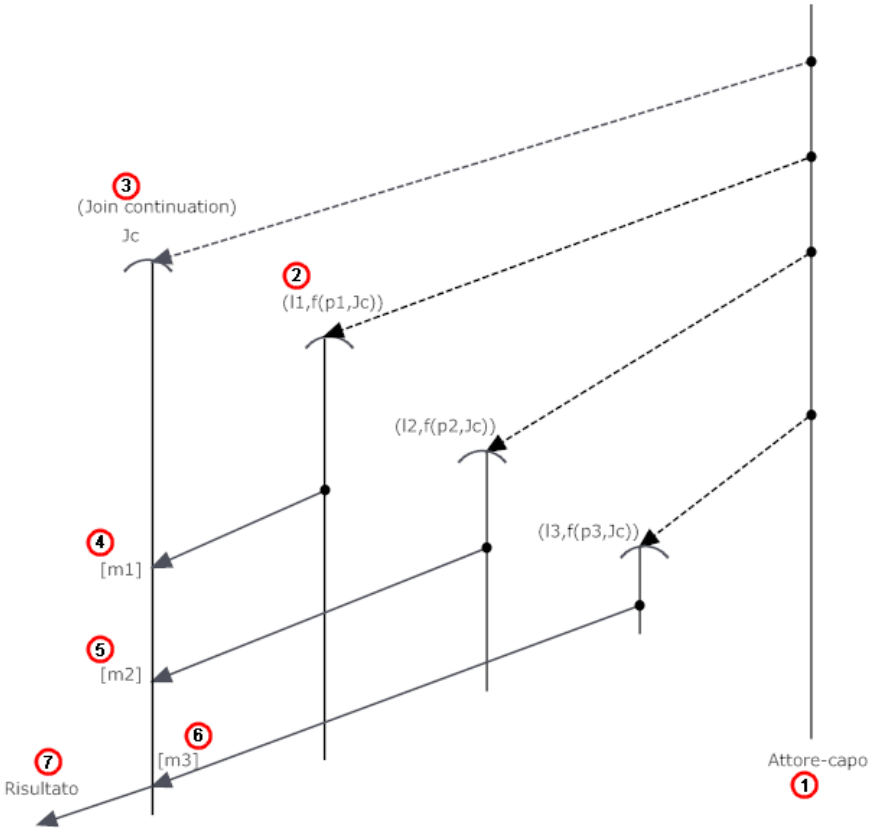

Figura 2.4: Un esempio di Join Continuation

2.9 Proprietà semantiche

Come menzionato in precedenza alcune delle proprietà semantiche fondamentali degli attori sono (vedi 2.6 e 2.2):

- *Encapsulation* (*incapsulamento*).
- *Esecuzione atomica dei metodi* (*macro-step semantics*).

- *Fairness.*
- *Location trasparency.*
- *Transparent migration* (trasparenza nella migrazione) .

Discuteremo nei sotto paragrafi successivi le implicazioni di tutte queste proprietà. Occorre però notare che non tutti i linguaggi ed i framework ad attori fanno valere queste caratteristiche, spesso infatti, per arrivare ad un implementazione semplice se ne trascurano alcune. E' possibile comunque, tramite trasformazioni sofisticate, compilazioni e ottimizzazioni runtime, guadagnare pressochè tutta l'efficienza persa con l'utilizzo di implementazioni di linguaggio semplici. Evitando di soddisfare queste proprietà in un linguaggio ad attori o in una implementazione di un framework, si aggiungono degli oneri ai programmatori che dovranno poi assicurarsi che il loro codice supporti queste caratteristiche.

2.9.1 Incapsulamento ed esecuzione atomica delle operazioni

L'*incapsulamento* (vedi 2.6) è uno dei principi base della programmazione object oriented. Esso permette nei linguaggi sequenziali che supportano questo principio di facilitare il ragionamento su alcune proprietà di safety del sistema, come:

- *Memory safety.*
- *Data race freedom.*
- *Safe modification of object state.*

Nei linguaggi object oriented si ha un cambiamento atomico dello stato degli oggetti (*safe modification of object state*): un oggetto invoca (manda un messaggio a) un altro oggetto, il quale finisce di processare il messaggio prima di accettarne un altro da un altro oggetto. Questo ci permette di ragionare sul comportamento dell'oggetto in risposta al messaggio, dato lo stato dell'oggetto destinatario prima

e dopo del processamento del messaggio. Se un secondo messaggio potesse interrompere l'oggetto destinatario e modificarne lo stato mentre esso sta ancora processando il primo messaggio, non sarebbe più possibile fare ragionamenti sul suo comportamento, dato che lo stato finale non dipenderebbe più solamente dal primo messaggio. Non solo: l'interleaving di messaggi potrebbe portare a stati erronei ed inconsistenti (*data race*). Un esempio di *memory safety* potrebbe venire dal linguaggio Java che nasconde i puntatori alla memoria dietro riferimenti agli oggetti, non permettendo quindi operazioni aritmetiche sui puntatori. Inoltre la *memory safety* fa in modo che si acceda allo stato di un oggetto solo attraverso interfacce predefinite.

L'incapsulamento nel contesto del modello ad attori si ottiene tramite due differenti requisiti: *state encapsulation* e *safe messaging*. Entrambi fanno comunque riferimento al fatto che non ci deve essere memoria condivisa tra due attori.

Lo *State encapsulation* è una proprietà che si verifica quando lo stato degli attori è interno (non condiviso) e modificabile da altri attori solo tramite l'utilizzo di messaggi, quindi in un'unica e definita maniera. Questo significa che la memoria di un attore che fa riferimento al suo stato non è condivisa con un altro attore ed è leggibile e modificabile solo tramite accessi controllati.

Il *Self mesaging* è simile allo *State encapsulation*, poichè si riferisce sempre alla possibilità che due attori agiscano su memoria condivisa, ma in questo caso non quella che descrive lo stato, ma quella passata in un messaggio. Infatti durante l'invio di messaggi è possibile che si debba passare ad un altro attore un oggetto complesso ma anche una semplice stringa. Questo può essere effettuato per valore o per riferimento. Nel caso si passasse il riferimento, due attori potrebbero accedere alla stessa sezione di memoria (memoria condivisa). Il passaggio deve essere dunque effettuato per valore e quindi si deve necessariamente effettuare una copia della parte di memoria che deve essere passata, con probabile decadimento delle prestazioni.

L'*esecuzione atomica delle operazioni* consiste invece nel portare a termine tutte le azioni in risposta ad un dato messaggio prima di processarne uno nuovo.

2.9.2 Vantaggi dell'incapsulamento e dell'esecuzione atomica delle operazioni

L'incapsulamento dello stato con modifica-lettura tramite metodi o operazioni, come nei linguaggi object oriented, fa si che l'accesso ai dati interni avvenga in maniera *safe* tramite interfaccia predefinita (i messaggi) e quindi siano, in un certo senso, controllati.

L'esecuzione atomica delle operazioni riduce drammaticamente il nondeterminismo (dato che l'interleaving sarà molto minore), fornendo inoltre dei macro step semantici che semplificano il ragionamento sul sistema ad attori stesso. Inoltre viene anche ridotto significativamente l'esplorazione dello spazio degli stati, ad esempio per controllare una proprietà nell'esecuzione potenziale di un sistema ad attori.

L'incapsulamento dello stato, associato all'esecuzione atomica dei metodi, permette di evitare gli annosi problemi derivanti dallo stato condiviso (vedi 1.6.4). Infatti nel caso lo stato fosse interno e modificabile solo tramite messaggi e non ci fosse il supporto per l'esecuzione atomica delle operazioni, si potrebbe creare un interleaving con data races derivante dall'esecuzione parallela di più operazioni contemporanee. Senza incapsulamento dello stato risulterebbe anche inutile parlare di esecuzione atomica delle operazioni, infatti sarebbe possibile una modifica dello stato pur senza l'utilizzo dei messaggi (e quindi senza operazioni), non avrebbe senso quindi rendere atomico un tipo di accesso se poi esistono altre modalità per modificare i dati.

2.9.3 Fairness

Il modello ad attori assume per vera la nozione di *fairness* che afferma che ogni attore farà dei progressi (sarà eseguito da qualche parte) se ha della computazione da portare a termine (non ci sarà starvation) ed ogni messaggio sarà prima o poi consegnato all'attore destinatario, a meno che quest'ultimo sia permanentemente disabilitato (in un loop infinito ad esempio oppure bloccato nell'esecuzione di un operazione non possibile). La *fairness* ci permette di verificare le proprietà di *liveness*[9] degli attori facenti parte il sistema. Se non ci fosse *fairness*

[9]La proprietà di liveness, se verificata afferma che il programma (o l'attore in questo caso) prima o poi entrerà nello stato desiderato specificato nella proprietà

potrebbe accadere che alcuni attori non vengano mai eseguiti e quindi sarebbe impossibile verificare che questi entrino prima o poi in determinati stati (*liveness*). Per esempio, se un sistema è composto da un gruppo di attori A e da un altro di attori B che sono perennemente occupati, tutto ciò non influisce sul progresso degli attori A, se è verificata la proprietà di *fairness*. Un esempio reale di *fairness* si ha nei browser, dove i problemi sono spesso causati dalla composizione dei componenti del browser stesso, con plug-in di terze parti. In assenza di *fairness*, tali plug-in portano a volte il browser in crash o in blocco. E' interessante notare che se un attore non è mai schedulato, i messaggi pendenti a lui indirizzati non saranno mai consegnati, per cui se un sistema è formato da entità cooperative, potrebbe accadere che tutta la computazione venga rallentata o addirittura bloccata a causa della sua mancata esecuzione. Nel codice sottostante si può vedere come possa accadere una situazione del genere. E' presente infatti un attore che richiede il calcolo di una somma ad un altro attore (calcolatore) e rimane in busy waiting in attesa di una risposta contenente la somma. Se il calcolatore non venisse mai schedulato, il primo attore finirebbe per andare in starvation. Il problema si può verificare non solo in sistemi con attori cooperativi (come appena visto), ma anche in sistemi con attori non cooperativi, in quanto un attore può comunque occupare la risorsa computazionale su cui è eseguito, facendo morire di stenti gli altri attori che aspettano di essere eseguiti.

Listing 2.7: Un esempio di sistema scritto in Scala Actors che mostra un attore fermo in busy waiting in attesa di una risposta. In caso di violazione della proprietà di fairness se l'attore che deve fornire la risposta non viene schedulato l'attore descritto nel codice finirà per morire di stenti.

```
import scala.actors.Actor
import scala.actors.Actor._

Object fairness{
        class FairActor() extends Actor {

                ...
```

stessa.

```
                def act(){ loop{ react{
                        case (v : int) => {
                                // se arriva
                                // il risultato
                                // inseriscilo
                                // in una
                                // variabile
                                data = v
                        }
                        case ("wait") => {
                                // sezione
                                // busy-waiting
                                if(data > 0)
                                println(data)
                                else self ! "wait"
                                // invia un
                                // messaggio
                                // a se stesso
                                // per rimanere
                                // in busy-wait
                        }

                        case ("start") => {
                                calc ! ("add",
                                 4, 5)
                                self ! "wait"
                        }
                }}}
        }


}
```

2.9.4 Location Trasparency

Gli attori comunicano scambiando messaggi con altri attori, che potrebbero essere sullo stesso core, sulla stessa cpu o su un altro nodo della rete. Per fare questo è necessario che all'atto di creazione dell'attore, gli venga assegnato un nome univoco su tutto il sistema distribuito. Un assegnamento di nomi si dice location trasparent, quando non dipende dalla locazione dove è eseguito o creato l'attore e quindi fornisce un'astrazione per i programmatori, abilitandoli a programmare senza preoccuparsi dell'attuale posizione fisica degli attori (o dell'attore considerato).

2.9.5 Mobilità

La *mobilità* è definita come l'abilità di una computazione di migrare tra nodi differenti. La mobilità detta *strong* è definita come l'abilità di un sistema di supportare il movimento sia del codice che dello stato esecutivo, quella *weak* invece permette solo lo spostamento del codice (e dello stato iniziale che può essere anch'esso trasferibile). Nei sistemi ad attori la mobilità *weak* può essere utile per spostare un attore in uno stato *idle*, ad esempio un attore bloccato a causa della mailbox vuota, visto che è come se fosse nel suo stato di partenza (non essendo in esecuzione nessuna operazione). La mobilità *strong* significa che è possibile migrare un attore mentre sta eseguendo un operazione, quindi durante il processamento di un messaggio. Il modello ad attori se ben implementato dovrebbe possedere la proprietà di incapsulamento che permette facilmente la migrazione dato che il solo attore agisce sui suoi dati (ad esempio se possedesse un riferimento ad una zona di memoria condivisa, la zona verrebbe spostata? Come dare i riferimenti nuovi a tutti coloro che li condividono?). Inoltre anche il fatto che venga processato solo un messaggio alla volta (esecuzione atomica delle operazioni) permette di eseguire più facilmente la migrazione dato che in un certo istante si può conoscere quale è l'unica istruzione (se l'attore non è *idle*) in esecuzione.

La mobilità è importante per il *load-balancing*[10], *fault-tolerance*[11] e la rapida riconfigurazione di un sistema (basti pensare all'arrivo nella rete di nodi più performanti, basterà solamente a runtime, spostare gli attori dai vecchi nodi ai nuovi). In particolare la *mobilità* è utile nel raggiungimento di performance scalabili soprattutto in sistemi dinamici. Essa può essere utile anche nella riduzione dell'energia consumata dall'esecuzione in parallelo delle applicazioni. Parti differenti di un'applicazione infatti possono essere portate a termine da algoritmi paralleli diversi e il consumo di energia di un algoritmo dipende dal numero e dalla frequenza dei core su cui questi algoritmi verranno eseguiti. La *mobilità* facilita la ridistribuzione dinamica di una computazione parallela al numero appropriato di core, che ad esempio minimizza l'energia consumata per una data performance richiesta.

[10]Il Load Balancing è una tecnica che consiste nel distribuire il carico di uno specifico servizio, ad esempio la fornitura di un sito web, tra più server. Si aumentano in questo modo la scalabilità, l'affidabilità e le performance dell'architettura nel suo complesso.Ad esempio se arrivano 10 richieste per una pagina web su un cluster di 3 server, alle prime 3 risponderà il primo server, a 3 il secondo ed alle ultime 4 il terzo.La scalabilità deriva dal fatto che, nel caso sia necessario, si possono aggiungere nuovi server al cluster, mentre la maggiore affidabilità deriva dal fatto che la rottura di uno dei server non compromette la fornitura del servizio; non a caso i sistemi di load balancing in genere integrano dei sistemi di monitoraggio che escludono automaticamente dal cluster i server non raggiungibili ed evitano in questo modo di far fallire una porzione delle richieste degli utenti. Per quanto riguarda le performance, esse possono essere migliorate impiegando diverse distribuzioni di carico nei differenti stage di computazione. Oppure le performance ottime o corrette potrebbero dipendere dalle condizioni runtime, come il carico istantaneo di lavoro o le caratteristiche di sicurezza di diversi nodi, per questo il Load Balancing permetterebbe di essere di fondamentale importanza in questo senso. Ad esempio le applicazioni web potrebbero migrare dai server ai client mobili dipendentemente dalle condizioni della rete oppure dalle capacità del client.

[11]La tolleranza ai guasti (o fault-tolerance, dall'inglese) è la capacità di un sistema di non subire fallimenti (cioè intuitivamente interruzioni di servizio) anche in presenza di guasti. Essa è uno degli aspetti che costituiscono l'affidabilità. È importante notare che la fault-tolerance non garantisce l'immunità da tutti i guasti, ma solo che i guasti per cui è stata progettata una protezione non causino fallimenti.Inoltre può portare al peggioramento di altre prestazioni, per cui nella progettazione di un sistema è necessario trovare adeguate ottimizzazioni e compromessi.

2.10 Implementazioni

Erlang è molto probabilmente l'implementazione più conosciuta del modello ad attori. Esso è stato sviluppato nei laboratori *Ericsson* circa vent'anni fa, mentre oggi possiamo trovare delle implementazioni che si concentrano su un particolare dominio, come ad esempio *SALSA* per Internet, *Erlang and E* per le applicazioni distribuite, *ActorNet* per le reti di sensori e più recentemente *Scala Actors library* e *ActorFoundry* per processori multi-core, più tanti altri ancora in sviluppo.

2.10.1 SALSA

SALSA è stato introdotto da C. Hewitt ('77), poi rifinito e sviluppato da G. Agha ('85-oggi) [8]. In *SALSA* ogni attore incapsula un thread (per l'esecuzione dell'attore stesso), una collezione di oggetti (che contengono i dati e lo stato dell'attore) ed una mailbox (per lo scambio di messaggi). Gli attori di *SALSA* presentano tutte le caratteristiche viste nel paragrafo 2.2, e quindi solo il thread dell'attore può accedere ai suoi oggetti direttamente e cambiarne lo stato. La rete, in una sua visone globale è caratterizzata da un enorme numero di processori, un insieme di nodi non completamente affidabili e da un' infrastruttura comunicativa che rispetto alla velocità dei processori risulta estremamente lenta. Per questi motivi, se si vuole creare una piattaforma di computazione sopra questo livello è necessario che sia presente un middleware[12] che:

- Permetta un protocollo (affidabile) di messaggistica per far fronte all'inaffidabilità della rete in generale.
- Fornisca un supporto per il naming che permetta così al sistema di essere *location trasparent* (vedi 2.9.4).
- Fornisca un supporto per la migrazione degli attori (vedi sempre 2.9.4).

[12]Il middleware è un software che consiste di un insieme di servizi che permettono a più entità (processi, oggetti ecc.), residenti su uno o più elaboratori, di interagire attraverso una rete di interconnessione a dispetto di differenze nei protocolli di comunicazione, architetture dei sistemi locali, sistemi operativi, ecc..

