ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Seconda Facoltà di Ingegneria con sede a Cesena
Corso di Laurea in Ingegneria Informatica

IL MODELLO AD ATTORI PER LA PROGRAMMAZIONE CONCORRENTE: IL LINGUAGGIO AXUM SU .NET COME CASO DI STUDIO

Elaborato in
SISTEMI OPERATIVI

Tesi di Laurea di
ANGELO CROATTI

Relatore
Prof. ALESSANDRO RICCI

Anno Accademico 2011 – 2012
I^{a} Sessione di Laurea

PAROLE CHIAVE

Modello ad Attori
Programmazione Concorrente
Scambio Asincrono di Messaggi
AXUM
.NET Framework

alla mia famiglia

Indice

Introduzione

Nell'ambito dei linguaggi di programmazione per lo sviluppo del software, l'avvento del paradigma orientato agli oggetti ha certamente rivoluzionato i meccanismi per la realizzazione di applicazioni.

Sebbene tale modello sia stato per molto tempo il riferimento per la progettazione e lo sviluppo del software in ambiti commerciali, accademici e di ricerca, è opinione diffusa, ad oggi, che alcune sue caratteristiche difficilmente possano adattarsi alle nuove esigenze del software in relazione all'evoluzione della tecnologia.

L'aspetto della programmazione concorrente, anche e soprattutto in relazione all'evoluzione dei dispositivi hardware, rappresenta sicuramente una delle più evidenti lacune del modello ad oggetti, evidenziando come sia necessario un nuovo modo per progettare il software. In relazione a ciò, inizialmente, il parallelismo all'interno delle applicazioni è stato introdotto definendo quei meccanismi che sono racchiusi nella cosiddetta programmazione multithread.

Effettivamente, la separazione dei processi e la loro possibile distribuzione ha enormemente ampliato le possibilità offerte dal modello ad oggetti in materia di concorrenza ed ha anche permesso una notevole evoluzione delle applicazioni in termini di flessibilità e performance.

Tuttavia, la programmazione multithread, mantenendo un approccio di base ad oggetti, continua a manifestare alcune lacune soprattutto per quanto riguarda l'aspetto della comunicazione tra i vari componenti delle applicazioni. Nonostante ogni thread abbia un flusso d'esecuzione separato dagli altri, le necessità di sincronismo e coordinazione portano spesso a dover limitare le performance per poter garantire la correttezza della computazione.

Per quanto detto, appare quindi evidente che il continuare ad introdurre nuovi aspetti e funzionalità, rispetto a quanto previsto dal modello ad oggetti originario, potrebbe non essere la strada giusta da percorrere.

Un'alternativa a questo approccio potrebbe essere rappresentata da un diverso modello di programmazione che nasce proprio per agevolare la programmazione concorrente: il modello ad attori.

Con una logica incentrata sul meccanismo di scambio asincrono di messaggi tra entità software chiamate attori, il modello ad attori riprende la proposta originaria del modello ad oggetti che inizialmente prevedeva una comunicazione basata sullo scambio di messaggi piuttosto che sul meccanismo di chiamata a procedura (o a metodo) con conseguente trasferimento del controllo.

Purtroppo, l'idea originaria fu accantonata da chi scelse di implementare il modello ad oggetti e fu favorita una logica più procedurale e simile a quanto già esisteva nei linguaggi disponibili all'epoca.

Il modello ad attori, nato poco dopo quello ad oggetti, mantenne invece l'idea di una comunicazione basata sullo scambio di messaggi e ciò, considerando la rapida evoluzione del modello ad oggetti grazie soprattutto alla sua efficienza e semplicità, fu probabilmente la causa di un iniziale basso apprezzamento del modello che restò così vincolato ai soli ambienti accademici e di ricerca.

Oggi invece, alla luce delle nuove richieste in relazione alle problematiche imprescindibili sollevate dalla concorrenza, il modello ad attori è fortemente rivalutato e molti sono i linguaggi che lo supportano o lo implementano completamente.

Naturalmente, come è avvenuto per il modello ad oggetti, anche le varie implementazioni del modello ad attori introducono qualcosa o modificano qualcos'altro rispetto al modello teorico originario.

Questo è inevitabile in quanto alcune problematiche pratiche che la teoria spesso non considera, emergono poi nel momento in cui dalla teoria si passa alla pratica. Ovviamente, però, le caratteristiche fondanti del modello restano e questo fa si che le varie implementazioni esistenti possano essere validi linguaggi di programmazione per rispondere alle questioni sollevate dalle sempre maggiori necessità in termini di concorrenza nell'attuale sviluppo delle applicazioni software.

Per questi motivi quindi, l'idea di sviluppare una tesi con l'obiettivo di analizzare e comprendere gli aspetti fondanti del modello ad attori e studiarne le possibili applicazioni, appare ben integrabile nel contesto tecnologico attuale.

Comprendere l'essenza del modello ad attori ed analizzarne le differenze rispetto al modello ad oggetti, risulta fondamentale per capire se lo sviluppo del software mediante linguaggi che implementano tale modello possa essere la risposta alle opportunità in termini di performance offerte dalle nuove disponibilità in ambito tecnologico (hardware e non solo).

Questa tesi, pertanto, si propone, dopo una panoramica generale sullo stato dell'arte in termini di software e concorrenza, l'obiettivo di studiare

il modello ad attori ed analizzarne alcune significative implementazioni.

Successivamente, sarà scelta una specifica implementazione, in particolare il linguaggio AXUM in ambiente .NET Framework, che sarà dettagliatamente discussa ed analizzata al fine di evidenziare se tale linguaggio rispetti innanzi tutto i principi del modello ad attori e quindi se possa risultare o meno una valida alternativa ai linguaggi attuali per lo sviluppo di software orientato alla concorrenza.

Sommario

La presente tesi, che studia e analizza il modello ad attori per la programmazione di applicazioni software concorrenti, è articolata in sei capitoli.

I primi quattro capitoli, con un percorso dal generale al particolare, affrontano il rapporto esistente tra software e concorrenza per poi dedicarsi allo studio delle caratteristiche del modello ad attori e ad alcune sue implementazioni.

Gli ultimi due capitoli, invece, si concentrano sul caso di studio scelto, ovvero sul linguaggio AXUM introdotto in ambiente .NET e su una sua analisi critica.

Capitolo 1 Nel primo capitolo, viene affrontato in modo generale il tema della concorrenza nello sviluppo del software e la sua importanza al giorno d'oggi. Analizzando l'evoluzione dei microprocessori, in special modo quella dei microprocessori multi-core di ultima generazione, si dimostra come l'aspetto concorrente delle applicazioni software non possa essere né sottovalutato né tanto meno tralasciato o ignorato.

Capitolo 2 Nel secondo capitolo, si introducono brevemente i concetti fondamentali della programmazione multithread, focalizzando principalmente l'attenzione sulla gestione dello spazio delle informazioni condivise e sui meccanismi per gestire la sincronizzazione. Sebbene la tesi si proponga di studiare un modello di programmazione alternativo alla programmazione di tipo multithread, introdurne alcuni aspetti risulta essere fondamentale per descriverne gli svantaggi che potrebbero favorire una migrazione da tale tipo di programmazione verso l'adozione di uno stile basato sul modello ad attori.

Capitolo 3 Il terzo capitolo rappresenta il cuore della tesi. In esso viene descritto ed analizzato il modello di programmazione ad attori. Si introducono i concetti principali e se ne descrivono le proprietà e le

caratteristiche. Vengono introdotti i meccanismi chiave del modello e i vantaggi da esso apportati. Inoltre, vengono anche analizzate le principali problematiche e discusse alcune interessanti osservazioni. Infine, vengono presentati alcuni esempi di applicazioni del modello ad attori nella realtà odierna; come l'applicazione per microblogging Twitter, il linguaggio per applicazioni web strutturate Dart e la piattaforma Orleans per il cloud computing.

Capitolo 4 Nel quarto capitolo sono presentate, descritte ed analizzate brevemente alcune implementazioni del modello. Per ognuna di esse vengono proposte le strategie utilizzate per la definizione ed analizzate le eventuali modifiche apportate alla descrizione originale del modello. Inoltre, sono riportati i gradi di performance offerti da ciascuna implementazione.

Capitolo 5 Il quinto capitolo entra nello specifico del caso di studio scelto. Esso infatti descrive il linguaggio AXUM, ne illustra le caratteristiche principali e ne analizza gli aspetti più significativi. Partendo da un'introduzione generale si passa a descrivere il meccanismo per lo scambio asincrono di messaggi e gli elementi base per la comunicazione. Quindi, vengono illustrati i componenti principali e le loro modalità di utilizzo, nonché strutture e componenti a corredo per aumentare la flessibilità del linguaggio. Successivamente, l'attenzione viene concentrata sulla realizzazione di applicazioni basate sul concetto di controllo gestito tramite i dati, che rappresenta probabilmente l'aspetto più interessante e innovativo di AXUM.

Capitolo 6 Il sesto ed ultimo capitolo affronta il delicato compito di effettuare un'analisi critica del linguaggio AXUM. In particolare vengono discussi tutti quegli aspetti del modello ad attori modificati nell'implementazione fatta da AXUM. Inoltre, vengono analizzati gli aspetti innovativi e ne viene valutato il grado di utilità. Infine, vengono espresse alcune considerazioni conclusive sul linguaggio in generale e sul modo adottato per implementare gli aspetti centrali ed identificanti il modello ad attori.

In definitiva, perciò, la tesi si sviluppa mediante un percorso che, partendo dall'analisi generali degli aspetti relativi alla programmazione concorrente e dall'attuale approccio ad essa, evolve considerando il modello

di programmazione ad attori e descrivendone caratteristiche, meccanismi e vantaggi d'uso.

Inoltre, l'analisi di alcune significative implementazioni consente di avere una visione ad ampio spettro relativamente a quanto già esiste relativamente all'ambito del modello ad attori.

Infine, il caso di studio considera una particolare implementazione del modello, descrivendola ed analizzandola in modo molto dettagliato al fine di garantire una comprensione tale da rendere immediatamente utilizzabile il linguaggio.

Capitolo 1

Il Software e la Concorrenza

Il problema della concorrenza nella progettazione del software è noto da tempo. Come sostengono i ricercatori H. Sutter e J. Larus nel loro articolo "*Software and the Concurrency Revolution*" [2], infatti, la concorrenza è da sempre definita come "*the next big question*" e "*the way of the future*". Ciononostante, ad oggi, pochi sono i linguaggi di programmazione che considerano la concorrenza elemento centrale su cui basare la propria logica di costruzione del software. La massiccia disponibilità di hardware prevalentemente orientato alla computazione sequenziale, inoltre, non ha certamente favorito un cambio di rotta in tal senso.

Tuttavia, l'avvento del processori multi-core a basso costo e la loro veloce integrazione in ogni genere di dispositivo, ha contribuito notevolmente a portare la questione della concorrenza nella progettazione e nello sviluppo del software ad una posizione di primaria importanza; rendendo, in tal modo, visibilmente inefficaci i meccanismi di gestione della concorrenza presenti nei più diffusi ed utilizzati linguaggi di programmazione.

Pertanto, continuare ad ignorare l'aspetto primario della concorrenza, sarebbe non solo deleterio per l'aumento della complessità nello sviluppo del software ma soprattutto controproducente poiché le nuove architetture multi-core, per la loro struttura intrinseca, hanno l'effetto di rallentare i tempi di esecuzione delle applicazioni progettate ed ottimizzate per i processori single-core.

In ogni caso, comunque, *pensare in modo concorrente non è immediato*! La mente umana è fisiologicamente portata a ragionare in modo sequenziale; definire meccanismi in grado di gestire al meglio il pieno parallelismo richiesto dalle nuove architetture non è, quindi, compito facile.

1.1 Aspetti generali

Come è avvenuto per l'introduzione del paradigma ad oggetti anche la concorrenza, considerata come elemento da cui non poter prescindere, è destinata a segnare la nascita di una nuova "era del software". Tuttavia, il percorso verso la completa progettazione e sviluppo di software concorrente non sarà semplice, né tanto meno breve. Infatti, sebbene la programmazione Object Oriented ponga oggettivamente più di qualche difficoltà, è dimostrabile che la programmazione concorrente porti a molte più problematiche ed interrogativi da risolvere.

A titolo d'esempio, si pensi ai sistemi client/server. Per le applicazioni server-side, il problema della concorrenza è pressoché risolto in quanto tali applicazioni sono per loro natura sovrabbondanti di parallelismo e perciò nascono con un inevitabile approccio concorrente. Per le applicazioni client-side, invece, l'aspetto concorrente è spesso ignorato. Le applicazioni di tale categoria, infatti, hanno generalmente un flusso d'esecuzione ben definito e regolare; per cui, pensare ad una progettazione in cui il flusso venga suddiviso in task paralleli non appare né semplice né immediato.

In ogni caso, comunque, poiché la concorrenza porta a performance migliori, lo sforzo per introdurre il parallelismo nella progettazione di qualunque applicazione non è mai vano.

Per tutti questi motivi, quindi, l'informatica non può esimersi dal definire o introdurre, nei linguaggi di programmazione, nuovi meccanismi che permettano di affrontare efficientemente ed agevolmente l'idea che la concorrenza sia elemento cardine nello sviluppo di qualunque applicazione, dalle più semplici alle più complesse ed articolate.

1.2 Microprocessori multi-core

A partire dagli anni novanta, i produttori di microprocessori hanno iniziato a studiare metodologie per incrementare la potenza di calcolo dei microprocessori esistenti in modo da superare il limite fisico che la tecnologia cominciava ad imporre.

I primi tentativi per inseguire la legge di Moore (*Le prestazioni dei microprocessori raddoppiano ogni 18 mesi*) furono incentrati sulla realizzazione di architetture a multiprocessore (SMP, *Symmetrical Multiprocessors*) tra le quali va ricordata l'architettura dell'Intel© Itanium.

Successivamente tuttavia, lo studio delle architetture per microprocessori si è spostato, anche e soprattutto grazie alle nuove possibilità

offerte dalle moderne tecnologie di realizzazione di circuiti integrati, verso l'idea di inserire in uno stesso chip più core che fossero in grado di lavorare parallelamente in modo da aumentare così il numero di operazioni eseguite per ogni ciclo di clock. Questo ha portato alla realizzazione dei cosiddetti microprocessori multi-core.

I vari core dei microprocessori multi-core hanno la particolarità di poter condividere tra loro alcuni componenti (la chache di secondo livello, ad esempio) aumentando così, rispetto ai multiprocessori, la velocità dello scambio di dati. In questo modo, le performance generali risultano decisamente migliori.

Nel 2005 Intel© ha presentato i primi processori dual-core che si sono immediatamente diffusi in modo capillare in tutti i tipi di computer. Nel 2008 sono stati presentati i primi processori quad-core, anch'essi accolti con enorme successo. Attualmente sono stati annunciati per la prossima distribuzione i processori eight-core.

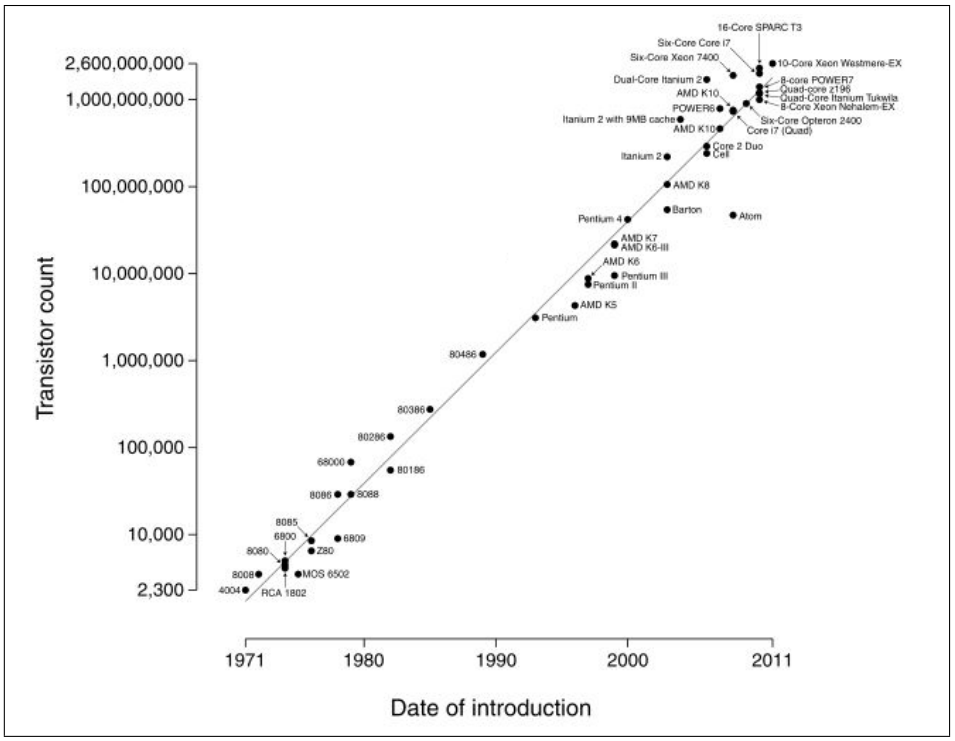

Figura 1.1: *Evoluzione dei microprocessori*

In Figura 1.1 è riportato un riepilogo dei vari microprocessori costruiti nel corso degli anni per i quali è evidenziato come la loro distribuzione

nel tempo segua la citata Legge di Moore[1]. In particolare, si noti la rapidissima evoluzione che hanno avuto i microprocessori multi-core negli ultimi 5 anni.