Per quanto riguarda il protocollo di messaggistica, viene utilizzato *RMSP* cioè *Remote Message Sending Protocol* e sempre grazie a quest'ultimo si effettua la migrazione degli attori. E' importante sottolineare che quando un attore migra, cambia la sua locazione ma non il suo nome. La locazione (cioè il luogo dove gli attori vengono eseguiti) viene fornita grazie ad un entità facente parte del middleware che prende il nome di *Theater*. Essa è formata dalla coppia indirizzo ip e porta dove risiede il *Theater* ed eventualmente dal nome della locazione:

Listing 2.8: Esempio di locazione in *SALSA*

```
rmsp://indirizzo_ip_theater:porta/nome_locazione
```

I *Theater* su richiesta possono fornire degli attori non mobili che prendono il nome di *Enviromental Actor* (attori ambientali). Essi portano a termine azioni specifiche come ad esempio la comunicazione su standard input e su standard output. Per quanto riguarda il supporto al naming, vengono utilizzati i cosiddetti *Universal Naming* (UN), cioè dei nomi universali, facilmente leggibili dall'uomo, appunto per permettere la *location trasparency*. Esistono all'interno del sistema, uno o più server, chiamati *Universal Actor Naming Server* (UAN Server) che contengono le coppie formata dall' *Universal Naming* e dalla locazione attuale di quel particolare attore in quell'istante. In questa maniera:

Listing 2.9: Tabella di uno UAN Server in *SALSA*

```
locazione1  universal_naming_actor1
locazione2  universal_naming_actor2
locazione3  universal_naming_actor3
   ...              ....
```

Quando si crea un attore, si deve esplicitamente dichiarare il suo *Universal Actor Naming* (UAN) cioè la coppia formata da UAN Server e UN. Questa dichiarazione fa si che il nome dell'attore (ora detto UAN) sia univoca in tutto il sistema, dato che sia l'UN che il suo UAN Server non cambieranno mai.

Listing 2.10: Esempio di UAN in *SALSA*

```
uan://indirizzo_ip_UANServer:porta/
universal_naming_attore
```

Grazie agli *Universal Actor Naming Server* è possibile realizzare il cosiddetto *Universal Actor*, cioè un estensione del modello ad attori, tale che ogni attore abbia un nome univoco e un luogo dove risiede unico in un certo istante. Inoltre queste informazioni sono raggiungibili e modificabili tramite gli UAN Server, infatti modificando le tabelle contenute al loro interno è possibile cambiare la locazione degli attori e quindi si può effettuare la cosiddetta migrazione, muovendo l'attore da un *Theater* ad un altro.

Per quanto riguarda la sincronizzazione (vedi 2.8), *SALSA* permette di realizzare una semplice implementazione di *RPC-like Messaging*, utilizzando il cosiddetto *Token Passing Continuation*. In pratica quando viene inviato un messaggio ad un attore, si aspetta la sua risposta prima di continuare ad eseguire altre azioni ed inoltre è possibile utilizzare questa risposta come argomento nelle successive computazioni. Di seguito un esempio, dove si aspetta la risposta al messaggio m1 inviato da a1 prima di spedire il messaggio m2 (l'argomento token è la risposta precedente) ad a2:

Listing 2.11: Esempio di Token Passing Continuation in *SALSA*

```
a1<-m1() @ a2<-m2( token );
```

In *SALSA* è possibile anche utilizzare le *Join continuation*, cioè dei costrutti (vedi 2.8.3) che estendono il concetto di *Token Passing Continuation* a più attori. Mentre con una *Token Passing Continuation*, si manda un messaggio e si aspetta una risposta, in una *Join continuation* si mandano più messaggi e si aspetta la risposta di tutti questi messaggi prima di passare all'azione successiva (sincronizzazione di un insieme di messaggi). Nell'esempio successivo si vede come vengano creati quattro attori differenti e tramite la primitiva *join* si aspetti che tutti abbiano risposto al messaggio *find(phrase)* prima di mandare al *resultActor* le risposte.

Listing 2.12: Esempio di Join continuation in *SALSA*

```
Actor[] actors = { searcher0, searcher1,
        searcher2, searcher3 };
```

```
join( actors<-find( phrase ) ) @
     resultActor<-output( token );
```

Questo tipo di costrutto permette la realizzazione del pattern *Divide and conquer* (vedi A.2) e quindi la creazione di particolari sistemi di attori, dove uno di questi, come se fosse un coordinatore o un capo-attore, dirige gli altri (lavoratori) e ne mette insieme i risultati. E' interessante notare che a differenza di quanto visto nell'esempio del paragrafo 2.8.3, dove la *Join continuation* era proprio costituita da un attore a se stante, differente dal capo-attore, mentre in *SALSA* viene utilizzata una primitiva per crearla, anche quindi all'interno del capo-attore stesso.

2.11 Framework per linguaggi ad attori

In questa sezione discuteremo se le quattro proprietà semantiche fondamentali dei sistemi ad attori (viste nel particolare in 2.9) siano soddisfatte o meno nei framework ad attori più comuni che sono:

- *Kilim.*
- *Scala Actor.*
- *JavaAct.*
- *Jetlang.*
- *ActorFoundry*
- *SALSA*
- *Actor Architecture*

2.11.1 Incapsulamento: state encapsulation

L'incapsulamento dello stato nonostante sia una delle proprietà più importanti di un linguaggio ad attori, in alcuni framework come *Kilim* e *Scala Actor* non viene correttamente rispettato. Ad esempio si

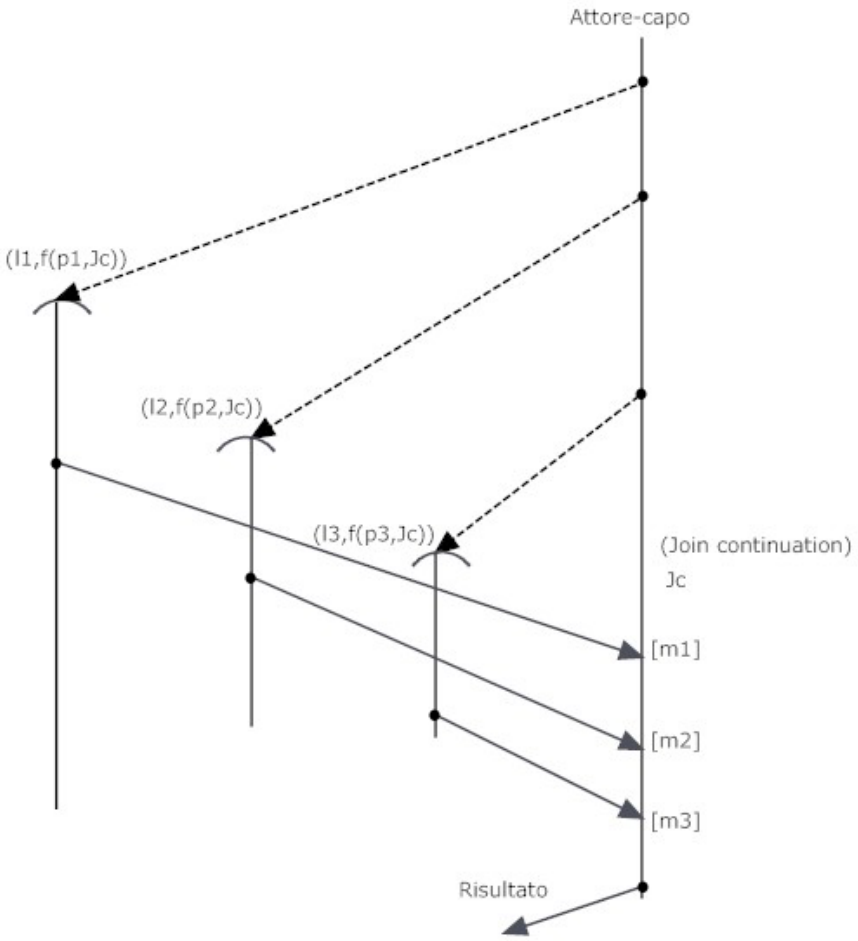

Figura 2.5: Un esempio di Join Continuation in *SALSA*

può notare come nel listato successivo, in *Scala Actor*, vi sia la possibilità di far eseguire un operazione ad un attore, sia tramite invio di messaggio che tramite chiamata di metodo. Come già ribadito in 2.9.1, la proprietà di *state incapsulation* è verificata solo quando l'accesso allo stato di un attore è effettuato tramite l'utilizzo di operazioni richiamate da messaggi e non tramite altre modalità. Potrebbe accadere infatti che le due modalità vengano eseguite concorrentemente oppure tramite interleaving e questo si tradurrebbe in data race.

Listing 2.13: Un esempio di sistema scritto in Scala Actors che mostra la violazione della proprietà di state encapsulation

```
import scala.actors.Actor
import scala.actors.Actor._

Object semaphore{
        class SemaphoreActor()
        extends Actor {

                ...

                def enter(){
                 // critical section
                }
        }

        def main(){
                var gate = new
                SemaphoreActor()
                gate.start
                gate ! "enter"
                // invio del messaggio
                // "enter"
                gate.enter
                // chiamata al
                // metodo "enter"
        }
}
```

Inoltre in *Kilim* gli attori possono avere i riferimenti ad altre mailbox di altri attori. Ancora una volta, questo viola la proprietà di incapsulamento, poichè due attori non possono agire sulla stessa memoria (memoria condivisa).

2.11.2 Incapsulamento: safe messaging

In alcuni framework come *Kilim*, *Scala Actors*, *JavaAct* e *Jetlang*, i messaggi possono contenere riferimenti a parti di memoria. Questo, come già visto nel capitolo 2.9.1 introduce memoria condivisa tra attori e viola la proprietà di *safe messaging*. Tutto ciò sarebbe problematico, a meno che gli oggetti passati non siano di tipo immutabile[13], infatti in questa maniera ci sarebbe un accesso condiviso solo in lettura, evitando possibili data race.

2.11.3 Fairness

Scala Actors, *ActorFoundry*, *SALSA* e *Actor Architecture* sono tutti framework che assicurano *fairness* nello scheduling ma questa garanzia è limitata dai vincoli della JVM su cui gira il framework e dalla piattaforma sottostante.

2.11.4 Location trasparency

La *Location trasparency* è supportata in *SALSA*, *Actor Architecture*, *ActorFoundry*, *JavaAct*, *Jetlang* mentre in *Scala Actors* e *Kilim* non c'è supporto. In *Scala Actors* un nome di un attore è un riferimento alla memoria dell'oggetto che rappresenta l'attore e in *Kilim* un nome di un attore è un riferimento alla sua mailbox. Questi riferimenti sono locali e non possono essere utilizzati al di fuori della JVM dove è in esecuzione il framework (se si utilizzeranno si riferiranno a attori differenti). In *SALSA*, ad esempio, come visto nel capitolo 2.10.1 quando si crea un attore, si deve esplicitamente dichiarare il suo *Universal Actor Naming* (UAN) cioè la coppia formata da UAN Server e UN. Questa dichiarazione fa in modo che il nome dell'attore (ora detto

[13]Un oggetto immutabile è un oggetto il cui stato non può essere modificato dopo che è stato creato.

UAN) sia univoca in tutto il sistema distribuito, dato che sia l'UN che il suo UAN Server non cambieranno mai. Quindi sarà possibile utilizzare ed eseguire l'attore contrassegnato dall'UAN specifico su qualsiasi macchina in rete. Per rendere *location trasparent*, framework che non lo sono nativamente, occorre utilizzare degli add-on come ad esempio *Terracotta* per *Scala Actors*.

2.11.5 Mobilità

La mobilità *weak* è supportata da *SALSA*, *Actor Architecture*, *JavaAct* e *ActorFoundry* mentre la mobilità *strong* è fornita da *ActorFoundry* che permette di catturare il contesto corrente di esecuzione, detto *continuation*.

Capitolo 3

Dart

3.1 Descrizione

Dart (chiamato originariamente *Dash*) è in linguaggio class-based [5] sviluppato da *Google*, svelato alla conferenza *GOTO* di Aarhus il 10-12 Ottobre 2011. Il goal di *Dart* è quello di rimpiazzare *JavaScript* come lingua franca dello sviluppo di applicazioni web.

Figura 3.1: Il logo di Dart dove si può notare lo slogan programmazione web strutturata

Dart intende risolvere i problemi di *JavaScript* che non possono essere affrontati solo evolvendo il linguaggio, mentre cerca di offrire migliori performance, più tool per progetti a larga scala e migliori caratteristiche di sicurezza.

Gli sviluppatori web infatti, si trovano spesso ad affrontare le stesse difficoltà come:

- L'utilizzo di piccoli script che spesso evolvono in grandi applicazioni senza un'apparente struttura. Queste ultime diventano

difficili da debuggare e mantenere, inoltre data la loro monoliticità queste applicazioni non possono essere suddivise e così non si può lavorare indipendentemente con diversi programmatori sulle differenti parti che dovrebbero comporre il sistema.Per questo motivo è difficile essere produttivi quando un applicazione cerca di diventare grande.

- La lettura e il mantenimento di codice altrui. I linguaggi di scripting sono popolari a causa della loro natura leggera che permette di scrivere codice velocemente. Generalmente si utilizza più energia nel commentare il programma piuttosto che nel realizzare una buona struttura, questo rende difficile la lettura ed il mantenimento del codice scritto da altre persone.

- L'impossibilità di creare sistemi omogenei che comprendano sia la parte client che la parte server, eccetto per pochi casi come *Node.js* e *Google Web Toolkit (GWT)*.

- L'uso di differenti linguaggi e formati che comportano context-switch ingombranti e aggiungono complessità al processo di scrittura del codice.

Per questo motivo si è cercato di creare un linguaggio:

- Che permettesse la creazione di sistemi web strutturati e flessibili oltre che efficienti e scalabili.

- Che fosse familiare, naturale da scrivere e facile da imparare.

- Che permettesse tramite tutti i suoi costrutti di creare rapidamente applicazioni performanti.

- Che fosse appropriato per tutto il range di dispositivi che interagiscono con il web come telefoni, tablet, portatili e server.

- A cui allegato ci fossero diversi tool per permettere di essere eseguito velocemente su tutti i maggiori browser.

3.2 Caratteristiche fondamentali

Le caratteristiche fondamentali di *Dart* sono:

- Classi (ed interfacce).
- Tipizzazione opzionale.
- Librerie.
- Tool.

Le classi e le interfacce abilitano l'incapsulamento (vedi 2.6) e il riuso, inoltre forniscono un meccanismo per definire efficacemente delle API.

I programmatori *Dart* possono aggiungere tipi statici al loro codice, dipendentemente dalle loro preferenze (o da quelle del sistema) oppure anche dallo stage di sviluppo dell'applicazione. Il codice quindi può migrare da un semplice e sperimentale prototipo non tipato ad uno complesso e modulare con tipizzazione.

Gli sviluppatori possono creare delle librerie che garantiscono la loro staticità runtime, inoltre è possibile sviluppare pezzi di codice che fanno affidamento a librerie condivise.

Dart includerà un ricco set di ambienti e tool di sviluppo costruiti per supportare il linguaggio. Questi tool abiliteranno uno sviluppo produttivo e dinamico, che include l'*edit-and-continue debugging*[1]. Ad esempio sarà possibile scrivere una bozza del sistema e mentre essa è in esecuzione, riempire e specificare le parti mancanti o quelle lasciate in precedenza generiche.

[1]L'edit-and-continue debugging è una caratteristica che permette di risparmiare molto tempo nella fase di debugging del programma. Essa abilita i cambiamenti al codice mentre ci troviamo in *break mode* e nel momento che riesumiamo l'esecuzione scegliendo comandi come *continue* e *step*, l'edit and continue applicherà automaticamente i cambiamenti (sempre con qualche limitazione). Questo permette di fare dei cambiamenti al codice durante una sessione di debugging, invece di dover fermare, ricompilare l'intero programma e far ripartire la sessione [12].

3.3 Semplici esempi di codice

Vediamo ora dei semplici esempi di codice che ci permetteranno di capire meglio le caratteristiche di *Dart* viste nel paragrafo 3.2.

3.3.1 Classi ed interfacce

Listing 3.1: Esempio di codice Dart

```
interface Forma {
  num perimetro();
}

class Rettangolo implements Forma {
  final num altezza, larghezza;
  // Sintassi per un costruttore "compatto"
  Rettangolo(num this.altezza, num this.larghezza);
  // Sintassi per una funzione "short"
  num perimetro() => 2*altezza + 2*larghezza;
}

class Quadrato extends Rettangolo {
  Quadrato(num lato) : super(lato, lato);
}
```

In questo esempio muoviamo i primi passi nel codice, notando l'utilizzo delle classi e delle interfacce come in un normale linguaggio ad oggetti. Grazie a queste ultime abbiamo un set di blocchi riusabili ed estensibili con cui costruire i nostri sistemi. In *Dart* una classe può implementare interfacce multiple ma ereditare solo da una singola superclasse. Nell'esempio in questione è presente una classe che estende una superclasse che a sua volta implementa un'interfaccia. Sono presenti anche delle particolarità sintattiche del codice come il costruttore compatto e la funzione short, il primo serve per dichiarare un costruttore che inizializzi direttamente i campi, senza inutili sprechi di codice, mentre la seconda è sempre una dichiarazione di funzione fatta su di una riga, utilizzando la stringa => al posto del return.

3.3.2 Tipizzazione opzionale

Come visto in precedenza, *Dart* fornisce come opzione al programmatore una tipizzazione sia statica che dinamica. Quando sta facendo esperimenti il programmatore può scrivere codice non tipizzato per una semplice prototipizzazione, ma nel momento in cui l'applicazione diventa più grande e stabile, i tipi possono essere aggiunti per aiutare il debugging e imporre una struttura dove è necessaria. Ad esempio di seguito troviamo del codice *Dart* non tipizzato che crea una nuova classe Punto, che ha due parametri (x e y) e due metodi (scala() e fattore()).

Listing 3.2: Esempio di codice Dart non tipizzato

```
class Punto {
  var x, y;
  Punto(this.x, this.y);
  scala(fattore) => new Punto(x*fattore,
  y*fattore);
  distanza() => Math.sqrt(x*x + y*y);
}

main() {
  var a = new Punto(2,3).scala(10);
  print(a.distanza());
}
```

In questo caso, come si può notare, le due variabili sono dichiarate come var e quindi non sono tipizzate. Ecco di seguito invece l'esempio con le due variabili tipizzate (num). In questo modo un Punto contiene due valori di tipo num e anche la funzione distanza() deve restituire un num.

Listing 3.3: Esempio di codice Dart tipizzato

```
class Punto {
  num x, y;
  Punto(this.x, this.y);
  scala(fattore) => new Punto(x*fattore,
  y*fattore);
```

```
  num distanza() => Math.sqrt(x*x + y*y);
}

main() {
  var a = new Punto(2,3).scala(10);
  print(a.distanza());
}
```

3.3.3 Librerie

Dart fornirà le seguenti librerie per supportare lo sviluppo web e web server (più ovviamente altre che in questo momento non ci interessano):

- *Core* che contiene le interfacce per supportare operazioni e strutture dati comuni.
- *DOM* che contiene le interfacce al *DOM*[2] dell'*HTML 5*.

3.3.4 Tool

Dato che *Dart* è un linguaggio in definizione, momentaneamente è disponibile un solo tool, chiamato *Dart Editor* (vedi figura 3.2), che è un add-on open-source di *Eclipse* che permette di creare, modificare e lanciare applicazioni web scritte in *Dart*. Con un click, è possibile compilare una applicazione *Dart*, trasformandola in *Javascript* e lanciarla in un browser.

3.4 Dove eseguire Dart?

Dart mostra la sua natura particolare anche nel modo in cui viene eseguito, infatti a differenza di un comune linguaggio, può essere mandato in esecuzione in diversi modi. E' possibile infatti:

[2]Il Document Object Model (spesso abbreviato come DOM), letteralmente modello a oggetti del documento, è una forma di rappresentazione standard (W3C) dei documenti strutturati come modello orientato agli oggetti. In questo modo è possibile accedere e manipolare il documento tramite le interfacce di programmazione delle applicazioni (alcune di esse sono standardizzate dal W3C).

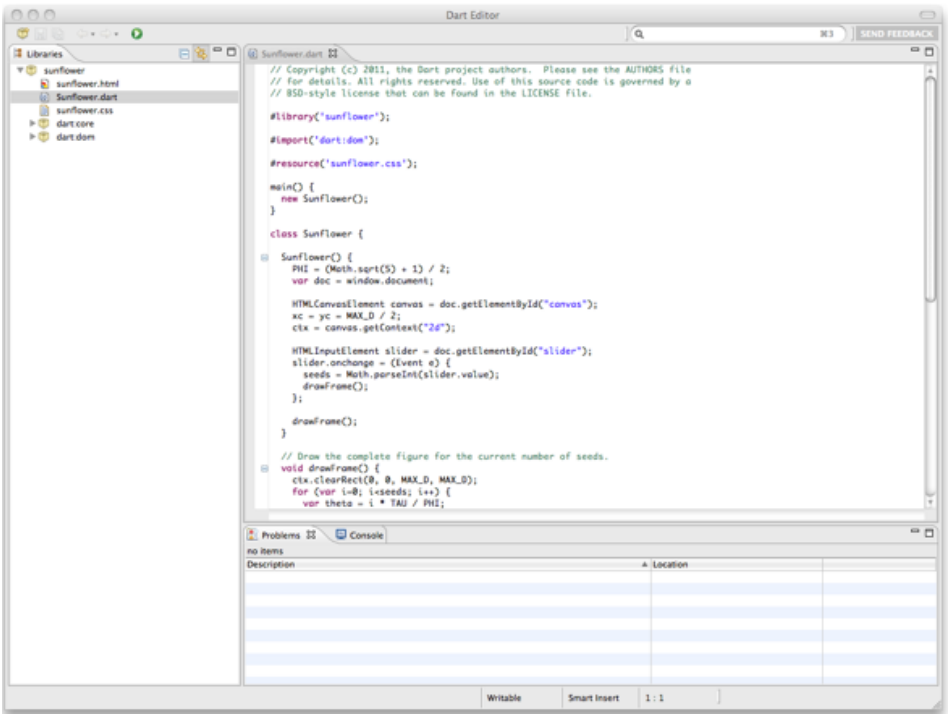

Figura 3.2: Il *Dart Editor* (add-on di *Eclipse*)

- Compilare codice *Dart* trasformandolo in codice *JavaScript* e poi mandarlo in esecuzione normalmente su qualunque browser che supporti quest'ultimo.
- Usare la *Dartboard* (vedi 3.4.1) per scrivere, modificare ed eseguire piccoli programmi *Dart* dentro il browser.
- Eseguire *Dart* direttamente su un browser (*Chromium* ma detto anche *Dartium* in questo caso, vedi 3.3) che contiene una virtual machine apposita (in questo momento questa è l'unica opzione disponibile per eseguire *Dart* su un browser). In questo caso non occorre compilare *Dart* in codice *JavaScript*, baseterà passarlo invece come tipo MIME.

Figura 3.3: Il browser Chromium

3.4.1 Dartboard

La *Dartboard* è una piccola finestra all'interno del browser per scrivere, modificare ed eseguire piccoli programmi *Dartboard*. Esiste anche una versione all'indirizzo *try.dartlang.org* che permette di eseguire il codice su una virtual machine *Dart* presente su un server remoto, e di vederne i risultati sempre dentro alla *Dartboard*. La *Dartboard* traduce il codice *Dart* in *Javascript* e nel caso di presenza di *isolate* di tipo *heavy* li mappa in *web worker*. Vedremo in 5.2.3 come questo renda possibile che codice eseguito sulla *Dartboard*, che presenti *heavy isolate*, sia effettivamente più veloce di codice sequenziale.

Figura 3.4: La Dartboard

Figura 3.5: L'esempio di Hello Word eseguito sulla Dartboard

Capitolo 4

Isolate

4.1 Introduzione

In *Dart*, è possibile strutturare le applicazioni tramite un modello ad attori ispirato ad *Erlang*, questo grazie all'utilizzo di entità dette *isolate*, che sono gli attori di *Dart*. Il nome *isolate* dovrebbe servire a sottolineare la loro proprietà più importante: ogni *isolate* ha il suo stato locale, quindi non ci dovrebbe essere condivisione di memoria/-campi/variabili tra due o più *isolate*. Essi dovranno avere la loro percezione del mondo separata, ognuno dagli altri e dall'esterno dovrà essere possibile far cambiare lo stato di un'altro *isolate* solo attraverso l'uso dei messaggi. Tutto ciò dovrebbe garantire una serie di vantaggi visti nel capitolo 2.6. Parliamo ovviamente al condizionale poichè propio nel capitolo successivo (sec:analisiIsolate) analizzeremo a fondo il linguaggio *Dart* per capire se il modello ad *isolate* rispecchia quello degli attori, visto già nel paragrafo 2.2 e se le proprietà fondamentali sono rispettate o meno.

Il supporto che ci fornisce *Dart* con gli *Isolate* per realizzare applicazioni web concorrenti è fondamentale per avere i benefici visti e descritti in 1.3 come:

- Performance migliori (su architetture hardware adatte).
- *Responsiveness* migliorata.
- Robustezza ed affidabilità.

4.2 Comunicazione tra isolate: le porte

Come già visto nelle caratteristiche del modello ad attori, gli *isolate* possono comunicare attraverso il protocollo di messaggistica asincrona, per fare questo utilizzano delle mailbox. In *Dart* però il concetto di mailbox è esteso ed è infatti possibile creare più mailbox per un solo attore (mentre ad esempio in *ActorFoundry* questo non era possibile dato che per l'invio dei messaggi si utilizzava proprio il nome dell'*isolate*).

Listing 4.1: Creazione di due porte di ricezione funzionanti sullo stesso *Isolate*

```
class Printer extends Isolate {
  main() {
    port.receive((message, replyTo) {
      if (message == null) port.close();
      else{
         print("mex: ${message}");
         replyTo.send(message,port
           .toSendPort());
      }
    });
  }
}

main() {
  ReceivePort _receivePort1;
  _receivePort1 = new ReceivePort();
      _receivePort1.receive(
                void _(var message,
                  SendPort replyTo) {
                    print ("Receiving from
                     PORT1: ${message}");
                    _receivePort1.close();
                }
            );
  ReceivePort _receivePort2;
  _receivePort2 = new ReceivePort();
```

```
      _receivePort2.receive(
                void _(var message,
                  SendPort replyTo) {
                    print ("Receiving from
                      PORT2: ${message}");
                    _receivePort2.close();
                }
            );

  new Printer().spawn().then((port) {
    for (var message in ['Hello1', 'from1',
        'other1', 'isolate1']) {
      port.send(message, _receivePort1
        .toSendPort());

    }
    for (var message in ['Hello2', 'from2',
        'other2', 'isolate2']) {
      port.send(message, _receivePort2
        .toSendPort());

    }
    port.send(null);
  });
}
```

Il risultato dell'esecuzione del codice (figura 4.1) non mostra nessun ostacolo alla creazione di due *ReceivePort* sullo stesso *Isolate*. Inoltre è possibile vedere come il processamento e l'esecuzione tra le due porte sia atomico, cioè viene processato un messaggio su di una porta poi eseguite le operazioni relative e si passa al successivo messaggio (di una delle due porte).

In *Dart* **esistono dei riferimenti alle mailbox che prendono il nome di *porte***, esse sono di due tipi differenti:

- *ReceivePort*, riferimento ad una mailbox, utilizzato per ricevere dei messaggi.

```
mex: from2
mex: other2
mex: isolate2
Receiving from PORT1: Hello1
Receiving from PORT2: Hello2
```

Figura 4.1: Risultato dell'esecuzione di interazioni tra due isolate in cui uno ha due *ReceivePort* a disposizione

- *SendPort*, riferimento ad una mailbox, utilizzato per spedire dei messaggi.

Questo fondamentalmente per un fattore di sicurezza, infatti un isolate dovrebbe ricevere i suoi messaggi e non poter vedere quelli degli altri, per cui è importante che il riferimento alla mailbox di un altro isolate sia tale per cui sia possibile solo inviare messaggi e non leggerli.

Ogni *isolate* può dotarsi di una porta *ReceivePort* in due modi differenti:

- In maniera implicita quando viene creato da un'altro *isolate* tramite il metodo *spawn*, il campo *port* dell'*isolate* si popola automaticamente.
- In maniera esplicita come istanza della classe *ReceivePort()*.

Per dotarsi invece di una porta *SendPort* è sufficiente richiamare il metodo *toSendPort()* su un oggetto *ReceivePort*, in questa maniera si ottiene la porta per la spedizione dei messaggi di una mailbox specifica.

E' importante sottolineare che nel caso venisse chiusa la *ReceivePort* di un *isolate* (attraverso il metodo *close()*), esso risulterà irraggiungibile attraverso quella porta.

Per ricevere i messaggi che arrivano su una porta *ReceivePort()* occorre utilizzare il metodo *receive()*:

Listing 4.2: Signature del metodo receive() di un oggetto *ReceivePort()*

```
void receive(void callback(message,
        SendPort replyTo))
```

Esso permette di settare una funzione di *callback* che verrà chiamata ogniqualvolta ci saranno messaggi pendenti sulla porta di ricezione (quindi non per un solo messaggio, ma per tutti quelli che arriveranno fino a che la porta sarà aperta). Per questo motivo occorre discriminare i vari messaggi in arrivo tramite dei costrutti di selezione poichè *Dart* non mette a disposizione un pattern matching per semplificare questa operazione. Come si può vedere la *callback* fornisce sia il messaggio che è stato inviato all'*isolate* sia la *SendPort* del mittente in modo tale che il ricevente possa poi inviargli una risposta in maniera semplice.

Per inviare un messaggio ad un *isolate* occorre utilizzare il metodo *send()* della *SendPort()* a cui vogliamo spedirlo:

Listing 4.3: Signature del metodo send() di un oggetto *SendPort()*

```
void send(var message, SendPort replyTo);
```

Questa funzione ha come parametri ovviamente il messaggio da inviare ma anche una *SendPort* che si riferisce alla mailbox del mittente, in questo modo è possibile che il ricevente invii una risposta al mittente.

4.3 Creazione di un isolate

Un *isolate* può essere creato estendendo la classe *Isolate* e mandato in esecuzione tramite il metodo *spawn()* all'interno di un altro *Isolate* o di un oggetto.

Listing 4.4: Esempio di creazione di un *isolate* tramite estensione della classe *Isolate*

```
1 class Worker extends Isolate {
2         Worker() : super.heavy();
3
4         main() {
5       ...
6     }
```

```
7
8 }
```

Listing 4.5: Chiamata del metodo *spawn()* per mandare in esecuzione un *isolate* ed utilizzo del metodo *then* per il recupero della future

```
1 Worker worker = new Worker();
2   worker.spawn().then((SendPort port) {
3         // callback
4 });
```

La messa in esecuzione con il metodo *spawn()* di un *isolate* restituisce la *SendPort* di quest'ultimo (in modo tale da avere un riferimento per comunicare con lui) in una future (così anche questa chiamata risulterà asincrona, in linea con il modello ad attori). Per questo motivo viene utilizzato il metodo *then()*, cioè una callback chiamata quando sarà disponibile la *SendPort* dalla future.

4.3.1 Heavy and light isolate

Si può notare come al punto 2 del primo dei due listati precedenti, si richiami il costruttore padre con nome (*named constructor*). Esistono due tipi di costruttore: *light* e *heavy*, entrambi creano un *isolate* con uno stato nuovo ovviamente non condiviso e possono solo comunicare in maniera asincrona via porte (tutto come nella norma). La differenza si trova nel fatto che i *light isolate* vivono nello stesso thread dell'*isolate* che li ha creati, quindi una sola esecuzione può avvenire in un istante (se sono due, o un *isolate* o l'altro) , mentre gli *heavy isolate* vengono eseguiti in thread creati ex-novo. Negli *heavy isolate* ovviamente è possibile che due *isolate* vengano eseguiti concorrentemente, dato che si trovano in thread diversi. Questo vantaggio si paga con dei costi di performance dato che si ha un maggiore overhead per creare un nuovo thread. Vedremo in seguito (5.2.3) come avviene lo scheduling di un *heavy isolate* e di un *light isolate* e se queste due modalità garantiscono o meno la proprietà di *fairness*.

Capitolo 5

Analisi del modello ad Isolate

5.1 Introduzione

In questo capitolo analizzeremo a fondo il linguaggio *Dart* per studiare com'è fatto il modello ad *isolate* e se quindi rispecchia quello ad attori, visto già nel paragrafo 2.2. Nella prima parte cercheremo di capire se gli *isolate* e *Dart* garantiscono le proprietà semantiche descritte a fondo nel capitolo 2.9, come già fatto per altri framework (vedi 2.11). Per fare questo dobbiamo necessariamente eseguire dei test, in entrambi gli - attualmente possibili - ambienti esecutivi, cioè la *Dartboard* e *Chromium*. Nella seconda parte metteremo in atto dei test di velocità su *Javascript* con l'utilizzo o meno di *web worker* per capire se *Dart* e gli *isolate* permettano performance migliori del linguaggio (*Javascript*) che vorrebbero rimpiazzare.

5.2 Verifica delle proprietà semantiche

5.2.1 Incapsulamento: state encapsulation

Per verificare che la proprietà di *state encapsulation* del modello ad *isolate* sia verificata, dobbiamo controllare che non sia possibile:

- Modificare lo stato di un isolate in modi differenti dal semplice utilizzo di operazioni richiamate da messaggi.
- Avere riferimenti all'interno dell'isolate ad altre entità (stato condiviso).

Listing 5.1: Verifica della proprietà di state encapsulation tramite tentativo di modifica dello stato attraverso metodi e non solo tramite operazioni richiamate da messaggi

```
class Printer extends Isolate {
  int status = 0;

  int changeStatus(){
    status++;
    return status;
  }

  int getStatus(){
    return status;
  }

  main() {
    port.receive((message, replyTo) {
      if (message == null) port.close();
      else{
         print("status: ${this.getStatus()}");
         replyTo.send(message,port.toSendPort());
      }
    });
  }
}

main() {
  Printer p = new Printer();
  print ("status "+p.getStatus());
  print ("status "+p.changeStatus());
  ReceivePort _receivePort1;
```

```
  _receivePort1 = new ReceivePort();
      _receivePort1.receive(
                void _(var message,
                        SendPort replyTo) {
                    print ("Receiving
                        from PORT1: ${message}");
                    _receivePort1.close();
                }
            );


    p.spawn().then((port) {
    for (var message in ['Hello1', 'from1',
                'other1', 'isolate1']) {
      port.send(message,_receivePort1.
                toSendPort());
      p.changeStatus();

    }
    print ("Status "+p.getStatus());

    port.send(null);
  });
}
```

```
status 0
status 1
Status 5
MyStatus: 0
MyStatus: 0
MyStatus: 0
MyStatus: 0
```

Figura 5.1: Verifica della proprietà di state encapsulation tramite tentativo di modifica dello stato attraverso metodi e non solo tramite operazioni richiamate da messaggi

Per quanto riguarda il primo punto abbiamo verificato grazie al codice precedente che è possibile creare un oggetto che risponde a dei metodi dalla stessa classe che creerà l'isolate, ma l'oggetto e l'isolate saranno entità completamente distinte (come si può vedere in 5.1 lo stato dell'isolate non viene modificato dalla chiamata al metodo). Questo test da lo stesso risultato indipendentemente dal fatto che l'isolate sia *heavy* o *light*. Quindi **non è possibile modificare lo stato dell'isolate in modi differenti dal semplice utilizzo di operazioni richiamate da messaggi**.

Per quanto riguarda invece il secondo punto, sappiamo che i riferimenti a memoria condivisa possono essere passati solo tramite messaggio (visto quanto detto sopra). Per questo motivo effettueremo due prove. Nella prima proveremo a passare un riferimento ad una Receive Port di un attore ad un altro attore mentre nella seconda proveremo a passare un riferimento ad un oggetto qualunque per poi utilizzarlo.

Come sappiamo la mailbox di un attore è privata, ed i messaggi in arrivo possono essere letti solamente dall'attore proprietario della stessa. Questo è parte fondamentale del principio di *state encapsulation*. Se così non fosse, significherebbe che due attori hanno riferimenti a memoria condivisa (in questo caso la memoria della mailbox). Dato che, come abbiamo visto in precedenza, è possibile comunicare solo tramite messaggistica, l'unico modo per far si che due attori ricevano messaggi sulla stessa mailbox è quello, nell'ordine di:

- Creare una mailbox su di un attore e provare a passarla tramite messaggio al secondo attore, che chiameremo *attore spia*.
- Una volta ottenuto il riferimento potrebbe essere possibile per l'attore spia, ricevere i messaggi del primo attore.