L'avvento dei microprocessori multi-core non ha avuto impatto solamente sul mercato dell'hardware ma anche e soprattutto su quello del software. Potendo sfruttare le potenzialità in termini computazionali offerte dai nuovi microprocessori, risulta ovvio ripensare al modo di costruire il software in funzione di un modello di programmazione concorrente.

In questo senso, quindi, si nota come l'aspetto della programmazione concorrente, che sia essa multithread o basata su altri meccanismi, non possa essere più evitato o accantonato nella progettazione di applicazioni software.

1.3 Una nuova rivoluzione nel software

Nella storia dei modelli di programmazione degli ultimi 30 anni, il maggiore cambiamento si è sicuramente verificato con l'introduzione del concetto di *oggetto* e il conseguente passaggio dalla programmazione strutturata a quella orientata agli oggetti [3]. Il passaggio è stato così radicale da poter essere etichettato come "rivoluzione del software", l'unica, fino ad ora, nella storia più recente del software.

Come detto al Paragrafo 1.2, se si vogliono sfruttare le nuove potenzialità offerte dai microprocessori multi-core, è necessario cominciare a pensare e a progettare le applicazioni in funzione di un modello concorrente. Al giorno d'oggi, purtroppo, le applicazioni vengono spesso ancora progettate ignorando l'aspetto concorrente e rendendole in tal senso molto meno veloci e performanti di quanto invece potrebbero esserlo se solo fossero state progettate per sfruttare completamente le caratteristiche delle CPU sulle quali verosimilmente saranno eseguite.

Per questi motivi, quindi, la concorrenza inevitabilmente costituirà la prossima grande rivoluzione nel modo di scrivere il software. Non è ancora chiaro se sia da considerarsi a maggiore o minore impatto di quella generata dalla programmazione ad oggetti; quello che è certo però, è che sarà una vera rivoluzione e dovrà coinvolgere tutte le varie fasi dello sviluppo del software, dall'ideazione alla realizzazione.

[1] Rappresentata in figura mediante una retta. Evidentemente quindi, la distribuzione è descritta su scala logaritmica in quanto la legge di Moore descrive un evoluzione in termini esponenziali.

1.3.1 Vantaggi e Costi dell'approccio concorrente

Come ogni novità, anche la concorrenza porta con se vantaggi che sono ben graditi e costi che invece si vorrebbero evitare. Tra i vantaggi, sicuramente sono da citare i seguenti:

- Separazione del controllo di flusso.
- Maggiore efficienza nella computazione.
- Migliori prestazioni del prodotto software finale.

Tra i costi, invece, è fondamentale citare innanzi tutto il fatto che non tutte le applicazioni possano essere pienamente parallelizzate. La metafora proposta da H. Sutter [3] per enfatizzare ciò, è decisamente calzante: *il fatto che ad una donna servano nove mesi per dare la vita ad un bambino, non implica che nove donne possano dare la vita ad un bambino in un solo mese.*

Al di la di questo aspetto, un altro costo decisamente importante della concorrenza deriva dal fatto che progettare e programmare applicazioni concorrenti risulta davvero difficile; soprattutto se ci si pone come termine di paragone lo sviluppo di applicazioni mediante la programmazione orientata agli oggetti. Pensare a flussi separati per la computazione spesso non è immediato.

1.4 Modelli per il parallelismo

Ad oggi, nei linguaggi di programmazione maggiormente diffusi (con particolare riferimento ai linguaggi Object Oriented), il concetto di parallelismo nell'esecuzione di istruzioni è già presente. Sfortunatamente, per come è stato implementato, è applicabile solamente a piccole sottocategorie di applicazioni. Spesso è difficile decidere, in modo automatico o meno, quale modello di parallelismo adottare per risolvere un particolare problema. Anche con un'attenta valutazione poi, non è sottinteso che si riesca a raggiungere un buon compromesso tra l'efficienza d'esecuzione in termini di tempo e la richiesta di risorse che dovrebbe essere esigua.

Inoltre, un altro punto da considerare quando si parla di parallelismo, è che maggiore è la percentuale di istruzioni (singole o suddivise in blocchi) eseguite parallelamente, maggiori saranno poi i costi, in termini computazionali, da affrontare per garantire il sincronismo e la condivisione delle informazioni.

In ogni caso, comunque, si possono brevemente introdurre tre diversi modelli di parallelismo che rappresentano l'approccio concorrente attualmente presente nei linguaggi Object Oriented.

Parallelismo indipendente

Rappresenta l'approccio più semplice di parallelismo ed è applicabile quando le diverse operazioni da eseguire non richiedono coordinazione e/o condivisione di informazioni tra loro. In questo caso, è immediato comprendere come implementare tale parallelismo. A titolo d'esempio, il parallelismo indipendente è applicabile agli algoritmi di *brute-force* in crittografia, agli algoritmi di ricerca in campo genetico, agli algoritmi per la determinazione dei valori dei frattali di Mandelbrot e a tanti altri.

Parallelismo strutturato

Rappresenta un approccio al parallelismo applicabile quando è necessario eseguire una operazione su tutti gli elementi di una collezione di dati e quando tale operazione risulta mutuamente dipendente dall'esecuzione delle altre. In tale approccio, è necessario disporre di meccanismi di sincronizzazione e di comunicazione tra le varie operazioni per garantire una strategia d'esecuzione che permetta di giungere a risultati corretti. A titolo d'esempio, si pensi al calcolo della convoluzione di un'immagine grayscale.

Parallelismo non strutturato

Rappresenta l'approccio classico al parallelismo effettuato mediante l'ausilio di thread. In questo caso, infatti è necessario un esplicito sincronismo affinché sia possibile scongiurare il non determinismo del risultato dell'esecuzione.

1.5 Concorrenza e Linguaggi

Sebbene si possa essere portati a pensare che l'approccio alla concorrenza nella programmazione dipenda solo dal modo di organizzare e progettare il software e non dal tipo di linguaggio utilizzato per realizzare l'applicazione, ciò non è del tutto vero. Infatti, alcuni tipi di linguaggi sembrano molto più adatti di altri per introdurre i meccanismi per la concorrenza.

La classificazione dei vari linguaggi di programmazione esistenti è piuttosto ampia. Per quanto riguarda la questione della concorrenza, tuttavia, è sufficiente considerare le due categorie di linguaggi maggiormente diffusi ed utilizzati: i linguaggi imperativi orientati agli oggetti e i linguaggi funzionali.

Linguaggi imperativi orientati agli oggetti

I maggiori linguaggi di programmazione commerciali ad oggi esistenti (tra cui spiccano Java e C#) sono linguaggi imperativi orientati agli oggetti. Un linguaggio è considerato imperativo quando un generico programma scritto in tale linguaggio è rappresentato da un insieme di istruzioni ordinate da impartire alla macchina esecutrice.

Tramite i linguaggi imperativi si ha il pieno controllo delle variabili e delle strutture dati, i cui cambiamenti di stato sono specificati mediante ben definite istruzioni.

Nei linguaggi imperativi orientati agli oggetti è possibile poi definire oggetti software in grado di incapsulare informazioni e definire funzioni per poter accedere a tali informazioni ed eventualmente modificarle.

Dal punto di vista della parallelizzazione automatica dei software scritti in tali linguaggi, è opinione comune che ciò sia molto difficoltoso. Infatti, caratteri dominanti di tali linguaggi come il controllo di flusso gestito da costrutti ben specifici, la manipolazione diretta dello stato delle variabili e la condivisione dello stato degli oggetti rendono molto complicata un'analisi automatica del software per comprenderne il funzionamento e conseguentemente suddividerlo in task paralleli.

Linguaggi funzionali

I linguaggi funzionali sembrano invece naturalmente adatti per la realizzazione di applicazioni pienamente concorrenti. Infatti, manipolando unicamente dati immutabili, tali linguaggi non introducono, apparentemente, "rischi" per la concorrenza. Effettivamente, molti linguaggi funzionali tollerano anche i side effect potenzialmente prodotti dall'utilizzo di dati mutevoli e questo grava naturalmente sulla piena possibilità di parallelizzare automaticamente le applicazioni.

In ogni caso, comunque, il maggior contributo offerto dai linguaggi funzionali alla programmazione concorrente emerge nello stile di programmazione da essi proposto. La gestione della logica dell'applicazione basata sul meccanismo di definizione di funzioni ben si presta ad essere resa pienamente parallela.

Naturalmente, anche i linguaggi imperativi possono adottare, prudentemente, uno stile di programmazione simile a quello dei linguaggi funzionali e pertanto avvicinarsi maggiormente ad un approccio favorevole alla concorrenza.

1.6 Tools per lo sviluppo

A conclusione della panoramica sul rapporto tra il software e la concorrenza, è opportuno sottolineare che, a causa della difficoltà nel pensare e progettare in termini paralleli, si potrebbero ottenere risultati decisamente migliori e molto più efficienti se si potesse disporre di strumenti a supporto adeguati.

Infatti, sarebbero necessari strumenti in grado di automatizzare, perlomeno in parte, la realizzazione di applicazioni concorrenti; essendo ad esempio capaci di parallelizzare automaticamente un'applicazione descritta in termini non pienamente concorrenti. Inoltre, strumenti che permettessero di identificare efficacemente quei bug che solitamente, soprattutto in ambito concorrente, emergono solo a livello di runtime, renderebbero sicuramente più agevole la programmazione. Infine, strumenti efficaci per il testing di applicazioni concorrenti garantirebbero minor impiego di tempo alla ricerca di errori e maggiore affidabilità ai prodotti sviluppati.

Pertanto, se la concorrenza è destinata a segnare, come si è detto, una nuova era nello sviluppo del software, è fondamentale non solo definire nuovi linguaggi o nuovi approcci per la progettazione e lo sviluppo di applicazioni concorrenti, ma è necessario anche studiare e realizzare strumenti in grado di agevolare uno stile di programmazione che autonomamente può portare a difficoltà per via di una logica non proprio familiare.

Senza tali strumenti, infatti, la concorrenza potrebbe essere identificata come un impedimento che porta alla realizzazione di software di minore qualità. Invece, è fondamentale che l'approccio concorrente diventi implicito per qualunque progetto di applicazione software futura, e questo può accadere solo con un ingente lavoro per la definizione di strumenti di supporto adeguati.

Capitolo 2

Programmazione Multithread

Attualmente, la forma di programmazione più utilizzata per la gestione della concorrenza nelle applicazioni software risulta sicuramente essere quella basata sul multithreading.

Teorizzata da Edsger Dijkstra, la programmazione multithread detiene il primato nella gestione della logica legata al parallelismo e si basa sull'idea di thread. Generalmente, infatti, un'applicazione è realizzata mediante un unico processo suddiviso in più thread autonomi, i quali rappresentano attività di tipo diverso in esecuzione in modo parallelo e concorrente tra loro.

Sebbene un tale stile di programmazione possa portare a svantaggi d'uso e criticità da risolvere, le moderne applicazioni fanno ancora un massiccio uso del meccanismo di multithreading. Praticamente, l'intera totalità dei sistemi operativi esistenti supportano il multithreading ed in quasi tutti i linguaggi di programmazione sono presenti meccanismi in grado di realizzare applicazioni concorrenti mediante l'uso di thread.

In definitiva, perciò, la programmmazione multithread è sicuramente, ad oggi, un'ottima scelta per realizzare applicazioni concorrenti. Tuttavia, essa non è l'unica scelta disponibile: esistono infatti diverse altre alternative, alcune delle quali, tra l'altro, maggiormente performanti.

2.1 Il concetto di Thread

Come è stato annunciato, la programmazione multithread si basa sulla definizione di thread. Un thread rappresenta un flusso d'esecuzione indipendente che può essere eseguito parallelamente e concorrentemente agli altri thread del sistema. Più thread possono condividere dati e risorse, sfruttando il cosiddetto spazio delle informazioni condivise.

La specifica implementazione dei thread e dei processi dipende dal sistema operativo sul quale si intende eseguire l'applicazione ma, in generale, si può comunque affermare che un thread è contenuto all'interno di un processo e che diversi thread contenuti nello stesso processo condividono alcune risorse. Al contrario, processi differenti non condividono tra loro le proprie risorse.

Ogni thread risulta essere composto principalmente da tre elementi: program counter, registri e stack. Le risorse condivise con gli altri thread di uno stesso processo sono essenzialmente la sezione di codice, la sezione di dati e le risorse del sistema operativo.

Analogamente a quanto accade per i processi, anche i thread hanno un proprio stato d'esecuzione e possono sincronizzarsi tra loro. Gli stati d'esecuzione di un thread sono denominati generalmente *ready*, *running* e *blocked*.

L'applicazione tipica dei thread è certamente la parallelizzazione di un'applicazione software, anche e soprattutto per sfruttare i moderni processori multi-core; infatti, ogni core può eseguire un singolo thread.

Il vantaggio dell'uso dei thread rispetto all'uso dei processi risiede nelle prestazioni, in quanto il context switch tra processi risulta essere molto più pesante rispetto al context witch tra thread appartenenti ad uno stesso processo.

2.2 Lo spazio delle informazioni condivise

La programmazione multithread predilige come metodo di comunicazione tra thread l'uso dello spazio delle informazioni condivise. Nel caso più generale del parallelismo non strutturato, tale scelta impone che il problema maggiore da affrontare programmando con i thread sia quello della gestione di tale spazio.

Quando due o più operazioni appartenenti a thread concorrenti tentano di accedere alla memoria condivisa e almeno una di esse ha facoltà a modificare lo stato dei dati, senza un appropriato meccanismo di coordinazione si ha una cosiddetta **corsa critica**; ovvero è possibile che vengano letti o scritti dati non consistenti o non validi.

2.2.1 Gestione delle corse critiche: i lock

Il modo più semplice per aggirare le corse critiche è l'uso dei lock. Il funzionamento dei lock è semplice: quando un thread vuole accedere ad una porzione di memoria condivisa deve necessariamente acquisire il lock

su tale porzione prima di poterla utilizzare. Inoltre, dopo aver concluso la sua operazione deve rilasciare il lock ottenuto in precedenza affinchè quella porzione di memoria condivisa torni disponibile per eventuali altri thread che volessero utilizzarla.

In tal modo, quindi, risulta evidente l'impossibilità di incorrere in corse critiche poichè la necessità del lock per il thread impone che in un preciso istante solo un determinato thread possa utilizzare tale parte di memoria condivisa evitando quindi che vengano letti/scritti dati e informazioni inconsistenti.

Pur nella loro semplicità, l'uso dei lock funziona, in linea teorica, efficacemente. Tuttavia, nella pratica si nota come tale approccio possa spesso portare l'esecuzione nella spiacevole situazione di deadlock.

2.2.2 Deadlock

Banalmente si verifica un deadlock quando, a causa dell'acquisizione di alcuni lock da diversi thread, sia impossibile procedere con l'esecuzione delle operazioni in quanto i vari lock tra loro bloccano l'accesso alle risorse.

Per maggior chiarezza, si pensi alla situazione in cui coesistono due thread concorrenti (T1 e T2) che hanno a loro disposizione le risorse R1 e R2. Si supponga che il thread T1 richieda la risorsa R1 e il thread T2 la risorsa R2.