Listing 5.2: Verifica della proprietà di *state encapsulation* tramite tentativo di ricezione di messaggi su una mailbox di un altro attore

```
class Printer extends Isolate {
  ReceivePort _receivePort1;

  main() {
    port.receive((message, replyTo) {
```

```
      if (message == null) port.close();
      else{
         print("porta arrivata!");
         _receivePort1 = message;
         // attivo la ricezione sulla
         // porta di un altro
         // isolate passatami, in modo
         // da poter leggere
         // i messaggi che gli arrivano
         _receivePort1 = new ReceivePort();
         _receivePort1.receive((message1,
                 replyTo1) {
         if (message1 == null) port.close();
         else{
           print("SPY ISOLATE: Receiving
                from PORT1: ${message1}");

         }
        });
        // Questo messaggio dovrebbe
        // leggerlo solo l'altro attore
        // e non noi...
        replyTo.send("ciao",_receivePort1.
                toSendPort());
      }
    });
  }
}

main() {
  ReceivePort _receivePort1;
  _receivePort1 = new ReceivePort();
      _receivePort1.receive(
                void _(var message,
                        SendPort replyTo) {
                    print ("ISOLATE: Receiving
                        from PORT1: ${message}");
```

```
                    replyTo.send(message);
                    _receivePort1.close();
                }
            );

  new Printer().spawn().then((port) {
    for (var message in ['Hello1']) {
      // invio come messaggio la mia Receive Port
      port.send(_receivePort1,
        _receivePort1.toSendPort());

    }

    port.send(null);
  });
}
```

Grazie al codice precedente, abbiamo verificato che:

- Utilizzando il secondo parametro non è possibile inviare la *Receive Port* (il riferimento alla mailbox in lettura) all'isolate spia (nelle prime prove il riferimento veniva automaticamente modificato in una *Send Port*, mentre nelle ultime l'esecuzione di blocca).

- Utilizzando il primo parametro è possibile inviare correttamente la *Receive Port* all'isolate spia.

Una volta ottenuta la *Receive Port*, l'isolate spia potrà ricevere i messaggi che arrivano sulla mailbox del primo isolate, tramite una nuova *receive* (vedi figura 5.2). Tutto questo funziona solamente se l'isolate spia è *light*, al contrario se è *heavy* non è possibile farlo. **Questo fa si che non venga garantita la proprietà di *state encapsulation* in *Dart* (solo per quanto riguarda i *light* isolate)** e quindi è stato necessario segnalare il problema alla community che lavora a *Dart*. Dalle risposte pervenuteci, sembra trattarsi di un bug (vedi 5.3).

```
porta arrivata!
ISOLATE: Receiving from PORT1: ciao
SPY ISOLATE: Receiving from PORT1: ciao
```

Figura 5.2: Verifica della proprietà di state encapsulation tramite tentativo di ricezione di messaggi su una mailbox di un altro attore

On 10 March 2012 10:18, resp...@hotmail.it <resp...@hotmail.it> wrote:

> Hi, I'm working at my thesis about Dart and Isolates (computer science
> engineering). I should prove if Dart guarantees encapsulation and
> method's atomic execution.
> I've already done some test about it and I found some interesting
> things (it's possible to pass a Receive Port to another actor and use
> it to read messages of another actor)

This is a bug: you should only be able to pass SendPort.

Figura 5.3: La community di Dart risponde sulla probabile presenza di un bug nella ricezione da ReceivePort altrui

5.2.2 Incapsulamento: safe mesaging

Per verificare che la proprietà di *safe mesaging* del modello ad *isolate* sia verificata, dobbiamo controllare che non sia possibile passare parametri per riferimento. Per effettuare questo test abbiamo creato un oggetto, tentando di passarne il riferimento via messaggio. Se questo è possibile e si può anche modificare lo stato dell'oggetto dal nuovo isolate allora non sarà garantita neanche la proprietà di *state encapsulation*.

Listing 5.3: Verifica della proprietà di *safe messaging* e di *State encapsulation* tramite tentativo di spedizione di messaggio contenente un riferimento ad un oggetto

```
class Printer extends Isolate {
  main() {
    port.receive((message, replyTo) {

      if (message == null){  print("null");
       port.close();}
      else{
         print("mex: ${message}");
         message.changeStatus(2);
         print("mex: ${message}");
         //replyTo.send(message,
                port.toSendPort());
      }
    });
  }
}

class Shared{
  int status;

  void shared(){
    this.status = 0;
  }

  int changeStatus(int c){
```

```
    this.status = c;
    return status;
  }

  int getStatus(){
    return status;
  }
}


main() {

  var s = new Shared();
  s.changeStatus(1);
  print ("Status in main actor: "
        +s.getStatus());

  ReceivePort _receivePort1;
  _receivePort1 = new ReceivePort();
      _receivePort1.receive(
                void _(var message,
                        SendPort replyTo) {
                    print ("Receiving
                      from PORT1: ${message}");
                    _receivePort1.close();
                }
            );


  new Printer().spawn().then((port) {
      print ("Sending Message");
      port.send(s,_receivePort1.
        toSendPort());
      print ("Message send");
      port.send(null);
  });
}
```

Il risultato dell'esecuzione del codice (figura 5.4) mostra come essa si blocchi nel tentativo di passare un riferimento ad un oggetto. Questo significa che è garantita la proprietà di *Self mesaging* dato che **non è possibile passare riferimenti in un messaggio (passaggio di parametri per valore) ed è garantita anche la proprietà di *State encapsulation* poichè non potendo passare un oggetto via messaggio è impossibile avere oggetti condivisi su due isolate differenti**. Questo test da lo stesso risultato indipendentemente dal fatto che l'isolate sia *heavy* o *light*.

```
Status in main actor: 1
Sending Message
```

Figura 5.4: Verifica della proprietà di *safe messaging* e di *State encapsulation* tramite tentativo di spedizione di messaggio contenente un riferimento ad un oggetto

5.2.3 Fairness

Per verificare la proprietà di *fairness* dobbiamo controllare che l'entità che si occupa dello scheduling permetta a tutti gli attori di eseguire, nella maniera più equa possibile, il loro lavoro. Sono due le situazioni in cui questa proprietà potrebbe venir compromessa:

- Un isolate processa un messaggio che fa eseguire una serie di operazioni potenzialmente infinite e tutti gli altri muoiono di stenti.
- Un isolate non viene mai schedulato nonostante non vi siano isolate che eseguono operazioni infinite.

Per controllare che non si verifichi il primo caso, creiamo un pool di isolate e gli inviamo subito dei messaggi (in modo tale che abbiano più di un messaggio nella mailbox ciascuno). Ad ognuno di essi facciamo eseguire molte operazioni per ogni messaggio nella mailbox (nel nostro caso inviano un elevato numero di messaggi) ed analizziamo il

comportamento dello scheduler (verifichiamo chi e quando viene messo in esecuzione).

Listing 5.4: Verifica della proprietà di fairness attraverso la creazione di attori che portano a termine molte operazioni per messaggio

```
class Printer extends Isolate {
  //Printer() : super.heavy();
  main() {
    port.receive((message, replyTo) {

      if (message == null){ port.close();}
      else{

        for(int x = 0;x<100;x++){
         print ("operation n.${x} -
           Actor ${message["id"]} sending :
                  ${message["say"]}");
         replyTo.send(message,port.toSendPort());
        }
      }
    });
  }
}


main() {

  int n = 0;
  ReceivePort _receivePort1;
  _receivePort1 = new ReceivePort();
      _receivePort1.receive(
                void _(var message,
                  SendPort replyTo) {
                    print ("Actor
                     ${message["id"]}
                       sent : ${message["say"]}");
```

```
                    //_receivePort1.close();
                }
            );

  for(int x = 0;x<5;x++){
    print ("${x}");
    new Printer().spawn().then((port) {

        var mex = { "id" : n, "say" : "ping" };
        print ("Create actor ${mex["id"]}
                and sending message");
        port.send(mex, _receivePort1.toSendPort());
        port.send(mex, _receivePort1.toSendPort());
        n++;

        port.send(null);
    });
  }

}
```

Il risultato che omettiamo per ovvie ragioni di spazio, mostra differenze a seconda che si utilizzino isolate *light* piuttosto che *heavy* nella *Dartboard*. Nel caso di isolate *light*, si può notare (vedi immagine 5.5) come la computazione non avvenga parallelamente, infatti in sequenza abbiamo che:

1. l'isolate main invia tutti i messaggi agli attori (due per ogni attore)

2. il primo isolate legge il primo messaggio e spedisce le 100 risposte all'isolate main

3. il primo isolate legge il secondo messaggio e spedisce le 100 risposte all'isolate main

4. il secondo isolate legge il primo messaggio e spedisce le 100 risposte all'isolate main

5. il secondo isolate legge il secondo messaggio e spedisce le 100 risposte all'isolate main

6. e così via per tutti i rimanenti isolate

7. il main gestisce tutte le 100x5 risposte degli isolate

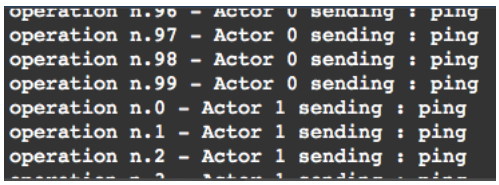

Figura 5.5: Verifica della proprietà di fairness nel caso di light isolate; Ogni isolate riceve tutti i suoi messaggi ed esegue le relative operazioni prima di essere schedulato per fare spazio ad un altro isolate

La schedulazione di un nuovo isolate, avviene quando il corrente ha terminato il suo lavoro (quindi ha processato tutti i messaggi ed eseguito le relative operazioni), questo in accordo con la teoria già vista in 4.3.1, dato che essi vivono in un unico thread. Ciò significa anche che **se un isolate esegue operazioni infinite, tutti gli altri isolate finiranno in starvation e se il thread che gestisce questi isolate è anche quello che gestisce la user interface, può bloccarsi il browser** (vedi immagine 5.6), **per questo motivo la proprietà di *fairness* non è garantita nel caso *light***.

Nel caso di *heavy* isolate invece, si può notare come ci sia una parvenza di concorrenza tra gli isolate, dato che l'output (vedi figura 5.7) mostra diverse operazioni di diversi isolate che sono intervallate. Questo significa che la proprietà di *fairness* è rispettata. Tutto ciò però non garantisce che vi sia una vera e propria esecuzione parallela (visto che potrebbero essere schedulati o meno sullo stesso thread), sarà compito di test successivi mostrare se effettivamente l'esecuzione avviene concorrentemente.

Quindi ricapitolando, grazie a questi test, si è potuto verificare come nella *Dartboard* esista una profonda differenza nel caso si utilizzino isolate *light* piuttosto che *heavy*. Nel caso *light*, ogni isolate processa

Figura 5.6: Il browser va in crash a causa dell'esecuzione di un loop infinito all'interno di un isolate light

```
operation n.58 - Actor 0 sending : ping
Actor 0 sent : ping
operation n.59 - Actor 0 sending : ping
operation n.0 - Actor 2 sending : ping
Actor 2 sent : ping
operation n.1 - Actor 2 sending : ping
Actor 2 sent : ping
```

Figura 5.7: Verifica della proprietà di fairness nel caso di heavy isolate: in questo caso le operazioni di vari isolate sono intervallate

tutti i suoi messaggi in coda ed esegue tutte le operazioni relative, prima di venire schedulato lasciando spazio ad un altro isolate. Per questo motivo se uno degli isolate che compone il nostro sistema processa un messaggio che da luogo ad una serie infinita di operazioni oppure ha una coda infinita di messaggi, tutti gli altri isolate finiranno in starvation. Nel caso *heavy* invece questo non succede, dato che le operazioni degli isolate in seguito al processamento di messaggi sono intervallate. Tutto ciò però non garantisce che vi sia una vera e propria esecuzione parallela (visto che potrebbero essere schedulati o meno sullo stesso thread), sarà compito di test successivi mostrare se effettivamente l'esecuzione avviene concorrentemente.

Tornando all'oggetto di questa sezione, dobbiamo ancora controllare se un isolate viene schedulato o meno nonostante non vi siano isolate che eseguono operazioni infinite. Per fare ciò creiamo un pool di isolate *light* (per gli *heavy* abbiamo già controllato la proprietà di *fairness*) che in continuazione si mandano messaggi (operazioni brevi) per vedere come vengono schedulati (e se quindi tutti a turno vengono eseguiti).

Listing 5.5: Verifica della modalità di schedulazione degli isolate light

```
class Printer extends Isolate {
  //Printer() : super.heavy();
  main() {
    port.receive((message, replyTo) {

      if (message == null){ port.close();}
      else{


        print ("Actor ${message["id"]}
               sending : ${message["say"]}");
        replyTo.send(message,port
               .toSendPort());

      }
    });
```

```
  }
}


main() {

  int n = 0;
  int w = 0;
  ReceivePort _receivePort1;
  _receivePort1 = new ReceivePort();
      _receivePort1.receive(
                void _(var message,
                        SendPort replyTo) {
                    print ("Actor
                      ${message["id"]}
                        sent :
                         ${message["say"]}");
                    replyTo.send(message,
                        _receivePort1
                          .toSendPort());
                    w++;
                    if(w>100){
                      _receivePort1.close();
                    }
                }
            );

  for(int x = 0;x<5;x++){
    print ("${x}");
    new Printer().spawn().then((port) {

        var mex = { "id" : n, "say" :
                "ping" };
        print ("Create actor
                ${mex["id"]} and sending
                        message");
        port.send(mex,
```

```
                    _receivePort1.toSendPort());

            n++;

            //port.send(null);
        });
    }

}
```

Dall'esecuzione di questo codice (vedi figura 5.8) si può notare che se gli isolate sono *light* e le operazioni da eseguire per ogni messaggio non sono infinite, ogni isolate sarà schedulato in sequenza (eseguendo tutte le sue ricezioni ed operazioni) e quindi nessuno morirà di stenti. Nell'esempio:

1. scheduling del main isolate
2. scheduling del primo isolate
3. scheduling del secondo isolate
4. scheduling del terzo isolate
5. scheduling del quarto isolate
6. scheduling del quinto isolate
7. scheduling del main isolate (ricomincia il ciclo)

```
Actor 0 sending : ping
Actor 1 sending : ping
Actor 2 sending : ping
Actor 3 sending : ping
Actor 4 sending : ping
Actor 0 sent : ping
Actor 1 sent : ping
```

Figura 5.8: Verifica della modalità di schedulazione degli isolate light

Tutto ciò ci fa capire come nel caso di isolate *light* la computazione non avvenga parallelamente e che occorre porre particolare attenzione nel caso si usi questo tipo di entità piuttosto che gli *heavy* isolate, se si devono portare a termine operazioni potenzialmente infinite.

In *Chromium* invece non esiste differenza tra *light* ed *heavy* isolate, essi si comportano tutti come *heavy* isolate, indipendentemente dalla presenza o meno del costruttore. Quindi all'interno di questo browser, la proprietà di *fairness* è garantita.

5.2.4 Location trasparency e Mobilità

La *Location trasparency* e la *Mobilità* per ora non sono proprietà che *Dart* garantisce, dato che si tratta di un linguaggio ancora troppo giovane per gestire questi aspetti.

5.3 Test sul parallelismo degli isolate

In questa sezione ci preoccupiamo di verificare se effettivamente gli isolate *heavy* sono eseguiti concoerrentemente oppure c'è solo un semplice scheduling che simula una concorrenza. Per fare questo prendiamo un problema che può essere risolto tramite un algoritmo che ha come modello un parallelismo indipendente (vedi 1.6.1) e ed eseguimo la computazione sia tramite isolate *heavy*, *light* e senza parallelismo (quindi in maniera sequenziale)[1]. Il problema considerato sarà quello del calcolo dell'algoritmo di Fibonacci su ogni numero di un array. L'algoritmo di fibonacci è molto semplice: ha un solo input e restituisce un solo output (per questo il modello è quello di parallelismo indipendente, non si computa su posizioni condivise dell'array).

Listing 5.6: Esecuzione dell'algoritmo di Fibonacci su un vettore per testare l'efficienza degli isolate heavy rispetto agli isolate light e ad una esecuzione sequenziale nella Dartboard

```
class PrettyStopwatch {
```

[1]Ricordiamo che nel caso di *Chromium* non c'è differenza tra *light* e *heavy*, quindi basterà effettuare il confronto tra isolate e esecuzione sequenziale.

```
  Stopwatch timer;

  PrettyStopwatch() {
    timer = new Stopwatch();
  }

  PrettyStopwatch.start() {
    timer = new Stopwatch.start();
  }

  start() {
    timer.start();
  }

  stop() {
    timer.stop();
    print("Elapsed time: "
    + timer.elapsedInMs() + "ms");
  }
}

main() {
  final fib_solver = new FibSolver();

  var list = [5, 40, 39, 32, 6, 41];
  var op = 0;


   var timer1 = new PrettyStopwatch.start();
   ReceivePort _receivePort1;
  _receivePort1 = new ReceivePort();
      _receivePort1.receive(
                void _(var message,
                SendPort replyTo) {
                      message.
                      forEach((i, answer) {
                        print("fib(" + i + ")
```

```
                        = " + answer);
                        op++;
                    });
                    if(op == 6){
                        timer1.stop();
                    }
                }

            );


new FibSolver().spawn().then((port) {

    port.send(list.getRange(0,1),
    _receivePort1.toSendPort());

});

new FibSolver().spawn().then((port) {

    port.send(list.getRange(1,1),
    _receivePort1.toSendPort());

});

new FibSolver().spawn().then((port) {

    port.send(list.getRange(2,1),
    _receivePort1.toSendPort());

});

new FibSolver().spawn().then((port) {

    port.send(list.getRange(3,1),
    _receivePort1.toSendPort());
```

```
});

new FibSolver().spawn().then((port) {

    port.send(list.getRange(4,1),
    _receivePort1.toSendPort());

});

new FibSolver().spawn().then((port) {

    port.send(list.getRange(5,1),
    _receivePort1.toSendPort());

});




 /*var value;
  int n;
  for(int x=0;x<2;x++){
    n=list[x];
    value = fib(n);
    print("fib(${n}) = ${value}");
    op++;
  }
  if(op==6){
    timer1.stop();
  }*/


/*new FibSolver().spawn().then((port) {

    port.send(list.getRange(3,3),
    _receivePort1.toSendPort());
```

```
    });*/

}

// Isolate class for solving
// lists of fibonacci numbers
class FibSolver extends Isolate {
  //FibSolver() : super.heavy();
  main() {
    var answers = {};
    port.receive((message, replyTo) {
      message.forEach((i) {
        answers[i] = fib(i);
        //print("fib(" + i + ") = " + answers[i]);
      });

      replyTo.send(answers);
    });
  }
}

fib(i) {
  if (i < 2) return i;
  return fib(i-2) + fib(i-1);
}
```

Listing 5.7: Esecuzione dell'algoritmo di Fibonacci su un vettore per testare l'efficienza degli isolate rispetto ad una esecuzione sequenziale in Chromium

```
<html>
  <script>{}</script>
  <body>
    <script type="application/dart">
      #import('dart:dom');
      #import('dart:isolate');
```

```
#import('dart:core');
#import('dart:coreimpl');


void main() {
  var elements = [];


   var e1 = window.document
   .createElement('div');

   elements.add(window.document
   .createElement('div'));

   final fib_solver = new FibSolver();

            var list =
            [5, 40, 39, 32, 6, 41];

            var op = 0;
            var time;

            var timer1 =
            new PrettyStopwatch.start();



          ReceivePort _receivePort1;
_receivePort1 = new ReceivePort();
    _receivePort1.receive(
          void _(var message,
           SendPort replyTo) {
                  message.forEach((i,
                   answer) {
                   elements.add(window.
                   document.
                   createElement('div'));
```

```
                        elements[op].
                        innerHTML =
                        "fib(" + i + ")
                        = " + answer;
                        window.document
                        .body.appendChild
                        (elements[op]);

                        op++;
                    });
                    if(op == 6){
                        time = timer1.stop();
                        elements.add(
                        window.document.
                        createElement('div'));
                                elements[op].
                                innerHTML = time;
                                    window.
                                    document.body.
                                    appendChild
                                    (elements
                                    [op]);
                }



            }

        );



new FibSolver().spawn().then((port) {

    SendPort rc = _receivePort1
    .toSendPort();
```

```
        port.send(list.getRange(5,1),
        _receivePort1.toSendPort());

    });

    /*

    new FibSolver().spawn().then((port) {

        SendPort rc = _receivePort1.
        toSendPort();
        port.send(list.getRange(1,1),
        _receivePort1.toSendPort());

    });


    new FibSolver().spawn().then((port) {

        SendPort rc = _receivePort1.
        toSendPort();
        port.send(list.getRange(2,1),
        _receivePort1.toSendPort());

    });


    new FibSolver().spawn().then((port) {

        SendPort rc = _receivePort1.
        toSendPort();
        port.send(list.getRange(3,1),
        _receivePort1.toSendPort());

    });

    new FibSolver().spawn().then((port) {
```

```
    SendPort rc = _receivePort1.
    toSendPort();
    port.send(list.getRange(4,1),
    _receivePort1.toSendPort());

});


new FibSolver().spawn().then((port) {

    SendPort rc = _receivePort1.
    toSendPort();
    port.send(list.getRange(5,1),
    _receivePort1.toSendPort());

});

*/




  var answer;

  for(int x=0;x<5;x++){

    answer = fib(list[x]);
    elements.add(window.
    document.createElement('div'));
    elements[x].innerHTML =
    "fib(" + list[x] + ") = " + answer;
    window.document.body.
    appendChild(elements[x]);
    op = op + 1;
  }
```

```
      if(op==6){
        time = timer1.stop();
        elements.add(window.
        document.createElement('div'));
        elements[op].innerHTML = time;
        window.document.
        body.appendChild(elements[op]);
      }









}

      class PrettyStopwatch {
        var c;
 Stopwatch timer;

 PrettyStopwatch() {
   timer = new Stopwatch();
   c=1;
 }

 PrettyStopwatch.start() {
   timer = new Stopwatch.start();
 }

 start() {
   timer.start();
 }

 String stop() {
   timer.stop();
   return "Elapsed time: " +
    timer.elapsedInMs() + "ms";
```

```
  }
}

// Isolate class for solving
// lists of fibonacci numbers
class FibSolver extends Isolate {
  //FibSolver() : super.heavy();
  main() {

     var answers = {};
    port.receive((message, replyTo) {
      message.forEach((i) {
        answers[i] = fib(i);

      });

      replyTo.send(answers);
    });



  }
}

fib(i) {
  if (i < 2) return i;
  return fib(i-2) + fib(i-1);
}


    </script>
  </body>
</html>
```

Per quanto riguarda l'esecuzione su *Dartboard*, i risultati visibili nella tabella 5.9 e nel grafico 5.10 mostrano come:

- **L'esecuzione con isolate *heavy* sia praticamente sempre**

migliore dell'esecuzione sequenziale e con isolate *light* (tranne che nel caso con un solo isolate, dove il tempo di esecuzione è simile agli altri due casi, questo è probabilmente dovuto al fatto che il codice nel main viene interrotto per effettuare la receive).

- Il miglior risultato si ha con 2 isolate *heavy* (8852,8 ms).
- **L'esecuzione con isolate *light* è sempre peggiore dell'esecuzione sequenziale** (probabilmente a causa dell'overhead generato quando si crea un nuovo isolate, pur essendo eseguiti in sequenza, ed è appunto per questo che più isolate si creano e maggiore è il tempo di esecuzione).

Questo significa che l'esecuzione di isolate *heavy* sulla *Dartboard* avviene parallelamente mentre l'esecuzione di isolate *light* no.

Numero di Isolate (oltre al main)	tipo di isolate	Esecuzione 1 (ms)	Esecuzione 2 (ms)	Esecuzione 3 (ms)	Esecuzione 4 (ms)	Esecuzione 5 (ms)	media (ms)
esecuzione sequenziale	-	16168	16175	16182	16197	16254	16195
1	light	16280	16310	16329	16371	16308	16320
2	light	16391	16350	16361	16692	16453	16449
3	light	16367	16355	16338	16390	16350	16360
6	light	16389	16589	16547	16407	16423	16471
1	heavy	16282	16709	16304	16288	16299	16376
2	heavy	8723	9254	8480	8456	9351	8853
3	heavy	9715	10924	10211	10909	10473	10446
6	heavy	11161	11235	11076	10837	11143	11090

Figura 5.9: Tabella contenente tutte le prove effettuate con l'algoritmo di Fibonacci su Dartboard

Per quel che riguarda invece l'esecuzione su *Chromium*, i risultati visibili nella tabella 5.11 e nei grafici 5.12, 5.13, 5.14 e 5.15, mostrano come:

- Nel caso che venga eseguita tutta la computazione sugli isolate, il tempo minore si ottiene utilizzandone due (mentre con uno ovviamente è peggiore) (vedi 5.12).
- Se spostiamo parte della computazione sul main (metà per uno), il caso migliore si ha con un isolate (come se lavorassero due isolate, e quindi riconducibile a quanto detto in precedenza) (vedi 5.13).

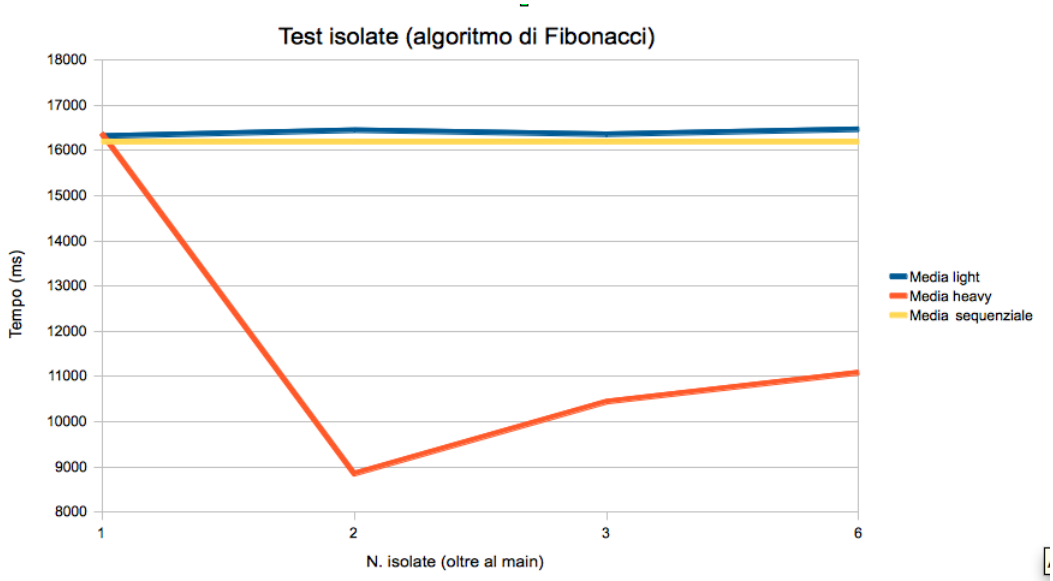

Figura 5.10: Grafico che visualizza le prove effettuate con l'algoritmo di Fibonacci su DartBoard

- Lavorando con 2 isolate (il main più un altro) e spostando la computazione, non si ottengono vantaggi degni di nota rispetto ai casi migliori precedenti (vedi 5.14).

- **A fronte di un esecuzione sequenziale da 6166 ms, il nostro caso migliore impiega 3227 ms (main più un isolate con metà computazione cadauno), cioè un miglioramento del 91 percento** (vedi 5.15).

L'ultimo punto mette in evidenza come anche l'esecuzione di isolate su *Chromium* avvenga parallelamente.

Riassumendo, si può dire che il codice ed i risultati sono descritti mostrano come **l'esecuzione di isolate *heavy* su *Dartboard* e *Chromium* avvenga parallelamente, mentre come già visto in precedenza, l'esecuzione *light* su *Dartboard* sia sequenziale.**

Numero di Isolate (oltre al main)	Quantità di esecuzione per main	Esecuzione 1 (ms)	Esecuzione 2 (ms)	Esecuzione 3 (ms)	Esecuzione 4 (ms)	Esecuzione 5 (ms)	media (ms)
esecuzione sequenziale	100,00%	6165	6168	6162	6165	6168	6166
1	0	6300	6168	6236	6410	6480	6319
2	0	3302	3305	3847	3287	3307	3410
3	0	4576	4612	4568	4638	3899	4459
6	0	4647	4799	4274	4249	4416	4477
1	50,00%	3252	3199	3195	3245	3243	3227
2	33,33%	3954	4149	3942	3876	3925	3969
5	16,66%	4624	4515	4592	4852	4620	4641
1	83,33%	3258	3245	3233	3223	3226	3237
1	66,66%	3235	3259	3239	3245	3285	3253
1	33,33%	4331	4338	4334	4350	4355	4342

Figura 5.11: Tabella contenente tutte le prove effettuate con Dart e l'algoritmo di Fibonacci su Chromium

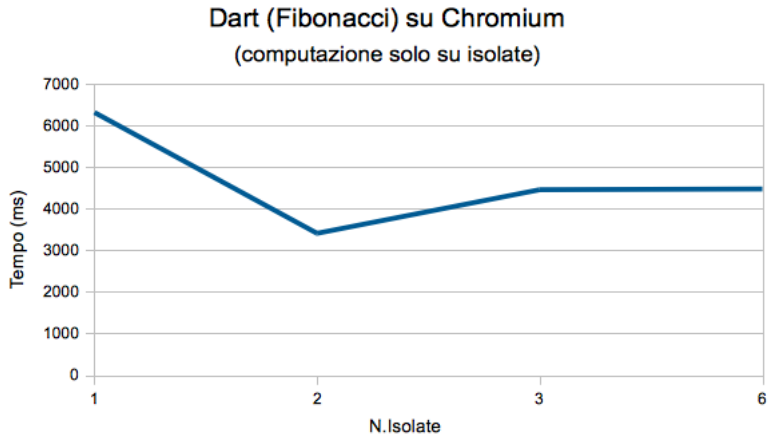

Figura 5.12: Grafico che visualizza le prove effettuate con Dart e l'algoritmo di Fibonacci su Chromium con computazione solo su isolate

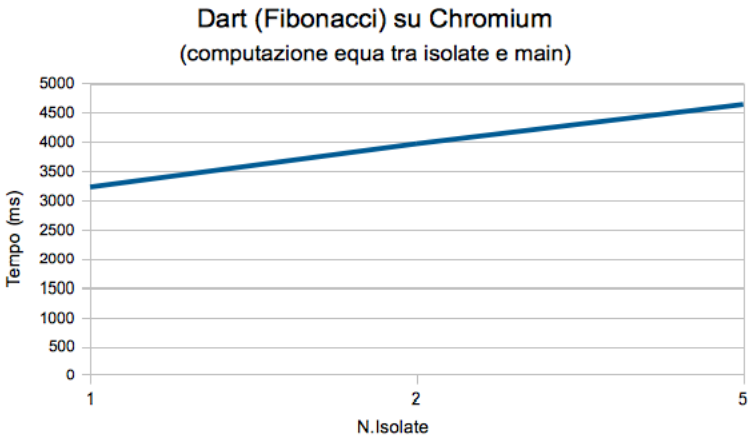

Figura 5.13: Grafico che visualizza le prove effettuate con Dart e l'algoritmo di Fibonacci su Chromium con computazione su isolate e main equa

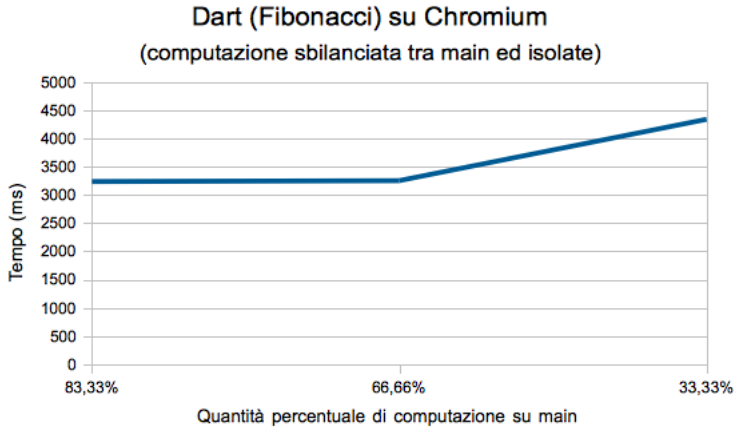

Figura 5.14: Grafico che visualizza le prove effettuate con Dart e l'algoritmo di Fibonacci su Chromium con computazione su isolate e main sbilanciata

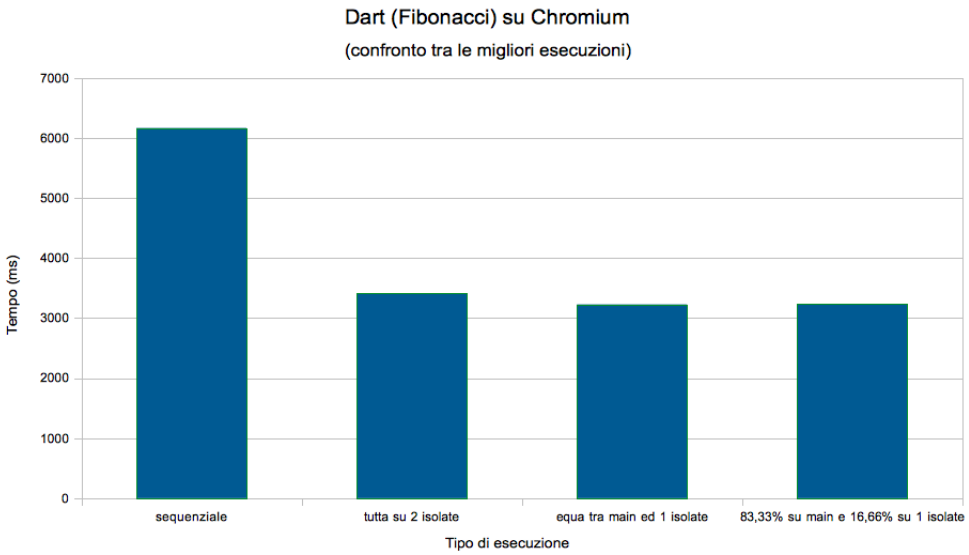

Figura 5.15: Grafico che visualizza le prove effettuate con Dart e l'algoritmo di Fibonacci su Chromium confrontando i casi migliori delle diverse modalità di computazioni effettuate

5.4 Comparativa con Javascript

Per capire se effettivamente *Dart* e gli *isolate* permettano performance migliori del linguaggio (*Javascript*) che vorrebbero rimpiazzare, dobbiamo cercare di eseguire lo stesso algoritmo (scritto ovviamente nei due linguaggi differenti) e confrontare i risultati. In precedenza abbiamo già utilizzato l'algoritmo di Fibonacci per capire se gli *isolate* fossero effettivamente messi in esecuzione in parallelo su *Chromium* e quindi abbiamo già a disposizione una serie di dati attendibili. Non ci resta altro che trascrivere l'algoritmo di Fibonacci in *Javascript* e confrontare le esecuzioni con le quelle di *Dart* trovate in precedenza. E' importante tenere in considerazione che anche *Javascript* fornisce tramite i cosiddetti *web worker*, un supporto per eseguire codice in maniera concorrente. Considereremo quindi anche questo tipo di implementazione dell'algoritmo di Fibonacci per confrontarne i risultati.

Listing 5.8: Esecuzione dell'algoritmo di Fibonacci su un vettore per testare l'efficienza di Javascript sequenziale

```
<html>
  <script>{}</script>
  <body>
    <SCRIPT LANGUAGE = "JavaScript">
        var start = new Date().getTime();
        var answer = 0;
        var op = 0;

        var list = [5, 40, 39, 32, 6, 41];
        var elements = [];



        for(x=0;x<6;x++){

        answer = fib(list[x]);
        elements[x] = window.
        document.createElement('div');
        elements[x].innerHTML =
```

```
        "fib(" + list[x] + ") = "
        + answer;
        window.document.body.
        appendChild(elements[x]);
        op = op + 1;
    }
    if(op==6){
        var end = new Date().getTime();
                var time = end - start;
        elements[op] = window.document.
        createElement('div');
        elements[op].innerHTML = time;
        window.document.body.
        appendChild(elements[op]);
    }


function fib(i) {
  if (i < 2){
        return i;
  }
  return fib(i-2) + fib(i-1);
}
</SCRIPT>  </body>
</html>
```

L'esecuzione dell'algoritmo sequenziale ha permesso di raccogliere i dati presenti nella tabella 5.16 dove si può notare (grazie anche al grafico 5.17) sia che l'esecuzione migliore è quella di Google Chromium (Dartium) ma anche che quella di Mozilla Firefox si è bloccata. Questo perchè il thread della User Interface è lo stesso che esegue i calcoli e quindi, eseguendo computazioni lunghe, il browser potrebbe non rispondere più alle sollecitazioni dell'utente.