In tal caso, entrambi i thread richiedono il proprio lock e fino a questo punto tutto procede al meglio. Si immagini però che successivamente, prima di rilasciare i rispettivi lock, il thread T1 richieda il lock sulla risorsa R2 e il thread T2 richieda il lock sulla risorsa R1 essendo entrambe ora necessarie ai due processi.

Poiché entrambe le risorse sono "bloccate" i due thread si bloccano attendendo che si liberi la risorsa di cui hanno bisogno, ma è evidente che tale situazione di blocco non si risolverà mai.

La situazione appena descritta costituisce l'esempio più emblematico del verificarsi di una situazione di deadlock.

Per quanto detto, perciò, si evince come l'uso dei lock per garantire la sincronizzazione nell'accesso alla memoria condivisa sia, da un lato, un metodo funzionante ma, dall'altro, anche potenzialmente distruttivo in determinati casi.

La teoria insegna che è possibile prevenire gli stati di deadlock analizzando preventivamente, ad esempio mediante il grafo delle attese, tutte le potenziali corse critiche ed evitare così, attraverso un'opportuna progettazione, gli stati di deadlock.

Nonostante il loro teorico buon funzionamento, tuttavia i lock non sono solo soggetti alle situazioni dannose di dedlock ma presentano anche molti altri aspetti negativi per l'applicazione nel suo complesso.

Per prima cosa, si tratta di un approccio conservativo che, per sua natura, introduce spesso overhead superfluo; è poi limitata la scalabilità dell'applicazione ed aumentata la sua complessità.

Inoltre, l'uso dei lock è decisamente in conflitto con l'eventuale necessità di imporre delle priorità d'accesso alla memoria condivisa da parte dei vari processi.

Dal punto di vista pratico, infine, un'applicazione contenente dei lock presenta notevoli difficoltà in fase di ricerca di eventuali errori (debug).

In conclusione, quindi, sarebbe opportuno utilizzare metodi alternativi per garantire sincronismo d'accesso alla memoria condivisa ed evitare corse critiche.

2.3 Sincronizzazione

Tra le alternative per la sincronizzazione di due o più thread e la gestione dell'accesso allo spazio delle informazioni condivise sono degni di nota gli approcci basati sull'utilizzo di semafori e quelli basati sul concetto di monitor.

2.3.1 Approccio mediante Semafori

Inventato da E. Dijkstra ed utilizzato per la prima volta nel sistema operativo *THE*, un semaforo è un ADT (tipo di dato astratto) gestito dal sistema operativo per sincronizzare l'accesso di più thread a dati e risorse condivise.

Essenzialmente, un semaforo è costituito da una variabile interna che identifica il numero di accessi contemporanei ad una risorsa alla quale esso è associato.

Il funzionamento di un semaforo si basa sulle due funzioni di *wait* e di *signal*, come illustrato di seguito:

- Ogni volta che un thread vuole accedere ad un dato o ad una risorsa alla quale è associato un semaforo, esso deve invocare l'operazione di wait, la quale decrementa la variabile interna del semaforo e consente l'accesso alla risorsa se il valore di tale variabile risulta essere non negativo. Qualora il valore risultasse negativo il thread verrebbe sospeso e posto in attesa della liberazione della risorsa da parte di un altro thread.

- Ogni volta che un thread ha terminato il proprio utilizzo del dato o della risorsa condivisa, deve liberare la risorsa mediante l'operazione di signal. In questo modo la variabile interna del semaforo viene incrementata e qualora fossero presenti nella coda del semaforo alcuni thread in attesa, viene concessa l'opportunità di accedere al dato o alla risorsa al primo di questi.

Sebbene ad una prima analisi il meccanismo dei semafori non presenti evidenti problematiche, esso funziona correttamente solo se le operazioni di wait e di signal vengono eseguite in blocchi atomici. Se così non fosse, ovvero se una delle due operazioni viene interrotta, potrebbero generarsi spiacevoli situazioni.

Si supponga che due thread eseguano contemporaneamente l'operazione wait su un semaforo la cui variabile interna ha valore 1. Si supponga anche che dopo che il primo thread ha decrementato il semaforo da 1 a 0, il controllo passi al secondo thread, il quale decrementa il semaforo da 0 a -1 e si pone in attesa visto il valore negativo della variabile interna. A questo punto, con il controllo che torna al primo thread, il semaforo possiede valore negativo e quindi anche il primo thread si pone in attesa. Pertanto, nonostante il semaforo fosse tale per cui l'accesso sarebbe stato possibile perlomeno per un thread, il fatto che l'operazione di wait non sia stata eseguita in termini atomici, ha portato ad una soluzione di stallo.

Un particolare uso dei semafori è rappresentato dal cosiddetto *mutex*. Un mutex infatti, altro non è che un semaforo con una variabile interna inizializzata al valore 1; il quale consente quindi la realizzazione della mutua esclusione nell'accesso a dati e risorse.

Inoltre, inizializzando un semaforo al valore zero, si ottiene un cosiddetto *semaforo evento* con l'unico scopo di sincronizzare la computazione di due o più thread senza che nessuno dei thread debba far necessariamente uso di dati o risorse comuni contemporaneamente.

2.3.2 Monitor, alternativa ai semafori

I semafori sono tutt'oggi normalmente utilizzati nei linguaggi di programmazione multithread, tuttavia essi presentano due importanti problematiche:

- Non impedendo la possibilità per un thread di effettuare più operazioni di wait su uno stesso semaforo, risulta molto facile dimenticare poi di fare tutte le signal necessarie in relazione al numero di wait eseguite.

- È possibile incorrere in situazioni di deadlock. Ad esempio, si ha una situazione di deadlock se il thread T1 esegue una wait sul semaforo S1, mentre il thread T2 esegue una wait sul semaforo S2 e poi T1 esegue una wait su S2 e T2 esegue una wait su S1.

Pertanto, i maggiori teorici della programmazione concorrente, compreso lo stesso Dijkstra, hanno dichiarato obsoleti i semafori come tecnica di programmazione.

Come alternativa ai semafori, è possibile utilizzare il concetto di monitor. Un monitor può esser definito come tipo di dato astratto, in cui le operazioni fornite per la manipolazione del dato (chiamate entry) possono essere eseguite da un solo thread alla volta.

L'implementazione interna del monitor è nascosta ai thread, i quali non possono accedervi direttamente, ma unicamente attraverso le entry (operazioni) predisposte dal monitor.

Autonomamente, un monitor gestisce il meccanismo di mutua esclusione tra thread che vogliano utilizzare le risorse messe a disposizione dal monitor. Per gestire anche la sincronizzazione è necessario utilizzare le cosiddette variabili di condizione, le quali permettono la gestione esplicita della sincronizzazione.

2.4 Criticità della programmazione multithread

Introducendo il concetto di lock per la gestione dello spazio per le informazioni condivise si è illustrato come un tale meccanismo, teoricamente semplice e privo di effetti collaterali, possa portare a situazioni di blocco indesiderate.

Tra le alternative proposte, i monitor sembrano essere quella con minori effetti indesiderati. Essi infatti possono essere facilmente implementati nei moderni linguaggi di programmazione che supportano il multithreading, in quanto risulta sufficiente dichiarare un'insieme di metodi di una classe con uno specifico marcatore (i metodi *syncronized* in Java sono l'esempio più ovvio) vincolando l'esecuzione di tali metodi a non essere parallela; in questo modo quindi risulta facile garantire la sincronizzazione.

Tuttavia, però, anche un tale approccio può risultare spesso inefficiente. Senza entrare nel dettaglio, si può affermare che in generale, qualunque meccanismo si usi per la gestione dello spazio delle informazioni condivise e per la sincronizzazione nella programmazione multithread,

esso può comportare criticità per via della grande attenzione richiesta al programmatore per utilizzare tali metodi.

Oltre alla difficoltà intrinseca della programmazione multithread (soprattutto nella localizzazione di eventuali errori o bug), altri sono i limiti e le criticità da affrontare.

Innanzi tutto, si pone un problema di performance quando i compiti da far eseguire ad un thread sono troppo brevi da non riuscire ad rendere pressoché nullo il costo dovuto alla creazione e all'inizializzazione del thread, nonché alla sua successiva dismissione. Inoltre, spesso si ha la necessità di eseguire operazioni che non possono essere rese parallele e, in questo caso, l'overhead dovuto alla gestione dei thread coinvolti risulta decisamente elevato.

Infine, emerge l'aspetto del context switch tra thread che, se nel sistema sono presenti molti thread, richiede molto tempo e diversi cicli di esecuzione.

Pertanto, è idea comune oggigiorno che continuare a sovraccaricare i paradigmi di programmazione esistenti con meccanismi per ottenere un migliore parallelismo, limitando gli sprechi di tempo, di risorse e di meccanismi a corredo (per coordinazione e sincronizzazione, ad esempio) basandosi esclusivamente sulla logica della programmazione multithread non sia propriamente la soluzione ottimale.

Risulta quindi opportuno orientarsi verso altri modelli di programmazione con un approccio alla concorrenza profondamente diverso al fine di ottenere un migliore e più efficiente stile di programmazione.

Capitolo 3

Il modello ad Attori

Il modello ad attori è un paradigma di programmazione per la computazione concorrente, definito mediante un modello matematico e basato sul concetto di attore software, che ne rappresenta la primitiva.

Nato nel 1973, tale modello è stato utilizzato sia per la comprensione teorica della logica relativa alla programmazione concorrente, sia come base per diverse implementazioni di sistemi concorrenti.

Diversamente da altri modelli di programmazione, il modello ad attori trae ispirazione dalla fisica moderna (in particolare dalla teoria della relatività generale e dalla meccanica quantistica) [5] ed è stato sviluppato con la prospettiva che in futuro potessero essere realizzati calcolatori con caratteristiche tali da aumentare incredibilmente le performance della computazione parallela; previsione che si è avverata con la nascita dei multiprocesori prima e dei microprocessori multi-core poi.

Da un punto di vista storico e cronologico, le prime pubblicazioni sul modello ad attori sono quelle di C. Hewitt, P. Bishop e R. Steiger del 1973 completate poi da I. Greif nella sua tesi di dottorato. Successivamente, H. Baker e lo stesso Hewitt pubblicarono nel 1975 una serie di leggi ed assiomi per il paradigma. La definizione del modello è poi culminata con la pubblicazione di Gul Agha [1] del 1986, in cui egli ha rimodellato il paradigma con semantiche operazionali completando così la teoria del modello ad attori.

3.1 Definizione di Attore

Similmente a quanto accade nel modello orientato agli oggetti, la filosofia del modello ad attori è basata sull'idea che *"everything is an actor"*. La differenza filosofica risiede tuttavia nel fatto che, mentre il modello

orientato agli oggetti prevede un'esecuzione tipicamente sequenziale, il modello ad attori è, invece, intrinsecamente concorrente.

Un attore è un'entità autonoma che opera concorrentemente agli altri attori del sistema in modo del tutto asincrono. Ogni attore è definito tramite un nome univoco all'interno del sistema e possiede un proprio *behavior* (comportamento, in termini computazionali).

Ogni attore, quando è inattivo, si trova nello stato di idle; stato in cui si attende la ricezione di messaggi. Quando un messaggio è pendente nella *mailbox*, l'attore accetta il messaggio ed esegue una certa computazione in base a quanto specificato nel proprio behavior. Il risultato della computazione è "comunicato" al sistema mediante l'esecuzione di una tra le seguenti tre operazioni:

- Invio di un messaggio ad un altro attore;
- Creazione di un nuovo attore;
- Aggiornamento dello stato interno dell'attore stesso.

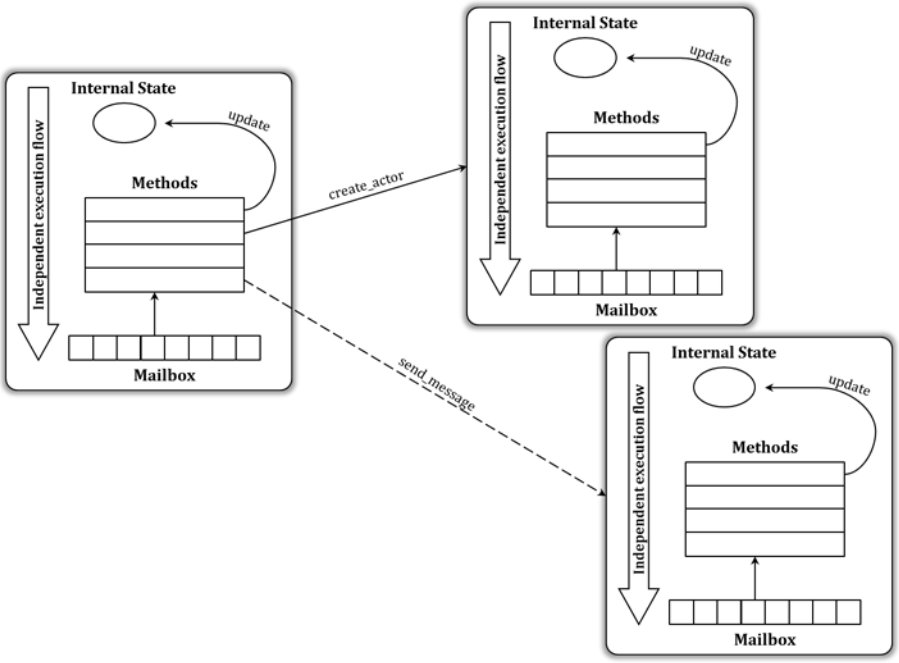

Figura 3.1: *Rappresentazione semplificata del concetto di attore*

Lo schema di Figura 3.1 riassume quanto descritto circa la definizione di un attore. Si possono notare gli elementi principali che costituiscono un

attore, tra i quali si osservi in particolare il fatto che ogni attore possiede un proprio flusso d'esecuzione indipendente dagli altri. Inoltre, sono evidenziate le operazioni che ogni attore può effettuare come risultato dell'esecuzione di un'operazione.

A conclusione della definizione del concetto di attore, è importante chiarire che ogni attore non condivide con gli altri attori del sistema il proprio stato interno. Quindi, per effettuare una modifica o ottenere informazioni relativamente a tale stato è necessario inviare un esplicito messaggio (asincrono) all'attore interessato.

Inoltre, va specificato che l'ordine di arrivo dei messaggi ad un attore, nella relativa mailbox, è del tutto indeterminato per via dell'asincronismo con cui tutti i messaggi sono inviati.

3.2 Proprietà semantiche

Le proprietà semantiche chiave del modello ad attori sono essenzialmente quattro:

- Incapsulamento dello stato interno;
- Atomicità d'esecuzione di ogni metodo come risposta ad un ben preciso messaggio;
- Fairness nella gestione dell'avanzamento della computazione degli attori del sistema e nella consegna dei messaggi;
- Location Transparency.

Nonostante tali proprietà siano condizione necessaria per garantire l'efficienza del modello, non tutte le implementazioni esistenti sono state realizzate imponendo il rispetto di tutte e quattro le proprietà. Questo indubbiamente rende più snelle le implementazioni del modello ad attori, ma aumenta il carico del programmatore il quale deve fare attenzione affinché anche le proprietà non espressamente considerate dall'implementazione in uso siano comunque garantite e rispettate.

3.2.1 Incapsulamento

La proprietà di incapsulamento (*Encapsulation*) racchiude in se quanto detto nella definizione del concetto di attore circa lo stato interno. Tale proprietà infatti impone che due o più attori non condividano tra loro il proprio stato interno.

Nel modello ad attori, se si vuole modificare o leggere lo stato interno di un attore, è necessario inviare un messaggio all'attore interessato ed attendere la risposta.

Poiché il modello ad attori, come detto, è un modello intrinsecamente concorrente, è teoricamente possibile che un messaggio raggiunga un attore mentre questo risulta impegnato nel processare un altro messaggio giunto in precedenza nella sua mailbox. Se ciò accadesse e se, per ipotesi, al secondo messaggio fosse consentito, secondo le politiche di priorità sui messaggi, interrompere la computazione dell'attore a cui è destinato e modificarne lo stato interno mentre esso sta ancora processando il precedente messaggio, allora si otterrebbe come conseguenza che nel modello ad attori non sarebbe più possibile assumere che il comportamento di un attore a fronte di un messaggio ricevuto possa essere predeterminato in funzione del valore che aveva lo stato interno al momento della ricezione del messaggio stesso.

3.2.2 Atomicità

La proprietà di atomicità (*Atomicity*) evita la potenziale inconsistenza dello stato interno che si otterrebbe nell'ipotesi del paragrafo precedente circa l'eventuale assegnazione di livelli di priorità ai messaggi.

Nel modello ad attori, infatti, si impone che ciascun attore processi ogni messaggio in un'unica unità atomica formata da tutte le azioni necessarie a dare una risposta al messaggio. In tal modo quindi, si ottiene, nel caso di più messaggi in ricezione, una coda d'esecuzione determinata dal momento di arrivo che permette, quindi, di predeterminare il comportamento dell'attore a fronte dell'ordine d'arrivo dei messaggi.

3.2.3 Fairness

La proprietà di Fairness (che, in modo forzato, potrebbe essere tradotta in lingua italiana come "equità e garanzia d'esecuzione") impone che ad ogni attore sia garantita la possibilità di computare se esso ha almeno un messaggio a cui rispondere.

Inoltre, la stessa proprietà impone anche che ogni messaggio inviato debba essere sempre consegnato all'attore destinatario, con l'unica eccezione per quei messaggi destinati ad attori permanentemente o temporaneamente disabilitati all'interno del sistema (per esempio perché impegnati in loop infiniti o che stanno tentando di eseguire un'operazione illecita oppure che risultano essere interdetti alla ricezione di messaggi).

3.2.4 Location Transparency

L'ultima proprietà semantica del modello ad attori è la cosiddetta Location Transparency, ovvero trasparenza rispetto alla dislocazione degli attori all'interno del sistema. Tale proprietà assicura l'importante caratteristica al nome di un attore di non dipendere dalla posizione fisica dell'attore stesso all'interno del sistema. In questo modo quindi, un attore può scambiare messaggi con un altro attore che si trovi sullo stesso core, sulla stessa CPU o su un altro nodo della rete senza preoccuparsi dei diversi meccanismi necessari alla comunicazione.

La proprietà di Location Transparency garantisce perciò indipendenza tra la locazione fisica e la locazione logica di ogni attore nel sistema, favorendo in tal senso un'agile realizzazione di applicazioni distribuite. Inoltre, mediante il rispetto di tale proprietà si conquista l'eventuale possibilità di effettuare una riconfigurazione del sistema in modo del tutto trasparente alla logica di funzionamento.

3.3 Sincronizzazione

Quando si ha a che fare con la programmazione concorrente, una problematica dalla quale non si può prescindere è sicuramente quella della sincronizzazione del lavoro delle varie entità autonome del sistema.

Nel caso del modello ad attori, la sincronizzazione tra gli attori (che costituiscono le entità autonome del sistema) è raggiunta attraverso la comunicazione. Due attori, infatti, possono sincronizzare il proprio lavoro semplicemente scambiandosi messaggi secondo ben precisi pattern.

Sebbene esistano innumerevoli pattern di comunicazione che potrebbero essere utilizzati per gestire la sincronizzazione, le varie implementazioni del modello ad attori hanno preferito adottare principalmente i seguenti pattern:

- RPC-like Messaging
- Local Synchronization Constraint

In ogni caso, comunque, i costrutti dei linguaggi consentono di implementare qualunque altro pattern si preferisca utilizzare per gestire la sincronizzazione.

3.3.1 RPC-like Messaging

La comunicazione tramite il pattern RPC-like prevede che il mittente di un messaggio attenda la ricezione della risposta al messaggio stesso prima di procedere a processare gli altri eventuali messaggi.

Per meglio comprendere il funzionamento del pattern si immagini che, in un sistema, l'attore A voglia sincronizzarsi con l'attore B. Per fare ciò, quindi, l'attore A invia all'attore B un messaggio e si pone poi in attesa della risposta relativa a quel messaggio, accantonando momentaneamente tutti gli altri eventuali messaggi che dovessero raggiungerlo prima dell'arrivo della risposta dall'attore B. Così facendo perciò, l'attore A procede nella propria computazione solo dopo essersi sincronizzato con l'attore B. Si noti che il pattern non vieta all'attore B di riceve messaggi da altri attori dopo aver ricevuto il messaggio dall'attore A e non vieta nemmeno di inviare le relative risposte prima di rispondere all'attore A.

La Figura 3.2 schematizza l'esempio proposto evidenziando come nonostante l'attore A riceva il msg_1 in un determinato momento, esso risponda a tale messaggio solo dopo aver ricevuto la risposta al messaggio di sincronismo. Viceversa, per l'attore B che non ha necessità di sincronizzarsi, può accadere che esso risponda ad un altro messaggio (msg_2 ad esempio) prima di inviare la risposta al messaggio di sincronismo dell'attore A.

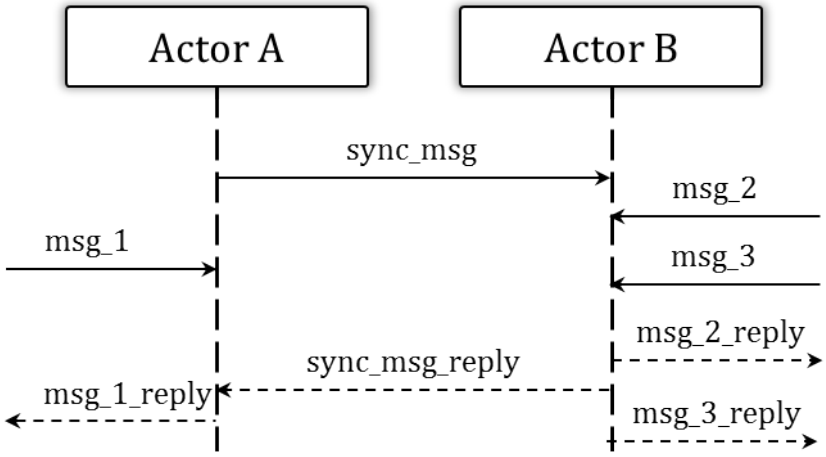

Figura 3.2: *Sincronizzazione mediante il pattern RPC-like.*

Il pattern RPC-like è supportato in quasi tutte le implementazioni del modello ad attori e risulta particolarmente comodo in tre tipici scenari di comunicazione.

Il primo scenario si ha quando un attore desidera inviare una sequenza ordinata di messaggi ad un altro attore, con la certezza cioè che il destinatario riceva i messaggi nell'ordine con il quale sono stati inviati.

Il secondo riguarda la necessità di un attore ad avere la sicurezza che un altro attore abbia ricevuto ed elaborato un messaggio prima di comunicare con un terzo attore.

Il terzo ed ultimo scenario è relativo invece al caso in cui lo stato interno di un attore dipenda dalla risposta ad un messaggio che questo invia ad un altro attore.

Gestire il sincronismo attraverso il pattern RPC-like è decisamente molto simile alle chiamate a procedura dei linguaggi orientati agli oggetti e per questo motivo, molto spesso, gli sviluppatori tendono a farne un uso sovrabbondante. Purtroppo, però, una tale pratica porta ad introdurre dipendenze nel codice e si oppone alla logica concorrente del modello ad attori; inoltre, può introdurre la problematica dei deadlock già vista nel Capitolo 2 che il modello ad attori, invece, non possiede per la natura asincrona del sistema di scambio di messaggi. Pertanto, è fondamentale non abusare del pattern RPC-like.

3.3.2 Local Synchronization Constraint

La tecnica Local Synchronization Constraint (letteralmente, vincoli locali per la sincronizzazione) permette di evitare le cosiddette *busy waiting*, o attese attive.

Per meglio comprendere il concetto di attese attive, si consideri il seguente esempio. Si supponga di avere un sistema nel quale ci sia un attore che possa distribuire compiti da eseguire e una serie di attori esecutori di tali compiti. Ogni attore esecutore, una volta terminato il proprio compito, invia all'attore "distributore" un messaggio per chiedere un nuovo compito aspettandosi di ricevere il compito da eseguire come risposta al messaggio. Può accadere tuttavia che al momento della richiesta non vi siano nuovi compiti da eseguire e perciò che l'attore "distributore" cestini il messaggio ricevuto dall'attore "esecutore". In questo modo, tuttavia, l'attore alla ricerca di un compito continuerà ad inviare messaggi di richiesta trovandosi quindi in una attesa attiva, ovvero un'attesa nella quale viene comunque eseguita della computazione ridondante.

Per ovviare a questo problema, è possibile realizzare vincoli interni all'attore che riceve i messaggi di richiesta compiti, affinché esso bufferizzi il messaggio e ne ritardi l'esecuzione, senza cestinare il messaggio e conseguentemente porre l'attore "esecutore" in un vero stato di attesa.

3.4 Esecuzione di un programma ad attori

Un'applicazione software basata sul modello ad attori è costituita, come è ovvio, da attori software; ognuno dei quali possiede un nome univoco noto, inizialmente, solo al creatore dell'attore (che, salvo eccezioni, risulta essere un altro attore).

Si consideri un programma costituito da una una serie di attori che sono in grado di comunicare tra loro unicamente mediante l'invio di messaggi, attraverso la generica primitiva `send()`

```
send(Actor to, Message msg);
```

che ha come effetto quello di aggiungere alla mailbox dell'attore identificato dal parametro `to` il messaggio specificato nel parametro `msg` e di ritornare immediatamente al chiamante, senza attendere alcuna risposta, concludendo la propria esecuzione.

Vista la modalità di funzionamento di tale primitiva di comunicazione e considerando i ritardi introdotti di protocolli di comunicazione e dalla rete in generale, è ora chiaro il non-determinismo dell'arrivo dei messaggi ai vari attori dell'applicazione.

All'avvio del programma, le mailbox degli attori del sistema risultano essere vuote e, affinché sia possibile avviare l'esecuzione, l'ambiente (nel quale viene eseguito il programma) invia uno specifico messaggio ad un attore del sistema che in tal modo funge da *entry point*.

Ogni attore può essere considerato alla stregua di un loop infinito che esegue ciclicamente le seguenti operazioni:

- Prelevare il primo messaggio presente nella propria mailbox;
- Interpretare la richiesta presente nel messaggio;
- Eseguire il metodo corrispondente alla richiesta ricevuta.

Qualora la mailbox risultasse vuota, l'attore si pone in uno stato detto di *idle* nel quale resta in attesa della ricezione di nuovi messaggi.

Durante l'esecuzione della richiesta presente nel messaggio, è possibile che venga aggiornato lo stato interno dell'attore mediante la primitiva `update()`

```
update(StatusElement name, StatusValue value);
```

oppure che sia creato un nuovo attore mediante la primitiva `create()`

```
create(Actor name);
```

oppure, infine, che sia inviato un nuovo messaggio ad un altro attore sempre mediante la primitiva `send()`.

Concludendo, il programma termina quando tutti gli attori presenti sono contemporaneamente

1. nello stato di idle, ovvero nessun attore ha messaggi pendenti nella propria mailbox;
2. chiusi rispetto alla ricezione di messaggi, ovvero per ciascun attore deve essere stata interdetta la possibilità di ricevere messaggi.

Si noti che, per la breve descrizione della modalità di esecuzione di un generico programma ad attori effettuata nel presente paragrafo, si è volontariamente preferito evitare qualunque riferimento a specifiche implementazioni del modello, rimandando ad una successiva e più dettagliata descrizione quando, nei prossimi capitoli, saranno analizzate alcune specifiche implementazioni.

3.5 Vantaggi del modello ad attori

Il modello orientato agli oggetti ha introdotto l'idea della *separation of concerns* che impone di distinguere l'interfaccia degli oggetti dalla loro rappresentazione, ovvero di separare il "cosa fa" dal "come lo fa". La separation of concerns ha garantito al modello orientato agli oggetti grande popolarità e ha reso le applicazioni basate su tale modello agevoli da progettare e realizzare.

Perseguendo gli stessi obiettivi di popolarità e semplicità, anche il modello ad attori adotta l'idea della separation of concerns e la estende al caso concorrente. In particolare, il modello ad attori tende a separare il controllo ("dove" e "come") dalla logica della computazione.

Ciò, fa si che i programmi basati sul modello ad attori possano essere decomposti in componenti autonomi ed interoperanti tra loro in modo asincrono. Tale caratteristica agevola molto lo sviluppo, tra gli altri, dei sistemi distribuiti, dei sistemi mobile e dei sistemi realtime.

3.5.1 Coordinazione

Oltre ai vantaggi già descritti, uno dei più importanti del modello ad attori è, sicuramente, la sua caratteristica di semplificare la gestione della

coordinazione per le applicazioni software, aumentando la granularità dei suoi componenti.

In questo senso, la coordinazione è agevolmente ottenibile pensando a come i vari messaggi tra gli attori vengono intervallati tra loro piuttosto che in termini di accesso alterno a variabili condivise, che costituisce invece una delle parti centrali della logica di coordinazione del modello orientato agli oggetti.

Naturalmente, a causa dell'alto numero delle possibili combinazioni di arrivo dei messaggi ad un qualunque attore, la coordinazione, per quanto agevole, potrebbe diventare fonte di possibili errori se non ben ottimizzata tramite specifici tool di supporto alla programmazione.

3.6 Problematiche e osservazioni

Con particolare riferimento alle applicazioni distribuite e non solo, i vantaggi introdotti nella progettazione e nello sviluppo del software mediante un approccio ad attori sono abbastanza evidenti dalla descrizione del modello.

Tuttavia, anche il modello ad attori risente di alcune problematiche; le più importanti delle quali sono riassunte nel seguente elenco.

1. È completamente assente il concetto di gerarchia tra attori; con la conseguente impossibilità di aggregare tra loro attori con comportamenti simili. Inoltre risulta impossibile specializzare attori "ereditando" funzionalità presenti in altri attori.

2. Il meccanismo di comunicazione tra attori tramite lo scambio di messaggi in modo asincrono mal si presta all'implementazione di particolari algoritmi o alla realizzazione di specifiche strutture dati (per esempio, strutture dati a *Stack*).

3. È mancante la possibilità di definire dati mediante tipi primitivi. Se nel modello ad oggetti all'affermazione "everything is an object" sono state consentite deroghe introducendo anche tipi primitivi non costituiti da oggetti, nel modello ad attori l'idea che "everything is an actor" appare pienamente rispettata con l'inevitabile implicazione di rendere lievemente più "pesanti" le applicazioni implementate seguendo il modello ad attori.

In aggiunta a quanto detto, alcuni studiosi hanno osservato e discusso altre possibili implicazioni negative all'uso del modello ad attori.

Degno di nota è l'articolo [7] di Paul Mackay nel quale egli sostiene che la possibilità per un attore di poter creare nuovi attori, possa essere potenzialmente dannosa all'intero sistema qualora non sia minuziosamente guidata. Creare un altro attore aggiungendolo al sistema significa, infatti, modificare lo stato dell'intero sistema e ciò può notevolmente influire sulle sue performance. Decidere dove collocare un nuovo attore, creato in fase di esecuzione, risulta essere un azione particolarmente delicata; sopratutto se la creazione avviene in un'applicazione (molto) distribuita o potenzialmente soggetta a alte variazioni delle performance in funzione della propria dislocazione sulla rete.

In definitiva, comunque, in base a quanto esposto risulta particolarmente evidente come le maggiori problematiche relative al modello ad attori siano identificabili in quanto tali, quasi esclusivamente se si parte dall'idea di applicazioni basate sul modello orientato agli oggetti e si tenta di riproporre le stesse applicazioni secondo il modello ad attori, senza voler apportare modifiche alla loro progettazione.

Risulta pertanto chiaro che realizzare applicazioni basate sul modello ad attori richiede principalmente un diverso approccio in fase di analisi e progettazioni che deve differire, a volte anche di molto, dall'approccio che si adotterebbe analizzando e progettando un'applicazione che sarà basata sul modello orientato agli oggetti.

3.7 Applicazioni del modello ad attori

A conclusione della descrizione del modello ad attori, si vogliono esaminare ora alcuni esempi di applicazione di tale modello. In particolare, saranno analizzate tre differenti applicazioni, ad oggi esistenti ed utilizzate, scelte per evidenziare essenzialmente l'eterogeneità dell'ambito di applicazione del modello ad attori.

- **Twitter**, un servizio web di microblogging e social netowork molto diffuso ed utilizzato ad oggi; i cui principali componenti risultano essere stati realizzati sfruttando una delle tante implementazioni del modello ad attori;

- **Dart**, un linguaggio sviluppato da Google per la realizzazione di applicazioni web strutturate;

- **Orleans**, una piattaforma software (framework) sviluppato dall'*eXtreme Computing Group* per agevolare la realizzazione di applicazioni client-side orientate all'uso dei servizi di clud computing.