Listing 5.9: Esecuzione dell'algoritmo di Fibonacci su un vettore per testare l'efficienza di Javascript attraverso l'utilizzo di web worker (file main)

```
<!DOCTYPE HTML PUBLIC
```

Browser	Versione	Esecuzione 1 (ms)	Esecuzione 2 (ms)	Esecuzione 3 (ms)	Esecuzione 4 (ms)	Esecuzione 5 (ms)	media (ms)
Dartium	19.0.1069.0	9093	9114	9218	9241	9202	9174
Chrome	17.0.963.79	9191	9190	9188	9173	9173	9186
Safari	5.1.2 (6534.52.7)	11199	10789	10762	10754	10689	10839
Firefox	03.06.06	bloccato	bloccato	bloccato	bloccato	bloccato	bloccato

Figura 5.16: Tabella che raccoglie i dati delle prove effettuate con l'algoritmo di Fibonacci in Javascript sequenziale su diversi browser

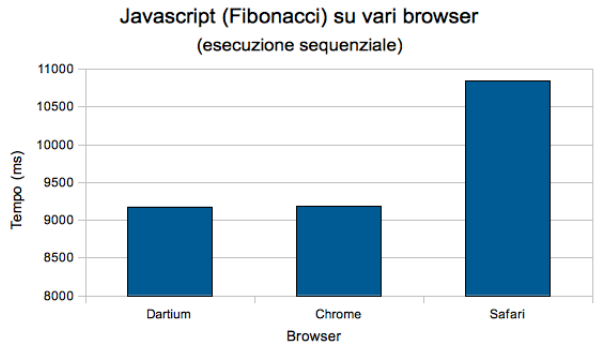

Figura 5.17: Grafico che mostra quale è il browser migliore nell'esecuzione dell'algoritmo di Fibonacci in Javascript sequenziale

```
"-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
  <title>Test threads fibonacci</title>
  <body>

  <div id="result"></div>

  <SCRIPT LANGUAGE = "JavaScript">



  var elements = [];
  var n = 0;
  var op = 0;

  var list = new Array();
  list = [5, 40, 39, 32, 6, 41];

  var start = new Date().getTime();

        var worker1 = new
        Worker('http://localhost/worker.js');
        var worker2 = new
        Worker('http://localhost/worker.js');
        //var worker3 = new
        Worker('http://localhost/worker.js');
        //var worker4 = new
        Worker('http://localhost/worker.js');
        //var worker5 = new
        Worker('http://localhost/worker.js');
        //var worker6 = new
        Worker('http://localhost/worker.js');

    worker1.onmessage
        = function(event) {
```

```
                for(i = 0;i < event.data.length;
                i++){
                n = event.data[i];
                i++;
                fib = event.data[i];

                elements[op] = window.
                document.createElement('div');
                elements[op].innerHTML
                = "fib(" + n + ") = " + fib;
                window.document.
                body.appendChild(elements[op]);
                op = op + 1;

        }

        if(op==6){
                var end = new
                Date().getTime();
                        var time =
                        end - start;
                elements[op] = window.document.
                createElement('div');
                elements[op].
                innerHTML = time;
                window.document.body.
                appendChild(elements[op]);
        }
    };


    worker2.onmessage
    = function(event) {


                for(i = 0;i <
                event.data.length; i++){
```

```
                n = event.data[i];
                i++;
                fib = event.data[i];

                elements[op] = window.document
                .createElement('div');
                elements[op].
                innerHTML = "fib(" + n + ")
                 = " + fib;
                window.document
                .body.appendChild(elements[op]);
                op = op + 1;

        }

        if(op==6){
                var end = new Date()
                .getTime();
                        var time =
                        end - start;
                elements[op] = window.
                document.createElement('div');
                elements[op].innerHTML = time;
                window.document.body
                .appendChild(elements[op]);
        }
};

/*
worker3.onmessage
= function(event) {


                for(i = 0;i <
                event.data.length; i++){
                n = event.data[i];
                i++;
```

```
                fib = event.data[i];

                elements[op] = window.
                document.createElement('div');
                elements[op].innerHTML
                = "fib(" + n + ") = " + fib;
                window.document
                .body.appendChild(elements[op]);
                op = op + 1;

        }

        if(op==6){
                var end = new
                Date().getTime();
                        var time = end - start;
                elements[op] = window.
                document.createElement('div');
                elements[op].innerHTML = time;
                window.document.body.
                appendChild(elements[op]);
        }
};

worker4.onmessage
= function(event) {


                for(i = 0;i <
                event.data.length; i++){
                n = event.data[i];
                i++;
                fib = event.data[i];

                elements[op] = window.
                document.createElement('div');
                elements[op].innerHTML =
```

```
                "fib(" + n + ") =
                " + fib;
                window.document.body.
                appendChild(elements[op]);
                op = op + 1;

        }

        if(op==6){
                var end = new
                Date().getTime();
                        var time =
                        end - start;
                elements[op] = window.
                document.createElement('div');
                elements[op].
                innerHTML = time;
                window.document.body.
                appendChild(elements[op]);
        }
    };

    worker1.onerror =
    function(error) {

      alert("worker error");
    };


    worker2.onerror =
    function(error) {

      alert("worker error");
    };

   */
```

```
        [5, 40, 39, 32, 6, 41];


worker1.postMessage([ 39, 32]);
worker2.postMessage([6 ,41]);
//worker3.postMessage([39]);
//worker4.postMessage([32]);
//worker5.postMessage([41]);
//worker6.postMessage([6]);

var answer = 0;

    for(x=0;x<2;x++){

    answer =
    fib(list[x]);
    elements[op] = window.
    document.createElement('div');
    elements[op].innerHTML =
    "fib(" + list[x] + ") = " + answer;
    window.document.body.
    appendChild(elements[op]);
    op = op + 1;
}
if(op==6){
    var end = new Date().
    getTime();
            var time = end - start;
    elements[op] = window.
    document.createElement('div');
    elements[op].
    innerHTML = time;
    window.document.body.
    appendChild(elements[op]);
}
```

```
function fib(i) {
  if (i < 2){
        return i;
  }
  return fib(i-2) + fib(i-1);
}



</SCRIPT>


  </body>
</html>
```

L'esecuzione dell'algoritmo con web worker ha permesso di raccogliere i dati presenti nella tabella 5.18 dove si può notare che l'esecuzione migliore è quella di Google Chromium con 2 web worker (vedi meglio in 5.20). Inoltre si vede (figura 5.19) come l'esecuzione di Mozilla Firefox sia notevolmente più lunga rispetto alle altre.

Numero di Isolate (oltre al main)	Browser	Esecuzione 1 (ms)	Esecuzione 2 (ms)	Esecuzione 3 (ms)	Esecuzione 4 (ms)	Esecuzione 5 (ms)	media (ms)
1	Chrome	9235	9224	9225	9214	9228	9225
2	Chrome	4755	4802	4765	4786	4760	4774
3	Chrome	5689	5802	5503	5422	5745	5632
6	Chrome	5720	5727	5875	5827	5558	5741
1	Safari	11031	11060	10998	10893	10866	10970
2	Safari	5751	5705	5667	5715	5687	5705
3	Safari	6766	6831	6822	6823	6798	6808
6	Safari	6882	6509	6572	6853	6638	6691
1	Firefox	134815	135991	133744	134287	136554	135078
2	Firefox	71006	71245	72185	76473	72534	72689
3	Firefox	81708	80666	82332	81493	81267	81493
6	Firefox	83529	82544	83447	82898	82331	82950
1	Chromium	9069	9119	9081	9076	9071	9083
2	Chromium	4699	4694	4703	4752	4750	4720
3	Chromium	5491	5350	5570	5530	5526	5493
6	Chromium	5693	5282	5585	5609	5781	5590

Figura 5.18: Tabella che raccoglie i dati delle prove effettuate con l'algoritmo di Fibonacci in Javascript con l'utilizzo di web worker

Ora che sono disponibili tutti i dati possiamo raccorglierli in una tabella per confrontare i risultati. In 5.21 e 5.22 si considerano solo le implementazioni migliori dei tre linguaggi eseguiti nel browser dove

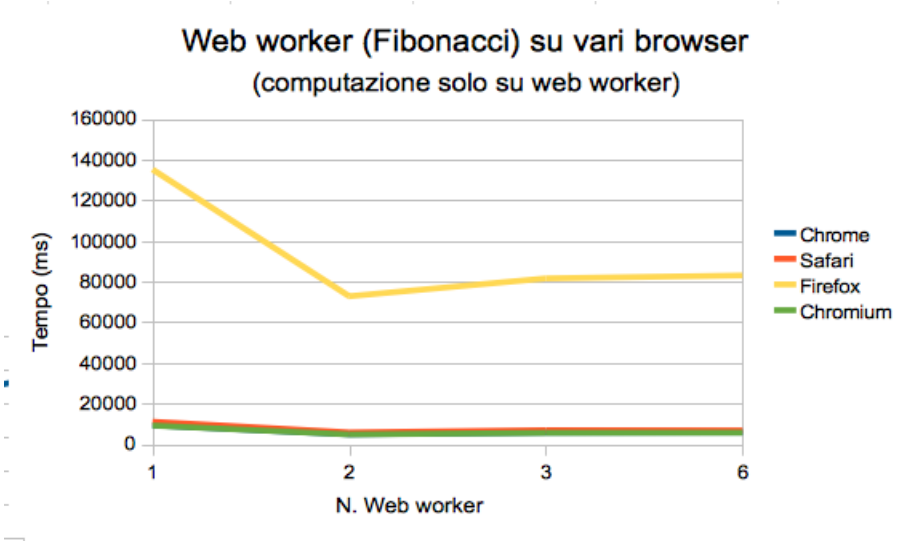

Figura 5.19: Grafico che mostra come l'esecuzione dell'algoritmo di Fibonacci in Javascript con l'utilizzo di web worker in Mozilla Firefox sia notevolmente dispendioso rispetto agli altri browser

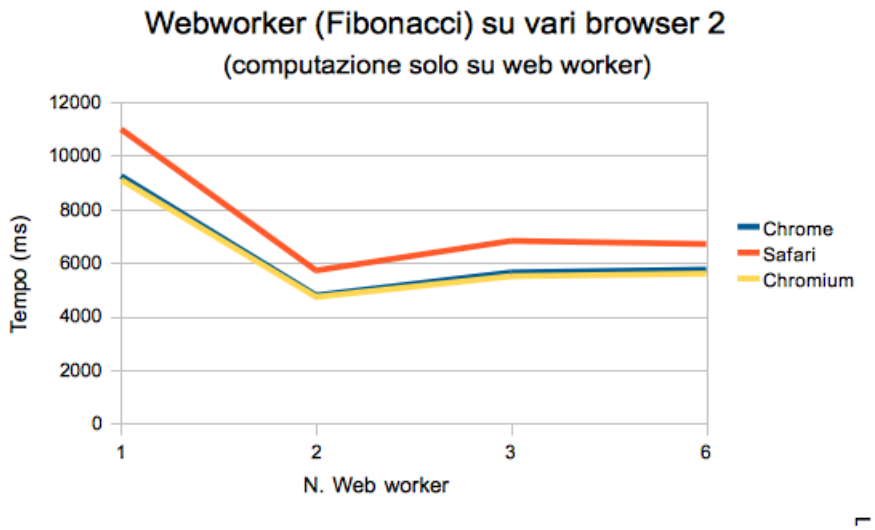

Figura 5.20: Grafico che mostra come l'esecuzione dell'algoritmo di Fibonacci in Javascript con l'utilizzo di web worker sia migliore con l'utilizzo di 2 web worker (in Google Chromium)

l'esecuzione era più veloce (per tutti Chromium). **Si può vedere come *Dart* con un isolate sia più veloce del 45 percento in più di quella di *Javascript* con web worker e del 183 percento in più di quella di *Javascript* sequenziale**.

Linguaggio	Computazione	Browser	Tempo	Differenza (%)
Javascript	Sequenziale	Chromium	9174	183,41%
Javascript	2 web worker	Chromium	4720	45,81%
Dart	1 isolate (83,33% su main)	Chromium	3237	ref.

Figura 5.21: Tabella che raccoglie i dati delle prove effettuate con l'algoritmo di Fibonacci prendendo in considerazione solo le implementazioni che forniscono la migliore esecuzione (su il browser dove la risposta era più veloce)

In 5.23 e 5.24 si considerano solo le medie delle implementazioni migliori dei tre linguaggi eseguite su tutti i browser. Si può vedere come *Dart* con un isolate sia più veloce del 585 percento in più di

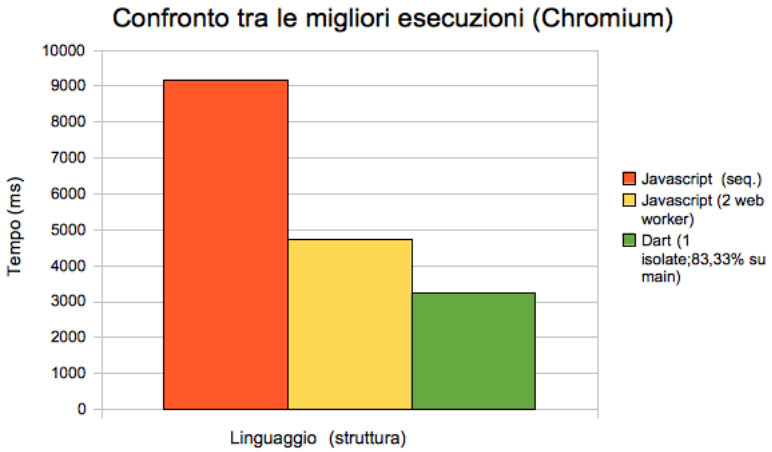

Figura 5.22: Grafico che mostra come l'esecuzione più veloce dell'algoritmo di Fibonacci sia quella di Dart (alla pari con le migliori esecuzioni degli altri linguaggi)

quella di *Javascript* con web worker (questo dovuto all'alto tempo di esecuzione su Mozilla Firefox) e del 200 percento in più di quella di *Javascript* sequenziale.

Linguaggio	Computazione	Tempo	Differenza (%)
Javascript	Sequenziale	9733	200,68%
Javascript	2 web worker	22196,51	585,71%
Dart	1 isolate (83,33% su main)	3237	ref.

Figura 5.23: Tabella che raccoglie le medie delle prove effettuate con l'algoritmo di Fibonacci prendendo in considerazione solo le implementazioni che forniscono la migliore esecuzione (su tutti i browser)

Dato che le medie esecutive dei web worker su Mozilla Firefox sono molto alte, riproponiamo la tabella ed il grafico senza tenere conto di quello specifico tempo.

In 5.25 e 5.26 si può notare come *Dart* con un isolate sia più veloce del 65 percento in più di quella di *Javascript* con web worker.

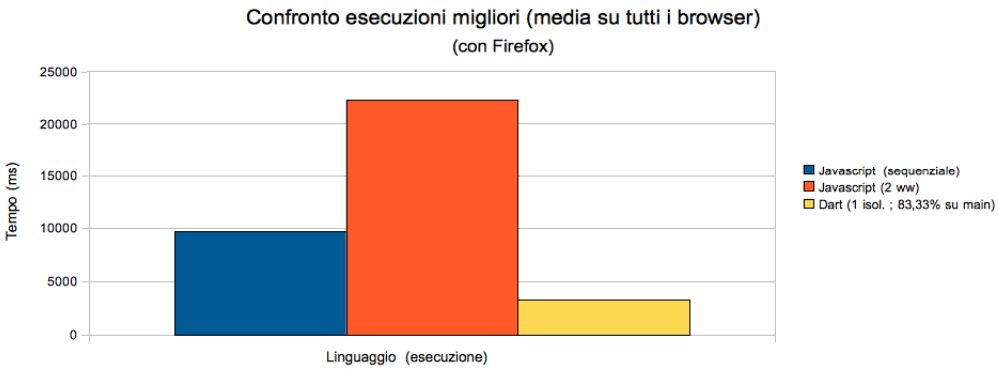

Figura 5.24: Grafico che mostra come l'esecuzione più veloce dell'algoritmo di Fibonacci sia quella di Dart (alla pari con le medie delle migliori esecuzioni degli altri linguaggi su tutti i browser)

Linguaggio	Computazione	Tempo	Differenza (%)
Javascript	Sequenziale	9733	200,68%
Javascript	2 web worker	5365,68	65,76%
Dart	1 isolate (83,33% su main)	3237	ref.

Figura 5.25: Tabella che raccoglie le medie delle prove effettuate con l'algoritmo di Fibonacci prendendo in considerazione solo le implementazioni che forniscono la migliore esecuzione (su tutti i browser tranne Firefox)

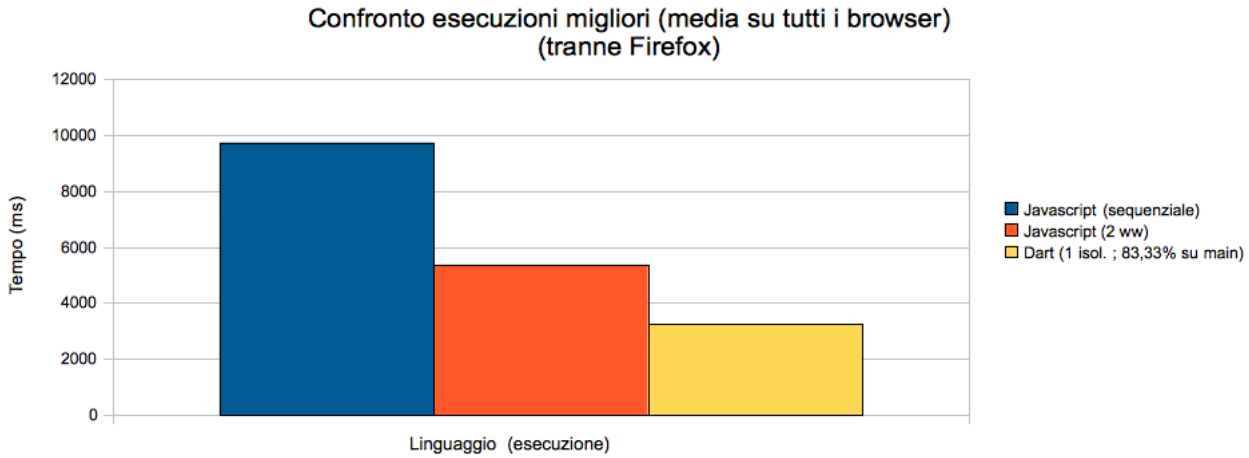

Figura 5.26: Grafico che mostra come l'esecuzione più veloce dell'algoritmo di Fibonacci sia quella di Dart (alla pari con le medie delle migliori esecuzioni degli altri linguaggi su tutti i browser tranne Firefox)

Riassumendo, i vari risultati mostrano come l'esecuzione dell'algoritmo di Fibonacci con *Dart* ed *isolate* risulti migliore sia di un esecuzione sequenziale di *Javascript* (presa sul miglior browser[2]) sia di un esecuzione di *Javascript* e *Web worker* (presa sul miglior browser). In particolare ***Dart* con *isolate* risulta circa il 45 percento più veloce di *Javascript* con *Web worker* e più del 183 percento di *Javascript* sequenziale**. La situazione è ancora migliore per *Dart* **considerando una media di esecuzione su tutti i browser dei suoi concorrenti, infatti *Dart* con *isolate* risulta circa il 65 percento più veloce di *Javascript* con *Web worker* e più del 200 percento di *Javascript* sequenziale (non considerando Mozilla Firefox, che incrementerebbe la percentuale di 9 volte circa per il caso web worker, e bloccherebbe addirittura il caso sequenziale)**.