Le applicazioni scelte saranno descritte molto brevemente, focalizzando principalmente l'attenzione sul perché sia stato adottato il modello ad attori (per il particolare aspetto per il quale è stato scelto) e sul come sia stato utilizzato.

3.7.1 Twitter: microblogging e social network

Creato nel marzo 2006 dalla Obvious Corporation di San Francisco, Twitter è un servizio gratuito di rete sociale e microblogging che fornisce agli utenti una pagina personale aggiornabile tramite messaggi di testo. Con oltre 500 milioni di utenti attivi, Twitter ha ricosso e riscuote tuttora ampio successo, soprattutto grazie all'uso che personaggi di spicco dello spettacolo, della politica e non solo fanno di questo strumento.

Funzionamento

Il funzionamento di Twitter risulta semplice ed intuitivo. Ogni utente, quando lo desidera, può pubblicare un testo di massimo 140 caratteri che rappresenta il tweet che egli vuole trasmettere in quel momento. Attraverso il meccanismo dei followers, ovvero degli utenti che si dichiarano interessati a seguire i tweet pubblicati su una specifica pagina, i vari tweet vengono notificati a chi risulta interessato.

Caratteristiche Tecniche

L'interfaccia web di Twitter è attualmente scritta in Ruby on Rails (un framework per progetti web basato sul linguaggio Ruby); viceversa, il nucleo dell'applicazione, ovvero la gestione dei tweet e dei followers è realizzato in linguaggio Scala, una delle tante implementazioni del modello ad attori.

Inizialmente, anche il nucleo era scritto in Ruby, tuttavia la crescente popolarità di Twitter e la necessità di ridurre i costi in termini di tempo per la gestione asincrona e parallela dell'inoltro dei tweet ai followers, ha convinto gli ingegneri di Twitter ad orientarsi verso un linguaggio che fosse basato su un modello di programmazione orientato intrinsecamente e naturalmente alla programmazione concorrente [16].

Secondo Steve Jenson (ingegnere di Twitter), Ruby e Scala sono estremamente compatibili e interfacciabili tra loro e anche per questo motivo la scelta è ricaduta su Scala.

Con un linguaggio ad attori come Scala, sostengono gli sviluppatori di Twitter, il codice risulta facile da scrivere e veloce da manutenere e modificare [17].

Infine, secondo Alex Payne, la scelta del modello ad attori (e di Scala) è risultata, dopo un duro lavoro per passare dalla logica di programmazione di Ruby a quella di Scala, molto performante e decisamente conciliante con l'architettura generale di Twitter, la quale ha visto un enorme opera di snellimento grazie alla semplicità data dal meccanismo di gestione della concorrenza del modello ad attori.

3.7.2 Dart: linguaggio per applicazioni web strutturate

Presentato nel mese di ottobre del 2011, Dart è un linguaggio di programmazione per il web sviluppato da Google con lo scopo di sostituire il linguaggio JavaScript.

Secondo Google infatti, non è possibile risolvere i problemi di JavaScript evolvendo ulteriormente il linguaggio; per questo motivo quindi l'introduzione di un nuovo linguaggio orientato allo sviluppo di applicazioni web strutturate potrebbe prima di tutto offrire maggiori potenzialità rispetto a quanto proposto da JavaScript ed, in particolare, maggiori supporti alla programmazione orientati alla concorrenza e alla sicurezza.

Concorrenza in Dart

Seguendo lo stile proposto da Erlang (linguaggio che implementa il modello ad attori), anche Dart introduce il modello ad attori per la gestione della concorrenza [18].

In Dart, gli attori sono chiamati *Isolate* [19], nome che richiama la loro più importante proprietà, ovvero quella di non possedere alcun stato condiviso.

Ogni nuovo `Isolate` può essere istanziato derivando dalla relativa classe base e la loro computazione può essere avviata chiamando la funzione `spawn()`.

Esistono due diversi tipi di Isolate: gli isolate detti *light* che sono eseguiti nello stesso thread in cui sono stati creati e gli isolate detti *heavy* che invece sono eseguiti in thread dedicati.

Gli isolate utilizzano un meccanismo di comunicazione basato sui concetti di ReceivePort, per accettare i messaggi, e di SendPort, per inviare messaggi. Un messaggi inviato da una SendPort vine consegnato in modo asincrono alla ReceivePort corrispondente (ovvero quella che ha creato la SendPort) senza bloccare l'isolate che lo ha inviato.

Attualmente, le API di Dart relative alla concorrenza ed in particolar modo agli Isolate sono in continuo upgrade e questo rende complicato

descrivere in modo dettagliato le opportunità offerte da Dart in termini di concorrenza.

In ogni caso, comunque, è importante sottolineare come un linguaggio, che si propone come sostituto di un più noto e molto utilizzato linguaggio come JavaScript, abbia considerato il modello ad attori come modello da implementare per gestire la concorrenza.

Questo aspetto risulta molto significativo poiché dimostra come il modello ad attori venga oggi rivalutato e come cominci ad essere implementato in linguaggi destinati ad ambienti commerciali piuttosto che ad ambienti di sola ricerca.

3.7.3 Orleans: piattaforma per il cloud computing

Sviluppato dall'eXtreme Computing Group nell'ambito commerciale Microsoft Research [20], Orleans nasce come una piattaforma con lo scopo di semplificare la progettazione di applicazioni per il cloud computing.

L'obiettivo principale di Orleans è quello di convincere gli sviluppatori di applicazioni *cloud oriented* ad adottare semplici pattern per la gestione della concorrenza, facili da comprendere e agevoli da realizzare, attraverso una piattaforma (Orleans) che, basandosi sul modello ad attori, permetta di realizzare applicazioni affidabili e scalabili.

Specifiche per la concorrenza

Il modello ad attori è implementato in Orleans mediante componenti chiamati *grain*, i quali sono unità computazionali isolate che comunicano tra loro attraverso messaggi asincroni.

All'interno di ogni grain, la gestione della concorrenza (sia per i messaggi asincroni che per i task locali) è basata su un meccanismo detto *promise*. Tale meccanismo è costituito da un semplice lifecycle. Inizialmente una promise risulta nello stato di *unresolved*, ovvero rappresenta l'attesa di un risultato in tempo futuro. Appena tale risultato viene ricevuto, lo stato muta in *fulfilled* e il risultato diventa il valore della promise.

In Orleans le promise identificano le primitive asincrone per lo scambio dei messaggi; ogni volta che un grain richiede un'operazione asincrona (invia un messaggio) esso riceve immediatamente una promise mediante la quale ottiene la certezza che in un tempo futuro riceverà un valore in risposta alla richiesta di operazione asincrona effettuata.

Come già accennato, i grain possiedono uno stato interno isolato dal resto del sistema ed inoltre sono vincolati a seguire un predeterminato

modello di esecuzione. Questo garantisce ad Orleans la consistenza dei dati gestiti dai grain fornendo così la possibilità di spostare e/o duplicare facilmente i grain all'interno del sistema. In questo modo, Orleans è in grado di identificare facilmente ed automaticamente eventuali errori e inconsistenze, con la conseguente possibilità di effettuare ripristini automatici.

Il modello di programmazione scelto da Orleans è di tipo generale e consente ai meccanismi di comunicazione asincrona di evolvere dinamicamente a runtime.

Concludendo, si può affermare che la caratteristica vincente di Orleans è quella di consentire agli sviluppatori di applicazioni per il cloud computing di concentrarsi unicamente sulla logica dell'applicazione, lasciando ad Orleans il compito di gestire tutte le altre caratteristiche richieste ad applicazioni di quel genere.

Capitolo 4

Alcune implementazioni del modello

Contrariamente a quanto si possa pensare, attualmente esistono molte implementazioni del modello ad attori. Alcune nate in ambiti di ricerca e poi utilizzate per scopi commerciale, mentre altre esistenti unicamente per scopi accademici.

Naturalmente, ogni linguaggio che ha scelto di implementare il modello ad attori ha liberamente introdotto qualche variante discostandosi quindi, a volte in modo anche consistente, dal modello originario teorizzato da C. Hewitt e perfezionato da G. Agha.

Tra le prime sperimentali implementazioni del modello, sono degne di nota le seguenti:

- **Act 1** (successivamente Act 2 e poi Act 3), sviluppata al MIT da Henry Lieberman nel 1981 con l'obiettivo di realizzare un linguaggio performante orientato principalmente all'intelligenza artificiale [15].

- **Cantor**, presentata nel 1988 al NFS Workshop per la concorrenza nell'ambito del paradigma ad oggetti e sviluppata con l'obiettivo di velocizzare e rendere più performante la computazione per scopi scientifici.

Seguirono poi molte altre implementazioni del modello ad attori; alcune delle quali, visto l'enorme successo, divennero veri e propri linguaggi di programmazione utilizzati attualmente anche per prodotti con fine commerciale. Sono degne di nota le seguenti implementazioni:

- **Erlang**, che fece la sua prima comparsa nel 1986 e ad oggi risulta essere probabilmente l'implementazione del modello ad attori maggiormente diffusa ed utilizzata.

- **Scala**, ideato da Martin Odersky alla École Polytechnique Fédérale de Lausanne (EPFL) tra il 2001 e il 2002, è un linguaggio definito multi-paradigma e ritenuto da alcuni l'unico possibile successore di Java.
- **SALSA**, sviluppato nel 2007 con l'obiettivo di utilizzare il modello ad attori per creare un linguaggio al fine di agevolare le tecniche per la programmazione di applicazioni distribuite.
- **AXUM**, sviluppato nel 2009 nei Microsoft DevLabs.

Oltre alle implementazioni citate, le quali costituiscono veri e propri linguaggi di programmazione, sono degne di nota anche molte librerie (e/o framework) basate sul modello ad attori ed integrate in altri linguaggi non ad attori. Tra esse si possono ricordare:

- **Akka**, innovativa libreria ad attori integrabile in Java, simile al linguaggio SCALA dal quale eredita gran parte delle caratteristiche.
- **Kilim**, framework per la gestione del message passing in modo veloce e sicuro integrabile in Java.
- **ActorFoundry**, sviluppata negli anni 1998-2000 presso l'università dell'Illinoys, risulta essere, probabilmente, la più nota libreria per Java orientata agli attori.
- **HaskellActor**, libreria sviluppata per introdurre il modello ad attori nel linguaggio funzionale Haskell.
- **Celluloid**, un framework ad attori datato 2012, per il linguaggio Ruby.

Tra tutte le varie implementazioni citate, saranno ora descritte ed analizzate brevemente quelle ritenute maggiormente significative. Nel Capitolo 5 poi sarà analizzata dettagliatamente l'implementazione scelta per il caso di studio.

4.1 Linguaggio Erlang

Ideato e sviluppato nell'Ericsson Computer Science Laboratory, Erlang è un linguaggio di programmazione concorrente orientato alla programmazione distribuita, alla gestione del fault-tollerance e alla realizzazione di sistemi software real-time.

Erlang è, tra i più noti linguaggi puramente funzionali, uno dei pochi che introduce specifici costrutti per la programmazione concorrente e distribuita.

Tra le sue caratteristiche principali è noto il meccanismo di dynamic type system, ovvero il meccanismo che consente di verificare runtime il tipo delle espressioni specificate, che rende quindi il linguaggio non tipizzato.

Un banale esempio che dimostra contemporaneamente la semplicità e la potenza offerta da Erlang è illustrato nel seguente Listato 4.1 che realizza un applicazione per il calcolo del fattoriale di un numero.

Listato 4.1: *Semplice applicazione in Erlang*

```
-module(factorial).
-export([fact/1]).

fact(0) -> 1;
fact(N) -> N * fact(N-1).
```

L'applicazione proposta, che può essere eseguita in un ambiente predisposto con un'istruzione del tipo

```
>
> factorial:fact(8).
>
```

evidenza innanzi tutto la completa assenza di tipi di dato, che Erlang prevede unicamente per contesti statici, e dimostra come sia possibile ottenere in pochissime righe di codice una computazione che in un qualunque linguaggio ad oggetti richiederebbe ben più definizioni (creazione di un'apposita classe, definizione del metodo Main, definizione della funzione apposita per la chiamata ricorsiva, ...).

Altra importante caratteristica di Erlang è il supporto all'hot-swapping code per garantire efficienza nelle applicazioni real-time. Si tratta della possibilità di sostituire parte del codice mentre l'applicazione sta computando (ad esempio per eseguire upgrade e manutenzione) senza interrompere l'esecuzione.

4.1.1 Attori in Erlang

In Erlang, essendo un linguaggio intrinsecamente orientato alla concorrenza, gli attori sono parte del linguaggio stesso.

Nato in un contesto orientato alle telecomunicazioni in cui la computazione concorrente risulta essere un aspetto imprescindibile, non sarebbe possibile pensare ad Erlang senza gli attori; i quali sono utilizzati anche per garantire l'aspetto distribuito.

Gli attori in Erlang sono chiamati *process* e vengono avviati mediante la chiamata alla funzione `spawn`.

Un semplice esempio relativo all'uso degli attori in Erlang è presentato nel Listato 4.2, dove viene definito un attore che funge da contatore al quale vengono inviati 100.000 messaggi per l'incremento del contatore e la contestuale richiesta di stampare sulla shell il proprio stato interno (ovvero il valore del contatore).

Listato 4.2: *Uso degli attori in Erlang*

```
-module(counter).
-export([run/0, counter/1]).

run() ->
  S = spawn(counter, counter, [0]),
  send_msgs(S, 100000),
  S.

counter(Sum) ->
  receive
    value -> io:fwrite("Value is ~w~n", [Sum]),
         counter(Sum);
    {inc, Amount} -> counter(Sum+Amount);
  end.

send_msgs(_, 0) -> true;

send_msgs(S, Count) ->
  S ! {inc,1},
  send_msgs(S,Count-1).
```

Nel Listato 4.2, dopo la definizione del modulo e delle funzioni esportate, viene descritta la funzione `run()` la quale permette di avviare l'attore `counter` e di attivare l'invio dei messaggi.

Il contenuto dei messaggi è descritto dalla funzione `send_msg()` la quale utilizza l'operatore specifico per il message-passing ("`!`").

Il corpo dell'attore e il corrispondente comportamento è infine descritto nella funzione `counter()`, la quale attraverso la primitiva `receive`

attende l'arrivo dei messaggi e avvia la computazione corrispondente.

4.1.2 Scheduling

Il linguaggio Erlang utilizza uno scheduler di tipo preemptive per gestire i *process* (che rappresentano gli attori di Erlang). Un tale tipo di scheduler riporta in fondo alla coda d'esecuzione un process dopo che questo è rimasto attivo per un prestabilito periodo di tempo oppure quando invoca la primitiva `receive` senza che siano disponibili messaggi da elaborare.

Relativamente all'aspetto concorrente, Erlang supporta dal 2006 il cosiddetto SMP (symmetric multiprocessing).

Grazie a ciò, Erlang è in grado parallelizzare automaticamente le applicazioni distribuendo la computazione in modo omogeneo su tutti i core a disposizione.

Unico vincolo, relativamente alla gestione degli attori, è che la computazione di un singolo attore non può essere eseguita separatamente su due o più core diversi; è consentito invece eseguire due attori diversi su altrettanti core in modo perfettamente parallelo.

Le statistiche dimostrano che questa tecnica ha permesso ad Erlang di raggiungere un altissimo grado di performance.

4.2 Linguaggio Scala

Scala è un linguaggio di programmazione nato con l'obiettivo di fornire entrambi gli stili di programmazione orientata agli oggetti e funzionale.

Il nome nasce dall'idea che il linguaggio Scala debba essere scalabile, ovvero che possa essere possibile estenderlo facilmente in funzione delle future esigenze.

La duplice offerta di Scala in termini di costrutti rende il linguaggio ottimale per qualunque tipo di applicazione.

Da un lato, la possibilità di scrivere codice in termini funzionali permette la realizzazione di codice chiaro, conciso e performante; dall'altro, la possibilità di utilizzare uno stile orientato agli oggetti garantisce facilità di realizzazione per tutte quelle applicazioni considerate "complesse".

Scala non prevede metodi e campi statici ed utilizza una tipizzazione definibile "leggera" che spesso consente di non specificare il tipo di dato. In scala, ogni valore è un oggetto e ogni operazione può essere ricondotta alla chiamata di un metodo.