[2]Si è utilizzato per il confronto l'esecuzione migliore presa tra diversi browser quali: Google Chrome, Apple Safari, Mozilla Firefox e Google Chromium.

Capitolo 6

Conclusioni e sviluppi futuri

L'analisi ed i diversi test effettuati sul linguaggio *Dart* e su gli *isolate* hanno permesso di comprendere in primis, che le caratteristiche del modello ad attori sono praticamente tutte implementate in *Dart*, tranne qualche piccolo particolare che risulta differente. Gli *isolate* sono l'astrazione che mappa gli attori, essi hanno uno stato non condiviso, modificabile tramite messaggistica asincrona.

Le primitive *create*, *send to* e *become* sono presenti in *Dart* con il nome di *spawn*, *send* e la *became* per modificare il comportamento è effettuabile tramite la scrittura di codice dentro la primitiva *receive*. Le mailbox sono presenti anche in *Dart*, però in questo caso, a differenza del modello ad attori originale, ogni attore ne può avere a disposizione più di una. E' stato verificato tramite test che questa possibilità non modifica le proprietà del modello di partenza, in quanto il flusso di controllo che preleva un messaggio da una delle caselle ed esegue l'operazione correlata è unico, quindi l'esecuzione dei metodi risulterà sempre atomica e l'accesso allo stato di un isolate sarà sempre *safe*. A differenza sempre del modello originale, gli attori non comunicano tramite la conoscenza del nome dell'attore destinatario, ma piuttosto tramite riferimenti alle mailbox. I tipi di riferimenti ad una casella sono due, cioè il riferimento in lettura (*ReceivePort*) che sarà quello utilizzato dall'attore per leggere i suoi messaggi e il riferimento per spedire un messaggio ad un attore (*SendPort*). Tutto ciò è stato implementato per un fattore di sicurezza, infatti un isolate dovrebbe ricevere i suoi messaggi e non poter vedere quelli degli altri.

Per verificare che questa scelta progettuale non generasse alcun tipo di problema, sono stati eseguiti dei test, che hanno portato alla scoperta di un bug relativo alla possibilità di ricevere messaggi relativi ad un altro attore. La ricezione di un messaggio è effettuata tramite l'utilizzo della primitiva *receive* sulla propria *ReceivePort*. Nella *receive* viene dichiarata una callback che sarà poi richiamata quando un messaggio arriverà. Per quanto riguarda la creazione di un *isolate*, in *Dart* esistono due tipi di costruttore: *light* e *heavy*. Entrambi creano un isolate con uno stato nuovo ovviamente non condiviso e possono solo comunicare in maniera asincrona via porte (tutto come detto in precedenza). Attraverso una serie di test si è potuto verificare come effettivamente gli isolate *heavy* vengano eseguiti concorrentemente, mentre gli isolate *light* no. Inoltre abbiamo testato come la differenza tra isolate *heavy* e *light* esista solo nell'esecuzione su *Dartboard* (che viene tradotta in Javascript con l'utilizzo anche di web worker) e non in *Cromium/Dartium* dove essi si comportano nella stessa maniera (cioè come gli *heavy* isolate).

Abbiamo poi verificato quelle che sono le proprietà di *encapsulation* e *fairness* che dovrebbe supportare un framework ad attori. Per quanto riguarda l'*encapsulation* essa risulta garantita in quanto non è possibile modificare lo stato dell'isolate in modi differenti dal semplice utilizzo di operazioni richiamate da messaggi, non è possibile effettuare passaggi di attributi per riferimento nei messaggi, non è possibile avere uno stato condiviso e a meno di bug, non è possibile leggere da mailbox altrui. Per quanto riguarda invece la proprietà di *fairness*, secondo i nostri test, essa è garantita solamente tramite l'utilizzo di isolate *heavy*. Gli isolate *light*, visto che sono eseguiti sullo stesso thread, vengono schedulati si in sequenza, ma solo quando ognuno di loro ha terminato di leggere ed eseguire tutti i messaggi nella mailbox. Questo significa che se uno di loro esegue delle operazioni potenzialmente infinite, tutti gli altri finiscono in *starvation*.

Nell'ultima parte della tesi abbiamo testato se le performance di *Dart* con gli *isolate*, fossero migliori di quelle di *Javascript* (sia sequenziale e sia concorrente con l'utilizzo di *web worker*). I vari risultati mostrano come l'esecuzione dell'algoritmo da noi utilizzato (algoritmo di Fibonacci), con Dart ed isolate risulti migliore sia di un esecuzione sequenziale di Javascript (presa sul miglior browser) sia di un ese-

cuzione di Javascript e Web worker (presa sul miglior browser). In particolare Dart con isolate risulta circa il 45 percento più veloce di Javascript con Web worker e più del 183 percento di Javascript sequenziale. La situazione risulta ancora migliore per Dart considerando una media di esecuzione dei suoi concorrenti, su tutti i browser. Infatti Dart con isolate risulta circa il 65 percento più veloce di Javascript con Web worker e più del 200 percento di Javascript sequenziale (non considerando Mozilla Firefox, che incrementerebbe la percentuale di 9 volte circa per il caso web worker, e bloccherebbe addirittura il caso sequenziale).

Dart è un linguaggio nato da pochissimo tempo ed è tutt'ora in fase di implementazione. Per questo motivo le primitive cambiano spesso (a volte anche nel giro di settimane) e quindi risulta molto complesso effettuare test e cercare di capire il modello che si trova dietro a questo linguaggio. Per quanto detto però, esso sembra implementare correttamente il modello ad attori con le sue caratteristiche e proprietà, anche se ancora l'interazione con il *DOM* e gli isolate non è ancora stata ben definita. Un possibile sviluppo futuro di questa tesi, potrebbe essere appunto quello di verificare, quando si avranno a disposizione nuove primitive, come queste entità possano interagire, con particolare attenzione alle proprietà di encapsulation ed esecuzione atomica dei metodi, che non devono venire compromesse.

Appendice A

Pattern fondamentali per la risoluzione di problemi concorrenti

A.1 Pipeline concurrency

La pipeline concurrency si ha quando un problema può essere risolto utilizzando una pipeline, cioè una struttura composta da più elementi (stage) messi in serie. Ognuno di questi riceve in ingresso un informazione, la elabora e poi trasmette il risultato al successivo stage della pipeline. Dal momento che tutte gli elementi che compongono la pipeline possono lavorare in contemporanea, si può raggiungere la concorrenza. Un esempio di questo pattern può essere il cosiddetto *prime sieve*[1] oppure il cosidetto *image processing network* nel quale una serie di immagini è fatta passare attraverso una serie di filtri o stage di trasformazione, e l'output finale è l'immagine processata.

[1]Per generare tutti i numeri primi fino ad un certo N, è possibile creare un generatore di numeri (da zero a N) e una pipeline formata da stage che dividono il numero che gli viene passato in ingresso per un numero fissato contenuto al loro interno (un numero primo). Se il numero passato è divisibile per il numero primo contenuto, allora viene scartato (non è un numero primo), altrimenti viene passato allo stage successivo. Se un numero arriva in fondo alla pipeline, allora è un numero primo e quindi verrà aggiunto un nuovo stage alla fine della stessa che contiene quest'ultimo.

A.2 Divide and conquer concurrency

La divide and conquer concurrency può essere espressa come un insieme di funzioni che risolvono i sottoproblemi in cui è possibile scomporre il problema di partenza. Esse sono valutate concorrentemente ed i loro valori in uscita sono collezionati per determinare il risultato finale. E' importante sottolineare come non ci sia alcuna interazione tra le procedure che stanno risolvendo i sottoproblemi. Un esempio di questo pattern può essere il calcolo del prodotto di tutti i numeri contenuti in un array. E' possibile utilizzare una procedura che divide a metà l'array che gli viene passato in ingresso. Essa restituisce in uscita il prodotto tra il risultato della stessa funzione chiamata con la parte sinistra dell'array e il risultato sempre della stessa con la parte destra, se invece gli viene passato un numero, lo ritorna. In questa maniera possono sussistere diverse funzioni che agiscono concorrentemente, occupandosi di parti diverse dell'array (cioè di parti diverse del problema).

A.3 Cooperative problem-solving

Per quanto riguarda il cooperative problem-solving, sono presenti diverse entità che effettuano ognuno la loro computazione, con la possibilità di comunicare in qualsiasi momento con le altre entità (ad esempio per condividere risultati intermedi che sono stati computati). Un esempio di utilizzo di questo pattern si potrebbe trovare nella simulazione di sistemi fisici, come ad esempio l'evoluzione dinamica di corpi sotto l'influenza dei campi gravitazionali creati da loro stessi.

E' interessante notare come nel cooperative problem-solving, sia possibile un trasferimento di informazione in qualsiasi momento, mentre nella pipeline concurrency e nel divide e conquer concurrency esso accada solo al termine della computazione.

Appendice B

Strutture per l'accesso a dati condivisi

B.1 I lock

B.1.1 Descrizione

I *lock* sono un meccanismi di sincronizzazione per limitare l'accesso ad una risorsa condivisa in un ambiente multitasking ad un solo thread o ad un solo tipo di thread alla volta. Concettualmente un *lock* è un oggetto di cui un thread deve venire in possesso prima di poter procedere all'esecuzione di una sezione protetta di un programma. Ogni processo che vuole accedere a dei dati condivisi, deve nell'ordine:

1. Acquisire il controllo del *lock* su quei dati.
2. Eseguire la computazione.
3. Rilasciare il *lock* (in modo tale che altri processi possano eseguire le loro operazioni su quei dati).

Quasi tutti i moderni sistemi operativi offrono un'interfaccia apposita per l'implementazione dei *lock* come parte delle API. In questo caso è il sistema operativo stesso che si preoccupa di gestire la coda tutti i processi che vogliono acquisire un *lock* bloccato. Occorre sottolineare che fino a quando i processi sono in coda, risultano fermi

(non eseguono altre operazioni). Esistono anche delle implementazioni alternative che prendono il nome di *spinlock* o anche *spinning*. Esse utilizzano un ciclo di attesa attiva (busy waiting) per implementare la routine di attesa di un *lock*. La tecnica consiste nel verificare periodicamente se il *lock* è stato sbloccato, effettuando un test che può aver luogo ad intervalli di tempo prestabiliti, oppure nel tempo più breve possibile consentito dal sistema.

B.1.2 Svantaggi

Sfortunatamente, sebbene i *lock* funzionino, viste le loro caratteristiche particolari pongono dei seri problemi nello sviluppo di applicazioni moderne, sia su CPU mono processore con multitasking che su infrastrutture multi-core, dove i problemi sono ancora di più accentuati data la natura caotica dello scheduling dei task su sistemi operativi multi-core. Di seguito viene riportata (e successivamente analizzata) una serie di queste annose problematiche che nascono dall'utilizzo dei *lock*:

- Componibilità che può portare a *deadlock* e *livelock*.
- Affidabilità dei programmatori (vulnerabili a insuccessi e a difetti, possibilità di *lockout*).
- Inversione di priorità.
- *Lock convoying*.
- Debug difficile.
- Prestazioni peggiori (overhead non necessario).

Un problema fondamentale è che i *lock* non sono componibili, infatti non è possibile prendere due pezzi di codice che ne fanno un uso corretto, e predirre se la loro esecuzione concorrente sarà anch'essa priva di problemi. Questo è ancor più problematico nello sviluppo di moderni software che fanno affidamento sulle librerie (che spesso sono anche composte le une con le altre) nelle quali non possiamo sapere senza esaminare la loro implementazione, se utilizzano o meno dei *lock*

al loro interno. I problemi nella composizione sono dovuti alla possibilità di *deadlock*, che accadono quando due *lock* devono essere acquisiti da due task in ordine opposto (vedi figura B.1).

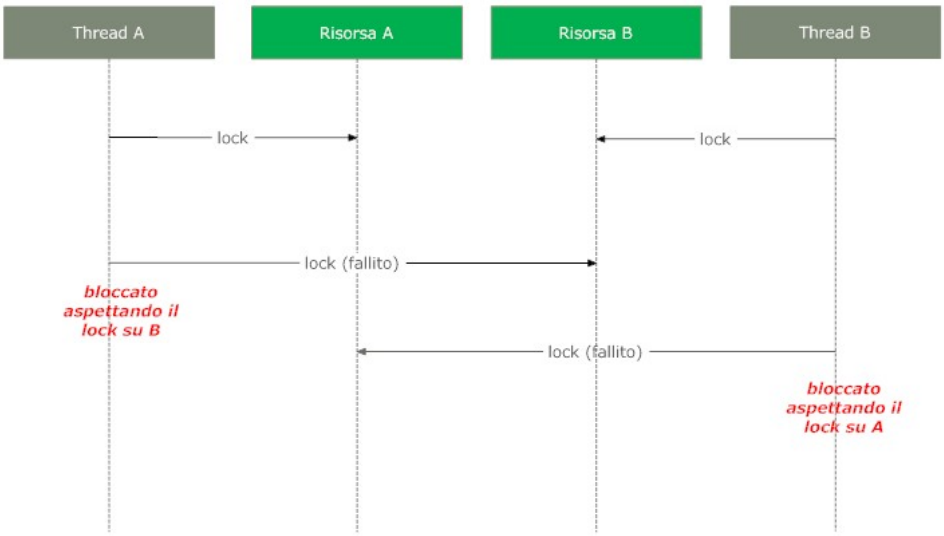

Figura B.1: Un esempio di Deadlock

Entrambi i task risulterebbero bloccati per sempre. Dato che questo può succedere ogniqualvolta due lock sono presi in ordine opposto, chiamare del codice non conosciuto (come una libreria) mentre si ha il controllo di un lock, è la formula giusta per avere un *deadlock*. Questo è quello che fanno esattamente i framework estensibili altamente specializzati come ad esempio il framework *.NET* e le *Java standard library*, che chiamano delle funzioni mentre hanno il controllo di un *lock*. Ancora non ci siamo scontrati con questo problema poichè i programmatori non stanno scrivendo tonnellate di codice concorrente che utilizzi frequentemente il *locking*. Molti modelli concorrenti provano a trattare il problema dei *deadlock* - con il protocollo *backoff-and-retry* ad esempio - ma spesso introducono loro stessi dei problemi, come ad esempio i *livelock*.

Un *livelock* è simile ad un *deadlock* eccetto per il fatto che lo stato dei processi coinvolti in un *livelock* cambia costantemente e nessuno però fa progessi. Un esempio nella vita reale potrebbe essere quello di

due persone che si incontrano in un corridoio stretto. Entrambi, essendo educati, si spostano da una parte per fare passare l'altro, peccato che lo facciano sempre dalla stesso e quindi possono andare avanti a muoversi per un tempo indefinito, senza che nessuno dei due riesca a passare. Una possibile soluzione potrebbe essere quella di assicurare che un solo processo (in maniera casuale o attraverso delle priorità) faccia le sue azioni, mentre l'altro aspetta. Il *livelock* è un rischio che è presente in certi algoritmi che riconoscono e cercano di recuperare da un *deadlock* in corso, come accade nella situazione descritta nell'immagine B.2, infatti al punto 1 l'acquisizione del lock fallisce in entrambi i thread, e tutti e due temendo una situazione di *deadlock* rilasciano i lock precedentemente acquisiti (punto 2). In un successivo momento (punto 3) i thread rieseguono lo stesso pattern per acquisire i lock, incappando sempre nella stessa problematica, che non avrà mai fine a meno che uno dei due non riesca a prendere entrambi i lock, prima dell'altro. E' interessante notare come in queste tecniche venga comunque utilizzato una modalità per sbloccare i processi in coda sui lock quando ci si accorga della possibile presenza di un *deadlock*.

Le tecniche per evitare i *deadlock* si basano sulla impostazione di un ordine nell'acquisizione dei lock. Se infatti i due lock visti in precedenza fossero stati presi nello stesso ordine (ad esempio prima i lock A per entrambi e poi, solo dopo l'acquisizione da parte di uno dei due, prendere il B) non ci sarebbero stati problemi. Per questo motivo possono essere usate delle tecniche come il *lock leveling* (o *lock hierarchies*) che suddividono i lock in livelli, in modo tale che ad esempio i lock di un livello possano essere presi solo in un determinato ordine, oppure che, se si sta mantenendo il lock di un livello, sia possibile solo prendere il controllo di lock a livelli superiori. Questa tecnica funziona bene per un programma od un framework manutenuto ed utilizzato da uno stesso team, ma è molto problematico nelle situazioni in cui vengono messi insieme pezzi di codice scritte da parti differenti (framework estensibili, add-on) a meno che non si sappia con certezza quali sono i livelli e soprattutto come vengono usati all'interno del codice.

Un problema più terra a terra con i lock è che essi fanno affidamento sulla correttezza del lavoro svolto dai programmatori, che devono eseguire strettamente le convenzioni per non andare incontro ad ulte-

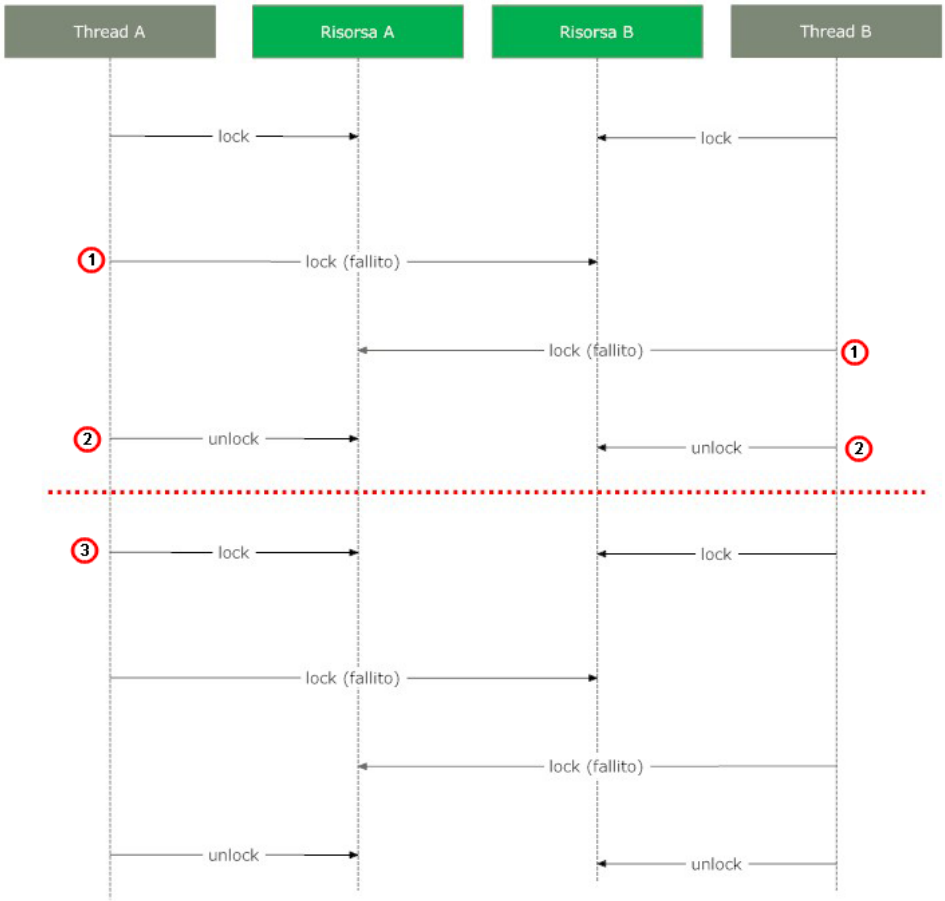

Figura B.2: Un esempio di Livelock

riori problemi. Infatti è il programmatore che decide quali parte del codice deve essere eseguita tra le primitive *lock* ed *unlock*, per questo motivo esso si deve sempre ricordare di prendere il giusto lock quando decide di operare su dati condivisi e soprattutto di rilasciarlo quando si ha finito. Se ci si dimentica di sbloccare il lock, qualunque task che provi ad prendere il controllo della memoria in questione non riuscirà nel suo intento, dato che il lock è ancora *locked* e quindi risulterà bloccato o letteralmente morto se visto dall esterno. Quest'ultima situazione prende il nome di *lockout*.

Inoltre un altro problema insidioso è che tutto il codice che accede ad un pezzo di memoria condivisa deve sottostare ad un particolare lock, così facendo anche thread ad alta priorità possono rimanere in attesa (*inversione di proprietà*), nonostante il buon senso direbbe di far rilasciare il lock per farli accedere.

Si può andare incontro al codiddetto *lock convoying* quando un processo che è padrone di un lock subisce de-scheduling (ad esempio in conseguenza di page fault, di interrupt, per avere esaurito il suo quanto di tempo, etc...) e durante questa interruzione, altri processi che sarebbero capaci di girare possono essere resi incapaci di procedere per l'impossibilità di impadronirsi del lock.

Quando proviamo a fare il debug di un applicazione concorrente, non sempre alcuni problemi come i *deadlock* possono venire alla luce, dato che dipendono fortemente dall'interleaving del codice. Per questo è molto difficile trovare gli errori e correggerli.

Nel caso in cui i lock vengano implementati da API del sistema operativo occorre che quest'ultimo si preoccupi di gestire tutte le code (su ogni operazione di ogni lock) che ovviamente è un lavoro molto oneroso in termini di tempo. Nel caso degli *spinlock* il calo delle performance invece è dovuto ai possibili busy-waiting. In entrambi i casi comunque c'è un sensibile peggioramento rispetto ad un sistema dove i lock non sono usati. Usare i lock è comunque un approccio conservativo, perché ogni thread deve acquisire il lock ogni qualvolta che vi è una possibilità di conflitto, che è situazione abbastanza rara nelle esecuzioni reali. Questo induce un grande overhead praticamente mai necessario, con consequente degradazione delle prestazioni.

Un tipo di dispositivo di sincronizzazione molto popolare può essere quello di *Java* che permette di modificare lo stato condiviso di un

oggetto tramite l'utilizzo di metodi particolari che possono essere richiamati solo dopo aver preso il lock per l'oggetto in questione, stiamo parlando dei metodi *Synchronized*. E' importante ricordare come una volta preso il lock su un oggetto, nessun altro processo può richiamare metodi *Synchronized* prima che esso venga rilasciato. Se l'oggetto è creato in modo tale che il suo stato (campi) venga modificato solo tramite metodi *Synchronized* allora questo approccio funziona.

Esistono però almeno tre problemi che nascono dall'utilizzo di queste nuova entità:

- Deadlock.
- Locking eccessivo.
- Inconsistenza dei dati condivisi per locking non sufficiente.

Una situazione di *deadlock* può comunque verificarsi se si chiama del codice scritto da terze parti all'interno di un metodo *Synchronized*, magari mantenendo altri lock, poichè per lo stesso motivo visto in precedenza non possiamo sapere che lock verranno acquisiti all'interno di questo codice.

Il secondo punto fa riferimento al fatto che quando un metodo di una classe è *Synchronized*, lo saranno ovviamente anche tutti gli oggetti derivati da questa. Ciò significa che nonostante la maggiorparte di essi non saranno neanche condivisi, occorrerà sempre prendere il lock e rilasciarlo ogniqualvolta si voglia eseguire un metodo contrassegnato con *Synchronized*.

L'ultimo punto sembra quasi un controsenso dato che nel precedente si parla di locking eccessivo. Purtroppo però ci sono situazioni in cui il semplice utilizzo di *Synchronized* non è sufficiente a mantenere sempre consistenti i dati condivisi. Ad esempio quando si chiamano due diversi metodi *Synchronized* su oggetti differenti o magari sullo stesso oggetto, che necessiterebbero di essere eseguiti in maniera complessivamente atomica.

Listing B.1: Esempio di inconsistenza dei dati condivisi per locking non sufficiente

```
conto1.accredita(ammontare);
conto2.preleva(ammontare);
```

In questo caso ci dovrebbe essere un trasferimento di denaro dal *conto1* al *conto2*, dove le operazioni *accredita* e *preleva* sono entrambe singolarmente *Synchronized*. Nessuno vieta però che ci sia un interleaving tale per cui un ulteriore processo legga lo stato del conto tra le due operazioni che in quel momento è assolutamente inconsistente. Per operazioni di questo tipo, la cui atomicità non corrisponde ad una singola chiamata di un metodo, occorre una sincronizzazione addizionale esplicita e quindi il nostro *Synchronized* da solo non basta.

B.1.3 Alternative

Come visto in precedenza (vedi B.1.1), quando scegliamo di utilizzare i lock, i progettisti, prima identificano le sezioni critiche del codice e poi le proteggono, scegliendo uno dei meccanismi tradizionali. In passato, evitare l'utilizzo dei lock poteva sembrare pericoloso o tendente a involvere intricati e convoluti algoritmi, per queste ragioni la programmazione senza l'utilizzo dei lock non è stata molto praticata. Ora però ci troviamo di fronte ad una serie non indifferente di problemi che nascono proprio dall'utilizzo dei lock e che sono ancor di più incidenti su infrastrutture multi-core rispetto alle singole CPU con multithreading data la natura caotica dello scheduling dei task su sistemi operativi multi-core. Basti pensare al fatto che su una singola CPU viene in ogni istante eseguito un singolo flusso di codice, mentre su una infrastruttura multi-core ne vengono eseguite un numero pari al numero di core, quindi le probabilità di un interleaving maligno su dei dati condivisi è molto più alto. Ora visti i numerosi svantaggi che l'utilizzo dei lock portano, ci chiediamo se sia possibile avere una condivisione delle risorse efficiente senza l'utilizzo di lock in un ambiente multicore. La risposta è che sistono due pricipali alternative ai lock e sono:

- *operazioni lock-free* dette *CAS (atomic compare-and-swap)*
- *transactional memory*

B.2 Compare and swap (CAS)

Le *CAS* sono operazioni offerte sia a livello hardware in varie infrastrutture multicore che a livello software in sistemi operativi multicore. Il modello di operazione è molto semplice ed è il seguente:

Listing B.2: Esempio di *CAS* in pseudocodice

```
CAS( variabile, valore_atteso, nuovo_valore){
  def var_loc;
  sezione_atomica{
        var_loc = variabile;
        if( var_loc == valore_atteso)
              variabile = nuovo_valore;
        }
  }
  return var_loc;

}
```

Quello che fa una *CAS* è di comparare il valore corrente di una variabile con un valore atteso. Se essi sono uguali allora viene modificato il valore della variabile con il nuovo valore, altrimenti no. Tutto questo, come ci suggerisce la keyword *sezione_atomica* viene fatto in una singola azione atomica quindi non interrompibile e nonpreentable.

In pratica per utilizzare una operazione *CAS* occorre procedere in questa successione ordinata:

1. Copiare il valore di una variabile condivisa in una variabile locale (*valore_atteso*).

2. Utilizzare una seconda copia di questo valore per portare a termine dei calcoli (anche complessi) che ci forniranno un nuovo valore locale da attribuire, se tutto andrà bene, alla variabile condivisa (*nuovo_valore*).

3. Confrontare il valore locale (*valore_atteso*) con una nuova lettura della variabile condivisa. Se i due valori sono uguali allora aggiornare la variabile condivisa con il nuovo valore (*nuovo_valore*) = utilizzo dell'operazione *CAS*.

Durante il secondo punto, un altro task potrebbe aver modificato la variabile condivisa e quindi tutti i nostri calcoli sul valore nuovo da dare ad essa porterebbero ad un valore inconsistente della stessa nel momento in cui andremmo ad aggiornarla (corsa critica). Infatti se ad esempio eseguiamo con due processi differenti, un incremento ed un decremento, su una variabile inizialmente nulla, potremmo trovarci nella situazione di avere, alla fine di tutte le computazioni, un valore non nullo (sbagliato) della stessa:

Listing B.3: Esempio tipico di problema nell'interleaving di istruzioni di thread differenti su variabile condivisa

```
C=0;

Thread A: Leggo C (punto 1)
Thread B: Leggo C (punto 1)
Thread A: Incremento valore locale di C;
localmente il risultato è 1. (punto 2)
Thread A: Memorizza
il risultato in C; C ora è 1.
Thread B: Decremento valore locale di C;
 localmente il risultato è -1. (punto 2)
Thread B: Memorizza
il risultato in C; C ora è -1.
```

E' importante quindi che dal momento in cui una variabile condivisa viene letta a quando viene aggiornata, il suo valore non venga modificato da un altro processo, è per questo motivo che controlliamo il suo valore attuale con quello letto la prima volta (punto 3), prima di decidere se modificare in maniera definitiva la variabile, è quindi per questo che viene usata l'operazione *CAS*.

Se la *CAS* mostra che la variabile condivisa non contiene più il valore atteso, allora essa non verrà modificata ed il software deciderà cosa fare in seguito. Quello che viene fatto più spesso è di ripetere l'intera sequenza di operazioni (punti 1,2 e 3) fino a quando la *CAS* non darà esito positivo ed il valore della variabile condivisa sarà correttamente aggiornato dal nostro processo.

Ecco di seguito un esempio completo di utilizzo di *CAS* per eseguire correttamente l'incremento di una variabile condivisa, senza l'utilizzo

di lock:

Listing B.4: Esempio completo di utilizzo di *CAS* per eseguire correttamente l'incremento di una variabile condivisa

```
int old, new, retVal;
 do{
  old = counter;// (punto 1)
  new = old + 1;// (punto 2)
  retVal = CAS(counter, old, new);
  (punto 3)
 }
 while(old != retVal);
```

Ci sono da fare due considerazioni fondamentali, guardando questo esempio:

- Il codice è molto più complesso e delicato della sua variante con i lock, in quanto sarebbe stato solo necessario acquisire il lock sull'oggetto che contiene il campo, eseguire l'incremento ed infine rilasciare il lock.
- iIl comportamento in loop può rendere il tempo esecutivo incontrollabile (se il *CAS* non verifica l'uguaglianza a causa di altri processi che modificano la variabile) e quindi non rende adatto il loro utilizzo in sistemi real-time.

Bibliografia

[1] Gul Agha *Concurrent Object-Oriented Programming*, Communications of the ACM Vol.33 N.9, September 1990.

[2] Herb Sutter, James Larus *Software and the concurrency revolution*, Microsoft Studios, September 2005.

[3] Rajesh K. Karmani, Amin Shali, Gul Agha *Actor Frameworks for the JVM Platform: A Comparative Analysis*, Open System Laboratory, University of Illinois, August 2009.

[4] Rajesh K. Karmani, Gul Agha *Actors*, Open System Laboratory, Department of Computer Science Univesity of Illinois.

[5] *Dartlang - Dart official site* `http://www.dartlang.org/`.

[6] *Google Dart discussion group* `https://groups.google.com/a/dartlang.org/group/misc/topics?start=0&sa=N`.

[7] *Chris Strom Blog - a blog on Dart* `http://japhr.blogspot.it/`.

[8] Carlos Varela, Abe Stephens, *SALSA: Language and Architecture for Widely Distributed Actor Systems.*, Department of Computer Science, Rensselaer Polytechnic Institute, Troy NY, USA.

[9] Carl Roger Manning *Acore: The Design of a Core Actor Language and its Compiler*, Computer Science Massachusetts Institute of Technology, May 1985.

[10] *Rosette Developing InfoSleuth Agents Using Rosette: An Actor Based Language*, Darrell Woelk Microeclectronics and Computer Technology Corporation (MCC).

[11] *Separation of concerns* `http://aspiringcraftsman.com/2008/01/03/art-of-separation-of-concerns/`.

[12] *Edit-and-continue debugging* `http://msdn.microsoft.com/en-us/library/bcew296c(v=vs.80).aspx`.

Ringraziamenti

Il primo ringraziamento va alla mia famiglia che mi ha aiutato, sostenuto e sopportato per tutti questi anni. Senza di voi tutto ciò non sarebbe stato possibile.
A mio papà *Gliberto* perchè sei stato sempre un esempio per me, in tutte le cose che hai fatto. Ti ringrazio per avermi insegnato a non arrendermi di fronte alle difficoltà e a rispettare fino in fondo gli impegni presi. E soprattuto perchè sotto la tua corazza di autorevolezza, nascondi un cuore d'oro.
A mia mamma *Morena* perchè sei un esempio di disponibilità, di altruismo e di forza interiore. Ti ringrazio per avermi sempre aiutato e per i sacrifici che fai per la nostra famiglia. Quando ero in difficoltà mi sei sempre stata vicino, e anche se non te lo chiedevo, hai sempre capito come aiutarmi.
Ad entrambi perchè quando vi vedo chiudere il negozio all'una di notte e andare al mercato alle cinque di mattina il giorno dopo, capisco cosa significhi sacrificarsi per altre persone. Vi ringrazio per questo.
A mia sorella *Giulia* perchè riesci sempre a farmi sorridere. Hai un grande talento e sei molto intelligente. Spero che tu possa trovare presto la tua strada e spero che la vita ti riservi tante soddisfazioni.
A mia nonna *Albertina* perchè sei un esempio di vita. Ti ringrazio per tutto quello che hai fatto fino ad ora per me e anche perchè mi dimostri ogni giorno che si può riuscire a fare tutto, nonostante mille difficoltà.
Alla mia fidanzata *Antonietta* perchè non mi piaci solo esteriormente ma anche interiormente. Sei una persona forte e determinata (e lo sai), ma sai essere anche estremamente dolce. Mi hai dimostrato come l'amore vince sempre, nonostante tutti i problemi e nonostante la lontananza. Spero di stare tutta la vita con te, e anche le prossime se

ci saranno.
Ai genitori della mia fidanzata, *Laura* ed *Eugenio*, perchè mi avete accolto in casa vostra come se fossi vostro figlio. Siete due persone meravigliose e la vostra simpatia rende piacevole la mia permanenza lontano da casa.
Ai miei cugini *Claudio* e *Matteo* insieme ai miei compagni di merenda *Thomas*, *Luca* e *Denis* perchè sapete trasformare ogni volta un incontro settimanale in qualcosa di eccezionale. Siete amici da *venti puro*!
Un ringraziamento particolare va al mio relatore, *prof. Alessandro Ricci*, per aver dedicato tempo al mio progetto, ma soprattutto perchè grazie alla sua disponibilità, competenza ed entusiasmo ha reso la scrittura di questa tesi un piacevole lavoro.
Grazie ai miei amici di sempre *Andrea*, *Milos*, *Mattia* e *Alberto*, i quali mi hanno sempre accompagnato in momenti felici della mia vita e mi hanno sostenuto con la loro spontaneità ed amicizia.
Grazie al mio terapista ed amico *Nicola*. I tuoi consigli hanno sempre fatto in modo che la mia vita cambiasse in meglio.
Grazie infine al mio compagno di facoltà *Denis*, che ha condiviso con me un bellissimo percorso. Non dimenticherò mai le lezioni seguite insieme, i momenti di studio che grazie a te diventavano anche momenti di felicità. Ti ringrazio anche per tutte le volte che mi hai aiutato, sei stato davvero come un fratello. Spero che le nostre strade non si dividano dopo questo giorno perchè sarà veramente difficile trovare un altro compagno di lavoro e di studio così in gamba come te...e ora *passami il microfono!*
L'ultimo grazie va a me stesso, che sono arrivato fino in fondo a questa avventura. Ci sono stati momenti difficili ma abbiamo finalmente tagliato il traguardo...