Il listato che segue, costituisce un primo esempio in Scala, in cui è definita un'applicazione per il calcolo del fattoriale di un numero.

Listato 4.3: *Semplice applicazione in Scala*

```
object Factorial extends Application {
  def fact(n: Int): Int = {
    if(n == 0) 1
    else n * fact(n-1)
  }

  println("Il fattoriale di 8 vale "+fact(8))
}
```

Nell'esempio di Listato 4.3 è ben visibile la combinazione degli stili funzionale e orientato agli oggetti. Viene definito infatti un oggetto `Factorial` il quale definisce a sua volta un metodo `fact` che, in puro stile funzionale, non prevede la definizione di uno statement per il valore di ritorno (che invece sarebbe previsto per la programmazione orientata agli oggetti).

4.2.1 Attori in Scala

Il modello ad attori è implementato in Scala mediante la libreria `scala.actors`, la quale fornisce tutti i costrutti e tutte le primitive per creare attori e gestire lo scambio di messaggi.

L'esempio che segue, costituisce la versione in Scala dell'esempio di Listato 4.2 proposto in Erlang.

Listato 4.4: *Uso degli attori in Scala*

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

case class Inc(amount: Int)
case class Value

class Counter extends Actor {
  var counter: Int = 0;

  def act() = {
    while(true) {
      receive {
        case Inc(amount) =>
          counter += amount
        case Value =>
          println("Value is "+counter)
```

```
        exit()
      }
    }
  }
}

object ActorTest extends Application {
  val counter = new Counter
  counter.start()

  for(i <- 0 until 100000) {
    counter ! inc(1)
  }
  counter ! Value
}
```

Entrando nel dettaglio dell'esempio, dopo le importazioni della classe `Actor` e dei suoi membri, le istruzioni

```
case class Inc(amount: Int)
case class Value
```

definiscono gli identificatori dei messaggi dell'applicazione. In particolare, si definisce un messaggio `Inc` che prevede un parametro intero e un messaggio `Value` che, invece, non prevede alcun parametro.

L'attore `Counter` è definito mediante le seguenti istruzioni

```
class Counter extends Actor {
  var counter: Int = 0;

  def act() = {
    while(true) {
      receive {
        case Inc(amount) =>
          counter += amount
        case Value =>
          println("Value is "+counter)
          exit()
      }
    }
  }
}
```

Il metodo `act()` ereditato dalla classe base e sovrascritto, definisce il comportamento dell'attore; mentre lo stato interno è mantenuto nel campo `counter`. All'interno di un loop è poi richiamata la primitiva `receive` la quale permette di manipolare i messaggi definiti mediante gli identificatori specificati in precedenza.

La main application è definita invece attraverso le istruzioni

```
object ActorTest extends Application {
  val counter = new Counter
  counter.start()

  for(i <- 0 until 100000) {
    counter ! inc(1)
  }
  counter ! Value
}
```

in cui si costruisce un elemento `Counter` e lo si avvia, quindi vengono inviati i messaggi e, al termine, viene visualizzato il valore finale del contatore. Si noti come l'operatore `!` sia utilizzato per inviare messaggi ad un attore.

Meccanismo Request/Reply

Nell'esempio precedente, dopo aver ricevuto il messaggio, all'attore `Counter` non è richiesto di inviare una risposta all'attore che gli ha inviato il messaggio. Generalmente, invece, questa risulta essere la prassi comune. In Scala, una risposta ad un messaggio può essere inviata mediante la funzione `Response` come illustrato nel seguente frammento di codice.

```
//...
receive {
  case Msg(sender, value) =>
    val res = process(value)
    sender ! Response(res)
}
//...
```

Utilizzata nel modo esposto, tuttavia, la funzione `Response` richiede di conoscere il riferimento all'attore a cui inviare la risposta (`sender` in

questo caso) mediante l'apposito operatore; questo implica che chi invia il messaggio inserisca in uno dei parametri il riferimento a se stesso.

Per ovviare a questo, è disponibile in Scala la funzione `reply()`, utilizzabile nel modo seguente

```
//...
receive {
  case Msg(value) =>
    val res = process(value)
    reply(Response(res))
}
//...
```

In questo caso, quindi, il riferimento all'attore che ha inviato il messaggio è mantenuto internamente alla funzione `replay`.

4.2.2 Attori Thread-based e Attori Event-based

In scala è possibile definire due tipi diversi di attori, i quali si differenziano essenzialmente per il modo in cui vengono gestiti ed eseguiti all'interno della JVM. Esistono infatti gli attori *Thread-based* e quelli *Event-based*.

Gli attori Thread-based sono attori mappati dalla JVM ciascuno in uno specifico thread del sistema. Attori di questo tipo non sono mai d'intralcio alla computazione degli altri attori poiché ogni thread risulta isolato dagli altri.

Tuttavia, il più grosso svantaggio di questo approccio deriva dal fatto che ogni thread occupa un certo spazio in memoria con le proprie informazioni e qualora gli attori in esecuzione fossero un numero molto elevato la computazione di tutti potrebbe essere notevolmente rallentata o addirittura la JVM potrebbe incorrere in problemi di out of memory.

Viceversa, gli attori Event-based, sono eseguiti tutti all'interno di uno stesso thread (o in un ridotto pool di thread). Questo approccio garantisce massima efficienza, anche nel caso in cui l'applicazione preveda un alto numero di attori.

Gli attori Event-based possono essere implementati in Scala utilizzando la primitiva `react` al posto della primitiva `receive`, la quale permette infatti di gestire il flusso d'esecuzione all'interno dello stesso thread in funzione dei messaggi ricevuti.

Naturalmente, è possibile utilizzare contemporaneamente attori Thread-based e attori Event-based al fine di perseguire obiettivi di efficienza, scalabilità e riusabilità.

4.2.3 Messaggi Sincroni e relative problematiche

Generalmente, nel rispetto della filosofia del modello ad attori, tutte le implementazioni definiscono meccanismi per lo scambio di messaggi asincroni.

In realtà, Scala introduce anche il concetto di messaggi sincroni, ovvero messaggi che si bloccano in attesa della risposta. In questo senso, perciò, Scala consente anche di utilizzare messaggi che assomiglino all'invocazione di un metodo, intesa nel senso della programmazione orientata agli oggetti.

Un messaggio sincrono può essere inviato mediante l'operatore `!?` e attraverso la clausola `match` è possibile attendere la risposta; la quale deve necessariamente appartenere ad uno dei tipi specificati nelle clausole identificate dai vari `case`.

```
//...
myService !? Msg(value) match {
  case Response(res) => //...
}
//...
```

Il fatto che un tale messaggio sia di tipo sincrono implica, ovviamente, che un determinato attore che decide di inviarne uno, resti poi bloccato in attesa della risposta.

Sebbene spesso risulti facile utilizzare messaggi sincroni per garantire la sincronizzazione dell'applicazione, l'uso di tale tipo di messaggio può portare al verificarsi di situazioni di deadlock. Si pensi, ad esempio ad un'applicazione nella quale siano definiti due attori, il primo dei quali (attore A) esegua il seguente frammento di codice

```
//Body di actorA

//...
actorB !? Msg1(value) match {
  case Response res) => //...
}
```

```
receive {
  case Msg2(value) => reply(Response2(value))
}
//...
```

mentre il secondo (attore B) esegua, invece, le istruzioni del seguente frammento di codice

```
//Body di actorB

//...
actorA !? Msg2(value) match {
  case Response res) => //...
}

receive {
  case Msg1(value) => reply(Response1(value))
}
//...
```

Risulta evidente come l'applicazione proposta porti a situazioni di deadlock.

In Scala, tuttavia, è possibile evitare questa possibilità utilizzando i messaggi asincroni pur continuando a garantire il sincronismo mediante l'introduzione di loop adatti allo scopo. Per fare ciò, il codice dei due attori deve essere modificato come segue.

```
//Body di actorA

//...
actorB ! Msg1(value)

while(true) {
  receive {
    case Msg2(value) => reply(Response2(value))

    case Response1(res) => //...
  }
}
//...
```

```
//Body di actorB

//...
actorA ! Msg2(value)

while(true) {
  receive {
    case Msg1(value) => reply(Response1(value))

    case Response2(res) => //...
  }
}
//...
```

In questo modo, i messaggi sono di tipo asincrono e pertanto non possono portare a situazioni di deadlock; inoltre, la sincronizzazione è garantita in quanto ogni attore è vincolato sia ad attendere la risposta al proprio messaggio sia ad inviare la risposta al messaggio ricevuto.

4.3 Erlang e Scala: breve confronto

Ricapitolando quanto detto circa i linguaggi che implementano il modello ad attori Erlang e Scala, si può concludere che:

- Erlang è un linguaggio puramente funzionale che fornisce molte più funzionalità rispetto ai linguaggi funzionali classici. Ciò garantisce maggiore espressività e semplicità per la realizzazione delle applicazioni, soprattutto in relazione agli aspetti di concorrenza e distribuzione.

- Scala fa invece uso sia di un approccio orientato agli oggetti sia di un approccio funzionale. Questo rende molto agevole la scrittura del codice, ma può anche portare a problemi ai quali bisogna prestare particolare attenzione al fine di evitarli.

Il maggiore divario tra Erlang e Scala è quindi rappresentato proprio dallo stile di programmazione scelto e dalle conseguenze che questo comporta.

Se da un lato, l'aver reso disponibile uno stile orientato agli oggetti mescolato con uno stile funzionale rende un applicazione scritta in Scala potenzialmente soggetta a diversi side effect dovuti, ad esempio,

alla situazione in cui mediante la predisposizione di metodi o funzioni apposite sia possibile rendere pubblico e modificabile lo stato interno di un attore (possibilità peraltro espressamente vietata dal modello ad attori); dall'altro, la rigidità di Erlang vieta qualunque altra forma di comunicazione che non sia basata sullo scambio asincrono di messaggi (circostanza senz'altro in linea col modello ad attori, ma a volte poco adattabile a specifiche situazioni reali).

In ogni caso, comunque, entrambi i linguaggi implementano il modello ad attori in modo semplice e ciò favorisce molto la realizzazione di applicazioni concorrenti e orientate al parallelismo [11].

4.4 Libreria ActorFoundry

La prima versione di ActorFoundry fu inizialmente progettata e sviluppata presso l'*Open Systems Laboratory* dell'università dell'Illinois nel periodo 1998-2000 con l'obiettivo di sviluppare un framework basato sul modello ad attori da inserire in un nascente linguaggio object oriented chiamato Java.

Il framework ActorFoundry fornisce un semplice modello nel quale il controllo e la gestione delle mailbox è integrato e nascosto all'interno del framework stesso, lasciando allo sviluppatore il solo compito di gestire lo stato interno degli attori nonché il loro comportamento in relazione ai messaggi ricevuti.

Per implementale la semantica del modello ad attori, ActorFoundry fornisce essenzialmente le seguenti tre API:

- `send(actorAddress, message, arguments)`, che permette di inviare un messaggio asincrono all'attore identificato dall'indirizzo specificato.

- `call(actorAddress, message, arguments)`, che permette di inviare un messaggio asincrono ad un attore ed attenderne la risposta, che sarà costituita da un altro messaggio asincrono.

- `create(node, behavior, arguments)`, che permette di creare un nuovo attore presso uno specifico nodo con il comportamento indicato[1].

[1] L'indicazione del nodo in cui si vuole creare l'attore è opzionale; qualora non venga specificato, il nuovo attore sarà creato nel contesto locale.

Essendo un framework ad attori integrato nel linguaggio object oriented Java, ActorFoundry mappa ogni attore in uno specifico thread java; mentre i messaggi sono consegnati agli attori mediante l'API Java `Reflection`. Ogni messaggio può contenere qualunque oggetto Java a condizione che questo implementi l'interfaccia `java.lang.Serializable`: i messaggi sono trasmessi infatti utilizzando il meccanismo di serializzazione.

4.4.1 Esempio d'uso

Affinché sia possibile comprendere meglio il funzionamento degli strumenti messi a disposizione da ActorFoundry, si consideri il seguente esempio.

Listato 4.5: *Primo esempio in ActorFoundry*

```
package osl.examples.helloworld;

import osl.manager.Actor;
import osl.manager.ActorName;
import osl.manager.RemoteCodeException;
import osl.manager.annotations.message;

public class HelloActor extends Actor {
  @message
  public void hello() throws RemoteCodeException {
    ActorName otherActor = null;
    call(stdout, "print", "Hello ");
    otherActor = create(WorldActor.class);
    send(other, "world");
  }
}

public class WorldActor extends Actor {
  @message
  public void world() {
    send(stdout, "println", "World!");
  }
}

public class BootActor extends Actor {
  @message
  public void boot() throws RemoteCodeException {
```

```
    ActorName helloActor = create(HelloActor.class);
    send(helloActor, "hello");
  }
}
```

A prima vista, si può immediatamente notare che il programma scritto in ActorFoundry usa la stessa sintassi e gli stessi componenti del linguaggio Java.

Per imporre il fatto che una specifica classe rappresenti un attore, è fondamentale che questa estenda la classe `Actor`. Inoltre, per identificare i messaggi che possono essere inviati all'attore è necessario dichiarare il metodo corrispondente con l'identificatore `@message`.

Nello specifico dell'esempio proposto, sono stati dichiarati tre attori. L'attore `BootActor` crea un'istanza dell'attore `HelloActor` e invia ad esso il messaggio `hello` senza attenderne risposta mediante la primitiva `send`.

Quando l'attore riceve il messaggio esegue la sua computazione che comprende l'invio di un messaggio all'attore `stdout` e la creazione di un nuovo attore `otherActor`, istanza di `WorldActor`, al quale viene inviato un altro messaggio.

Grazie al diverso funzionamento delle primitive `send` e `call` è possibile scongiurare un comportamento non atteso come l'inversione dei messaggi stampati a video.

4.4.2 Meccanismo per l'invio di messaggi

Come è stato annunciato in precedenza, in ActorFoundry i messaggi sono inviati mediante le primitive `send` e `call`. La differenza esistente tra le due primitive è basata sul fatto che la primitiva `send` invia un messaggio e termina la propria computazione, mentre la primitiva `call` termina la propria computazione solo dopo aver ricevuto conferma dell'avvenuta elaborazione del messaggio inviato.

In ogni caso comunque, entrambe le primitive sono in grado di inviare un messaggio grazie ad un meccanismo di matching che lega la stringa specificata come parametro della primitiva al nome del metodo che rappresenta il messaggio. Con riferimento all'esempio di Listato 4.5, si noti che l'istruzione

```
send(helloActor, "hello");
```

può inviare il messaggio `hello` all'attore `helloActor`, istanza di `HelloActor`, in quanto nella classe che rappresenta l'attore `HelloActor` è definito un metodo (identificato come messaggio) di nome `hello()`.

```
@message
public void hello() throws RemoteCodeException {
  //...
}
```

È chiaro quindi che l'invio di messaggi in ActorFoundry avviene confrontando la stringa specificata nella primitiva d'invio con tutti i nomi dei metodi identificati come messaggio nella classe dell'attore specificato. Se il metodo cercato esiste, il messaggio viene considerato inviato, viceversa l'invio fallisce.

4.5 Analisi delle Performance

Concludendo l'analisi di alcune tra le più diffuse implementazioni del modello ad attori, risulta interessante l'esperimento condotto dai ricercatori Karmani, Shali e Agha nel loro articolo [6].

Attraverso un'applicazione di riferimento denominata *ThreadRing*, nella quale 503 entità concorrenti si scambiano un token in una struttura ad anello per 10 milioni di passaggi mediante il meccanismo a scambio di messaggi, vengono stimati i tempi per la creazione di attori, per la gestione del message passing e del context-switch.

Eseguendo l'applicazione[2] su un'architettura Intel Core 2 Duo da 2.40GHz (con 4GB di RAM e 3MB di L2 cache a disposizione) i risultati hanno evidenziato che ad ActorFoundry servono 695s per eseguire completamente la computazione, mentre ad Erlang servono solamente 8s e poco di più a Scala.

Tali risultati dimostrano come un'implementazione assolutamente pulita e coincidente al modello ad attori teorico (come quella di ActorFoundry, nella versione di base) può portare a performance più scadenti rispetto ad altre implementazioni che invece effettuano alcuni accorgimenti e miglioramenti.

Il grafico di Figura 4.1 fornisce una visione completa dei risultati. In particolare, si noti come, per la versione Erlang e Scala di Threa-

[2] Ovviamente, sono state realizzate diverse applicazioni perfettamente analoghe ma scritte nei vari linguaggi che si volevano testare.

dRing, il tempo d'esecuzione subisca lievi aumenti in funzione di notevoli incrementi del numero di messaggi scambiati.

Viceversa, le performance di ActorFoundry risultano decisamente scadenti in questa prima sessione di test.

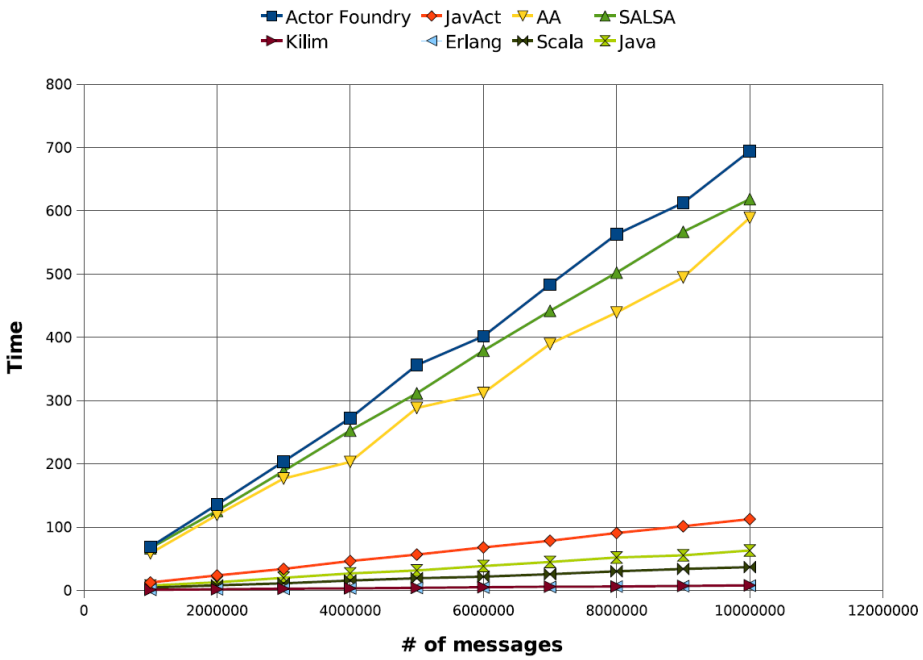

Figura 4.1: *Performance relative all'applicazione ThreadRing mediante diverse implementazioni del modello ad attori*

È necessario specificare che la versione di ActorFoundry considerata nel test proposto è la cosiddetta versione base, mentre oggi la versione che comunemente viene considerata ed utilizzata è quella che prevede la sua integrazione con alcune funzionalità offerte da Kilim (come ad esempio le trasformazioni CPS).

Alla luce di ciò, ripetendo l'esperimento utilizzando questa "nuova" versione di ActorFoundry, i risultato sono molto più confortanti e confrontabili.

Il grafico riportato in Figura 4.2 evidenza i risultati del test eseguito considerando la versione integrata di ActorFoundry.

In esso si può notare, in particolare, come questa integrazione porti ActorFoundry ad avere performance addirittura lievemente migliori di quelle di Scala e, in definitiva, renda la scelta di integrare alcune caratteristiche di Kilim in ActorFoundry decisamente vincente.

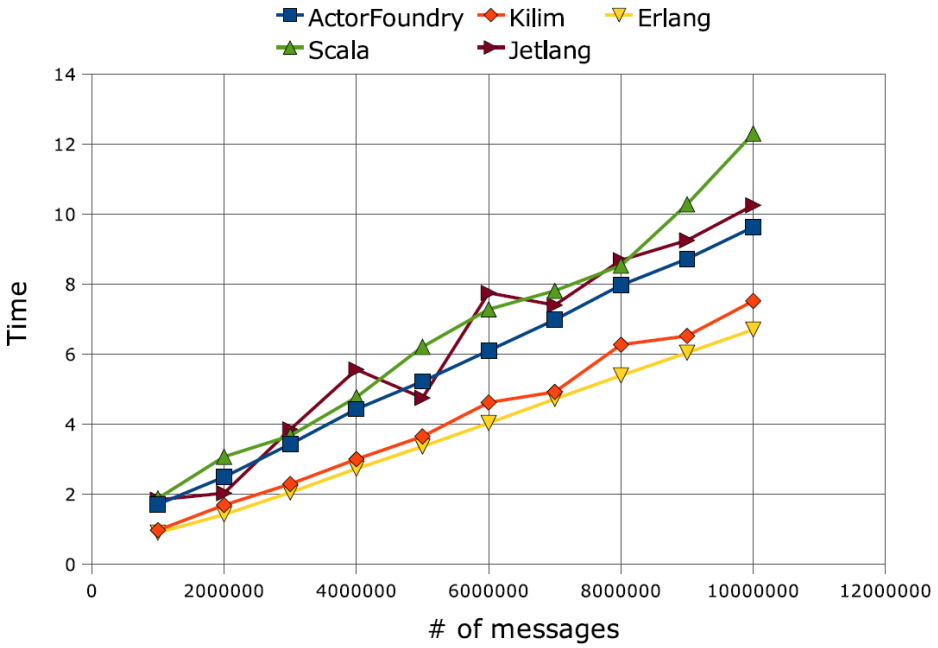

Figura 4.2: *Performance relative all'applicazione ThreadRing considerando l'integrazione di Kilim in ActorFoundry*

Si noti che in entrambi i grafici[3] proposti, oltre ad altre implementazioni, sono riportate anche le performance di Kilim le quali risultano molto simili a quelle ottenute mediante l'uso di Erlang. Tuttavia, l'integrazione tra Kilim e ActorFoundry è stata preferita al fine di sfruttare la grande popolarità e la rinomata semplicità di AcotorFoundry stessa.

[3]I grafici con l'analisi delle performance sono stati tratti dall'articolo di R. Karmani, A. Shali e G. Agha *"Actor Frameworks for the JVM Platform: A comparative Anlysis"*. Inoltre, altri dati utili possono essere rintracciati all'indirizzo `http://shootout.alioth.debian.org/`

Capitolo 5

AXUM: linguaggio ad attori

Nato nel Maggio del 2009 nei Microsoft DevLabs con il nome in codice *Maestro*, Axum è un linguaggio di programmazione basato sul modello ad Attori e sviluppato da Microsoft Corporation.

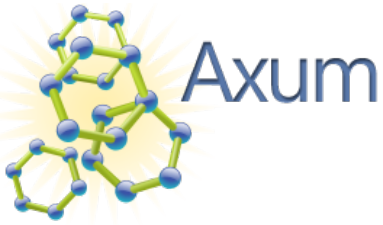

Figura 5.1: *Logo*

L'idea alla base di Axum è quella di creare un linguaggio semplice, scalabile e sicuro con l'obiettivo di rendere agevole la progettazione di applicazioni concorrenti in ambiente .NET Framework. In altre parole, se un'applicazione viene studiata, analizzata e progettata in termini di componenti interattivi, la sua codifica in Axum risulta immediata e, soprattutto, priva della maggior parte di bug derivanti dalle problematiche legate alla concorrenza.

In definitiva, l'obiettivo primario del linguaggio Axum è quello di consentire al programmatore un approccio implementativo senza occuparsi né preoccuparsi degli aspetti concorrenti dell'applicazione; tali aspetti, infatti, sono presenti intrinsecamente nella struttura di qualunque applicazione Axum, garantendo in tal modo all'applicazione stessa velocità d'esecuzione, alta reattività e maggiore efficienza.

Inoltre, Axum disciplina in modo molto rigoroso l'utilizzo di un eventuale spazio di memoria condivisa, affinché sia possibile prevenire l'intera

totalità dei più comuni errori di programmazione, evitando così situazioni spiacevoli e potenzialmente dannose.

Axum propone una sintassi simile a quella del linguaggio C# con il quale, tra l'altro, può facilmente interagire ed integrarsi. Per tale motivo quindi, nella trattazione che segue saranno tralasciati tutti quegli aspetti che possono essere facilmente compresi mediante la conoscenza del linguaggio C# (supposta acquisita) e saranno invece approfonditi gli aspetti rilevanti e distintivi del linguaggio Axum in relazione al suo obiettivo primario di realizzazione di applicazioni intrinsecamente concorrenti.

5.1 Uno sguardo d'insieme

Dal punto di vista ingegneristico, uno dei metodi per descrivere un linguaggio di programmazione è rappresentato dal cosiddetto *metamodello del linguaggio* in cui sono riportati gli elementi caratterizzanti il linguaggio e le relazioni che intercorrono tra tali elementi.

In particolare, un metamodello descrive come le varie componenti di un linguaggio sono tra loro interconnesse e, quindi, come possono essere utilizzate per la costruzione di applicazioni software attraverso l'uso del linguaggio a cui il metamodello si riferisce.

Relativamente ad Axum, non esiste un metamodello ufficiale fornito dagli ideatori; tuttavia, attraverso le specifiche [9] del linguaggio è possibile costruire un metamodello di Axum, mediante il quale avere un iniziale visione d'insieme delle componenti che lo costituiscono.

Dettagliatamente, risultano essere componenti fondanti di Axum i seguenti concetti:

- **agent** - rappresenta l'entità principale del linguaggio. Il suo comportamento è simile al concetto di classe nei linguaggi object oriented con la differenza che il riferimento ad un'istanza di un *agent* non può essere utilizzato ovunque. Inoltre, gli *agent* interagiscono tra loro mediante il meccanismo di scambio di messaggi asincrono attraverso i *channel*.

- **channel** - rappresenta il canale di comunicazione mediante il quale è possibile comunicare con un preciso *agent*.

- **port** - costituiscono gli accessi in ingresso e in uscita ad un determinato *channel*.

- **domain** - rappresenta un meccanismo per raggruppare determinati *agent* e offrire loro un più performante, ma potenzialmente non esente da side effect, metodo per comunicare.

- **schema** - permette la definizione dei dati, e quindi dei messaggi, ammessi attraverso specifiche *port* relative a specifici *channel*.

In Figura 5.2 è rappresentato un possibile metamodello per il linguaggio Axum.

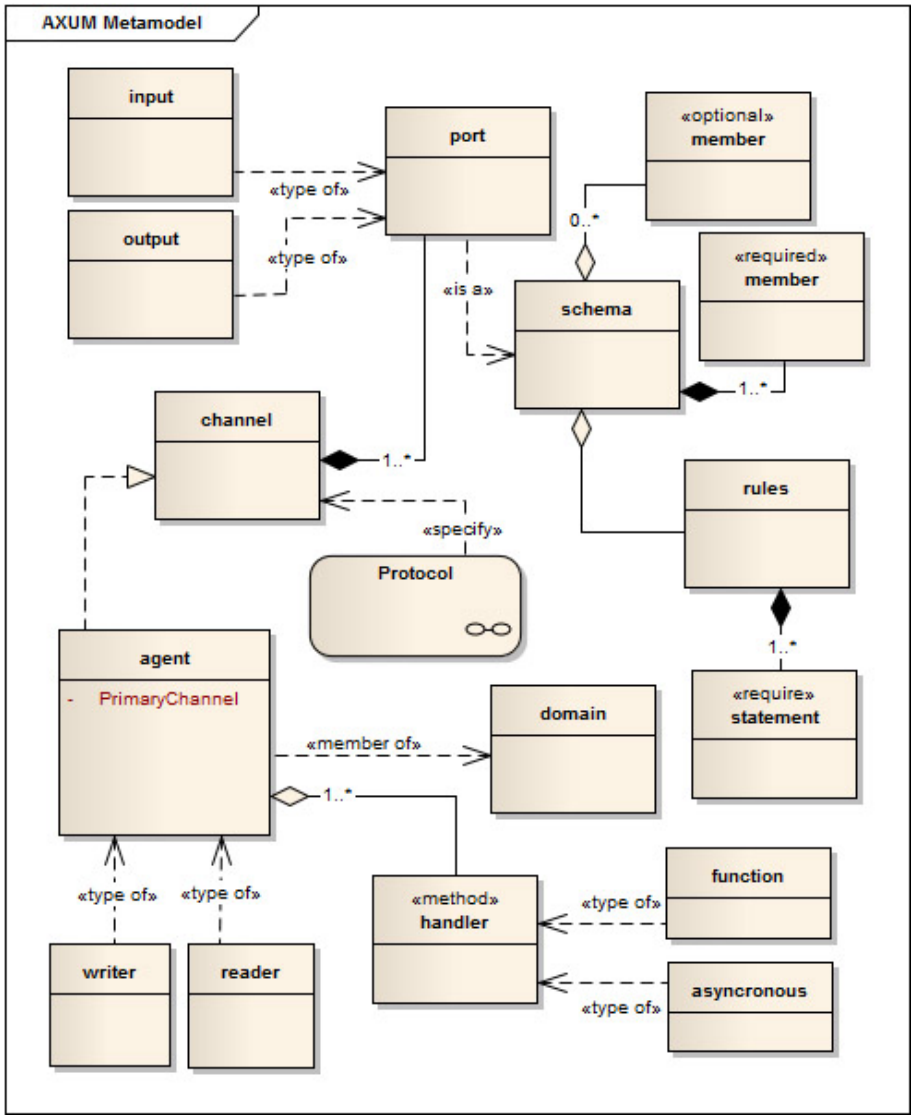

Figura 5.2: *Metamodello del linguaggio AXUM*

Tramite il metamodello si può notare immediatamente come le varie componenti si relazionino tra loro e quali siano i tipi associati a alcune di queste. Inoltre, è chiaro come certe componenti siano composte obbligatoriamente da altre oppure come alcune di queste siano associate tra loro.

In ogni caso, comunque, il metamodello non esaurisce la trattazione del linguaggio, la quale sarà completata in modo rigoroso nei successivi paragrafi.

Tuttavia, prima di procedere alla descrizione della sintassi del linguaggio e della relativa semantica, si propone una banale applicazione in Axum che visualizza un messaggio sulla shell a caratteri.

Listato 5.1: *Applicazione "Hello, World!"*

```
using System;

agent Program : Microsoft.Axum.ConsoleApplication
{
  override int Run(String[] args)
  {
    Console.WriteLine("Hello, Axum World!");
  }
}
```

È fondamentale chiarire immediatamente che **il concetto di attore in Axum è rappresentato da un componente chiamato Agente (agent)**, ma che tale denominazione non è in nessun modo collegata al concetto di *agente software* definito nel modello di programmazione ad agenti. Come nei linguaggi basati sul paradigma ad oggetti dove il concetto di oggetto è rappresentato da un componente denominato *classe*; in Axum, che invece è un linguaggio basato sul paradigma ad attori, il concetto di attore è rappresentato da un componente denominato *agente*.

Nonostante Axum abbia introdotto componenti non presenti nella definizione classica del modello ad attori, esso resta comunque un linguaggio profondamente diverso dai linguaggi che implementano il paradigma ad oggetti. Infatti, ogni agente non prevede metodi pubblici mediante i quali ottenere informazioni relative allo stato interno dell'agente e/o modificare tale stato. Inoltre, non è possibile invocare metodi su un agente ed attendere che esso completi la propria computazione prima di procedere. L'unico meccanismo di comunicazione ammesso tra agenti è quello basato sullo scambio (asincrono) di messaggi.

Focalizzando brevemente l'attenzione sull'esempio di Listato 5.1, si osservi che, dopo la dichiarazione di alcune librerie necessarie, è stato definito un agente (denominato `Program`) che deriva[1] da un agente presente nella libreria `Microsoft.Axum` chiamato `ConsoleApplication` e ne sovrascrive il metodo principale `Run()` con il quale è possibile dare il via all'applicazione. All'interno del metodo `Run()` viene inserita la logica dell'applicazione che permette di stampare un messaggio sulla shell.

5.2 Meccanismo di Asyncronous Message Passing

In Axum, ogni agente lavora su dati associati a messaggi, i quali viaggiano all'interno dell'applicazione attraverso ben definiti canali di comunicazione tra i vari agenti.

Senza entrare nel dettaglio delle specifiche (che sarà affrontato in seguito) è opportuno tuttavia definire come gli agenti possano scambiare messaggi. In Axum ciò avviene mediante i concetti di *channel* e *port*.

Listato 5.2: *Uso dei channels in Axum*

```
using System;

agent Program : channel Microsoft.Axum.Application
{
  public Program()
  {
    String[] args = receive(PrimaryChannel::CommandLine);

    Console.WriteLine("Hello, Axum World!");

    PrimaryChannel::ExitCode <-- 0;
  }
}
```

L'applicazione appena proposta (che ha lo stesso comportamento di quella precedente) implementa il canale `Microsoft.Axum.Application`

[1] Il concetto di *derivazione* differisce (anche se in termini molto sottili) dal concetto di *ereditarietà* presente nell'OOP. Infatti, quando un agente deriva da un altro agente ne estende il comportamento, sovrascrivendo alcuni metodi virtuali e/o aggiungendone di nuovi. Lo stato interno tuttavia resta distinto; cioè l'agente che deriva, non può in nessun caso accedere allo stato interno dell'agente dal quale deriva.

mediante la keyword `channel`. Implementare un canale significa "agganciarsi" al cosiddetto *implementing end* del canale e diventare così il server dei messaggi che passeranno per il canale [8]. L'altro lato del canale, detto *using end* è visibile solamente ai client del canale che tipicamente sono rappresentati da altri agenti del sistema.

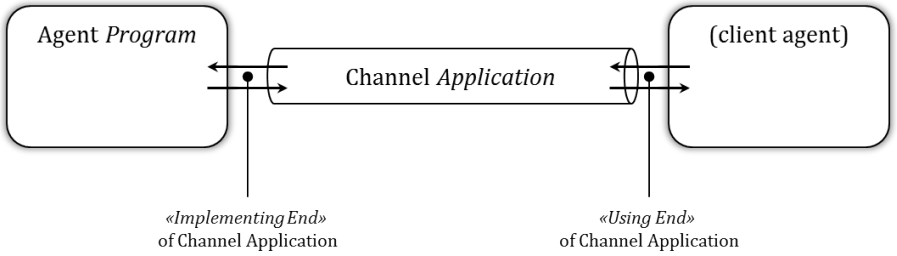

Figura 5.3: *Rappresentazione grafica del canale implementato nell'applicazione proposta, con l'identificazione dei due lati del canale.*

In generale, i canali hanno la funzione di permettere agli agenti lo scambio di messaggi. Durante la definizione di un canale vengono descritte delle regole relativamente ai tipi di messaggi e di dati che possono entrare e uscire dal canale. Ovviamente, a differenza degli agenti, i canali non effettuano nessuna modifica sui dati contenuti nei messaggi, limitandosi cioè a trattarne il trasferimento.

Riferendosi nuovamente all'applicazione descritta nel Listato 5.2, il client agent collegato all'*using end* del canale è implementato a runtime. Esso istanzia l'agent `Program`, quindi invia sulla porta `CommandLine` del canale i parametri della shell a linea di comando e, infine, si pone in attesa di un messaggio sulla porta `ExitCode`. Quando tale messaggio sarà recapitato, l'applicazione terminerà.

L'agente `Program`, invece, una volta creato, attende l'arrivo (mediante la primitiva `receive`) dei parametri della shell sulla porta specificata; ricevuti i dati, esegue la propria computazione e segnala lo stato di terminazione della computazione inviando un messaggio sulla porta `ExitCode` del canale.

Si noti come, l'invio del messaggio avvenga posizionando i dati sulla porta mediante l'operatore "`<--`", mentre l'operatore "`::`" sia utilizzato per accedere ad una porta di uno specifico canale.

Infine, si noti anche che l'operazione di ricezione di un messaggio è bloccante per l'agente che la invoca (l'agente resta in attesa della ricezione di un messaggio e fino a che questo non giunge sulla porta specificata l'agente non esegue alcuna computazione), mentre l'operazione di invio

di un messaggio è effettuata in modo asincrono (il messaggio viene inviato e non viene attesa nessuna conferma di ricezione[2]).

5.3 Elementi per la comunicazione

Sebbene l'elemento centrale su cui si basa il linguaggio Axum sia costituito dal concetto di `Agent`, prima di poter affrontarne una descrizione dettagliata è necessario descrivere dettagliatamente la semantica dei canali di comunicazione, in quanto essi consentono agli agenti di comunicare senza preoccuparsi della rispettiva implementazione. In analogia col modello di programmazione orientato agli oggetti, si può affermare che i canali definiscono il cosiddetto "contratto" che nel mondo OOP è descritto dalle interfacce.

5.3.1 `Channel` e `Port`

Come già annunciato, un canale rappresenta l'elemento che permette la comunicazione tra gli agenti del sistema. Formalmente esso è descritto mediante una collezione di porte che costituiscono l'accesso al canale.

Quando un'agente usa un canale, esso invia i propri messaggi sul canale attraverso le porte di input e riceve i messaggi a lui destinati attraverso le porte di output. Quando un agente implementa un canale, le porte di input sono assimilabili alle sorgenti del canale, mentre quelle di output sono assimilabili alla destinazione. Ogni porta porta può comportarsi sia come porta di input sia come porta di output a seconda di quale agente stia utilizzando il canale.

Generalmente, la connessione ad un canale è effettuata tramite l'*using end side* del canale. Il client agent infatti dichiara un canale con una sintassi del tipo

```
var myAdderChannel = new AdderChannel("adderchannel");
```

dove viene creata una variabile `myAdderChannel` che contiene l'indirizzo di un canale di tipo `AdderChannel`. Naturalmente, si suppone che il canale `AdderChannel` sia stato opportunamente definito.

La definizione di un canale avviene attraverso la descrizione delle porte che lo compongono. Per ogni porta va specificato se questa deve

[2]La conferma di ricezione è implicita nel fatto che, poiché si fa uso di un canale esso garantisce la trasmissione del messaggio.

essere una porta di input o di output per l'agente che andrà ad implementare il canale. Volendo definire il canale `AdderChannel`, una possibile implementazione potrebbe essere

```
channel AdderChannel
{
  input int Addend1;
  input int Addend2;
  output int Sum;
}
```

nella quale il canale è definito con tre porte; la prima e la seconda, denominate rispettivamente `Addend1` e `Addend2`, che possono ricevere messaggi contenenti un valore intero, la seconda, denominata `Sum`, mediante la quale si può spedire un messaggio contenente un valore numerico intero.

Si noti che, sebbene i nomi delle porte ricordino quello che presumibilmente sarà il comportamento dell'agente che andrà ad implementare il canale, questo non è vincolante. L'unico vincolo (*contratto*) è quello sul tipo di informazioni che un messaggio inviato attraverso il canale può contenere; nulla è definito relativamente alla semantica.

ESEMPIO

Si propone ora un esempio di utilizzo del canale appena creato. In particolare, si vuole realizzare una piccola applicazione che visualizzi a video la somma di due interi ottenuti tramite i parametri dell'applicazione.

Listato 5.3: *Esempio d'uso degli elementi* `channel` *e* `port`

```
using System;
using System.Concurrency;
using Microsoft.Axum;

channel AdderChannel
{
  input int Addend1;
  input int Addend2;
  output int Sum;
}

agent AdderAgent : channel AdderChannel
{
```

```
  public AdderAgent()
  {
    int add1 = receive(PrimaryChannel::Addend1);
    int add2 = receive(PrimaryChannel::Addend2);

    int result = add1 + add2;

    PrymaryChannel::Sum <-- result;
  }
}

agent MainAgent : channel Microsoft.Axum.Application
{
  public MainAgent()
  {
    String[] args = receive(PrimaryChannel::CommandLine);

    var adder = AdderAgent.CreateInNewDomain();

    adder::Addend1 <-- Convert.ToInt32(args[0]);
    adder::Addend2 <-- Convert.ToInt32(args[1]);

    int sum = receive(adder::Sum);

    Console.WriteLine("{0} + {1} = {2}", args[0], args[1], sum);

    PrimaryChannel::ExitCode <-- 0;
  }
}
```

Tralasciando l'aspetto relativo alla creazione e all'uso degli agenti che sarà trattato in seguito, l'esempio proposto nel Listato 5.3 mette in evidenza come il canale `AdderChannel` (implementato dall'`AdderAgent` ed utilizzato dal `MainAgent`) venga utilizzato per la comunicazione tra i due agenti. Per comprendere la logica d'esecuzione si ricordi quanto detto nel Paragrafo 5.2 relativamente al carattere bloccante della primitiva `receive`.

5.3.2 Tipi non ammessi e `Schema`

Contrariamente a quanto si potrebbe pensare, in Axum, i messaggi che gli agenti si scambiano attraverso i canali non possono contenere qua-

lunque tipo di dato; infatti solo una piccola porzione dell'insieme dei tipi previsti dalla piattaforma .NET Framework è accettata. In particolare sono accettati solo quei tipi di dato che sono immutabili e completamente serializzabili.

Le motivazioni di tale scelta sono dovute principalmente alle seguenti due osservazioni:

- Affinché sia possibile eseguire una computazione concorrente e, poiché il modello ad attori non prevede uno spazio delle informazioni condivise (*shared memory*), è fondamentale che gli agenti si scambino solo dati di natura non mutevole. Se così non fosse, sarebbe possibile ottenere inconsistenza nelle informazioni scambiate.

- In un ambiente distribuito (che, sebbene non necessario, resta senza dubbio il target principale del linguaggio Axum e del modello ad attori in generale) due o più agenti possono trovarsi su diversi nodi della rete. Per questo motivo, è fondamentale che i messaggi possano essere serializzati per essere trasmessi attraverso la rete e ciò risulta possibile solamente se tali messaggi contengono dati appartenenti a tipi che possano essere serializzati.

Affinché siano sempre garantite le caratteristiche di immutabilità e serializzabilità per i tipi di dato, Axum introduce il concetto di *Schema*.

Un elemento etichettato con la keyword `schema` rappresenta un tipo di dato che definisce un insieme di campi (alternativamente obbligatori o opzionali) ma non introduce alcun metodo per lavorare su tali campi.

Un dato di tipo `schema` è definito con una sintassi del tipo

```
schema Person
{
  required string Firsname;
  required string Surname;
  required int Age;
  optional string Address;
  optional string PhoneNumber;
}
```

Per come è stato definito, un tipo di dato `schema` garantisce intrinsecamente il fatto di essere immutabile e serializzabile[3].

[3]Similmente a quanto accade per i tipi di dato di XMLSchema.

La presenza di campi obbligatori e campi opzionali permette, inoltre, di trasmettere in un messaggio più tipi di `schema` a patto che tutti quelli che si vogliono trasmettere abbiano almeno lo stesso tipo e lo stesso numero di campi obbligatori.

Infine, i dati definiti come `schema`, sono spesso utilizzati per incapsulare all'interno di un unica struttura un insieme di informazioni[4].

Relativamente alla creazione di un'istanza di una variabile di tipo `schema`, si fa uso di un costruttore esplicito che, attraverso la sintassi delle proprietà *setter* introdotte nel linguaggio C#, permette di definire i valori di tutti i vari elementi del dato.

In particolare si osservi la seguente creazione di un'istanza del dato `Person`, definito in precedenza, in cui sono specificati necessariamente tutti i valori dei campi *obbligatori* ma solo in parte quelli dei campi *opzionali* (che potevano anche essere completamente omessi o completamente definiti).

```
var person1 = new Person{
        Firstname="Mario",
        Surname="Rossi",
        Age="28",
        PhoneNumber="0005536" };
```

Un'ulteriore possibilità per i dati di tipo `schema` è quella di definire delle regole sui dati. In questo modo, perciò, risulta estremamente efficace ed immediato effettuare controlli sui dati scambiati tra gli agenti e imporre particolari condizioni.

Nella definizione proposta di `Person` è stato specificato che il campo `Firstname`, ad esempio, debba essere obbligatorio. Tuttavia, nessuno impedisce di inviare un riferimento nullo o una stringa vuota per tale campo. Entrambe le possibilità sarebbero, invece, evitate definendo una regola mediante il costrutto `rules`, come illustrato di seguito.

```
schema Person
{
  required string Firsname;
  required string Surname;
  required int Age;
  optional string Address;
  optional string PhoneNumber;
```

[4]Funzionalità che può essere associata alle strutture del linguaggio C o del linguaggio C# , sebbene queste ultime generalmente non vengano poi serializzate.

```
  rules
  {
    require !String.IsNullOrEmpty(Firstname);
    require !String.IsNullOrEmpty(Surname);
    require (Age > 0);
  }
}
```

5.3.3 Request Correlator e porte Request-Reply

A causa dell'asincronismo nell'invio dei messaggi, all'agente che si trova allo *using end* di un canale si pone il problema di correlare le risposte ricevute a fronte dell'invio di più messaggi. Infatti, quando un agente invia più messaggi ad un altro agente, non è vincolante che quest'ultimo risponda ai messaggi nell'ordine con cui sono giunti a lui. Per questo motivo è fondamentale avere un meccanismo che consenta, a chi invia i messaggi, di essere in grado di far corrispondere le risposte ottenute con i messaggi inviati. Tale meccanismo, in Axum, è chiamato *Request Correlator* e si basa sul concetto di porte di tipo *Request-Reply*.

Una porta di tipo Request-Reply è una porta che prevede la sola specificazione dei parametri di input del messaggio e restituisce la risposta al messaggio tramite un correlatore dichiarato in fase di invio del messaggio. Il tipo del correlatore è specificato in fase di creazione del canale.

ESEMPIO

In questo esempio si vuole costruire una variante di quanto proposto nell'esempio di Listato 5.3, aggiungendo la possibilità per l'agente `AdderAgent` di accettare più messaggi e terminare la propria computazione solo ad applicazione conclusa e non dopo aver risposto al primo messaggio ricevuto.

Listato 5.4: *Esempio di Request-Correlator*

```
using System;
using System.Concurrency;
using Microsoft.Axum;

schema Addends
{
  required int Add1;
```

```
  required int Add2;
}

channel AdderChannel
{
  input Addends Numbers : int;
}

agent AdderAgent : channel AdderChannel
{
  public AdderAgent()
  {
    while(true)
    {
      var result = receive(PrimaryChannel::Numbers);
      result <-- result.RequestValue.Add1
                 + result.RequestValue.Add2;
    }
  }
}

agent MainAgent : channel Microsoft.Axum.Application
{
  public MainAgent()
  {
    var adder = AdderAgent.CreateInNewDomain();

    var coorrelator1 = adder::Numbers
                       <-- new Addends{Add1=5, Add2=10};
    var coorrelator2 = adder::Numbers
                       <-- new Addends{Add1=8, Add2=6};

    var sum1 = receive(correlator1);
    var sum2 = receive(correlator2);

    Console.WriteLine("Sum1 is {0} - Sum2 is {1}", sum1, sum2);

    PrimaryChannel::ExitCode <-- 0;
  }
}
```

Nell'esempio proposto due sono gli aspetti degni di nota. Il primo risiede nella creazione del canale dove si può notare la definizione

di una porta di tipo Request-Reply (riportata di seguito per maggiore chiarezza).

```
channel AdderChannel
{
  input Addends Numbers : int;
}
```

Una tale porta è riconoscibile in quanto prevede la definizione del input (elemento `Numbers`, di tipo `Addends` definito da uno `schema`) mediante un nome e il relativo tipo, mentre la definizione dell'output correlato a quell'input mediante la sola definizione del tipo. Infatti, per il meccanismo di Request Correlator non è necessario far riferimento alla porta di output specificandone il nome.

Il secondo aspetto degno di nota è il modo con il quale il `MainAgent` invia i messaggi e ne attende risposta. Tale agente infatti, all'atto dell'invio di un messaggio ottiene un riferimento (correlator) che risulta necessario per associare le risposte in fase di ricezione.

5.3.4 Protocolli di comunicazione

Fino a questo momento, si è sempre supposto che ogni agente invii un messaggio completo sul canale. Inviare un messaggio completo significa inviare un messaggio che contenga tutti i dati previsti dalle porte che sono state specificate.

Purtroppo, però, questo non può mai essere assunto come vero sempre. Potrebbe accadere infatti che un agente per qualche motivo si metta ad attendere la risposta ad un messaggio senza aver prima inviato tutti i dati necessari. Questo fa si che l'agente dall'altra parte del canale resti in attesa dell'arrivo di tutti i dati prima di procedere alla risposta. Tale situazione tuttavia non si verificherà mai poiché l'agente che ha inviato il messaggio risulta in stato di attesa a causa della primitiva receive. Questo porta al verificarsi di una spiacevole situazione di deadlock.

Per prevenire situazioni come quella proposta, è necessario definire un protocollo per l'invio dei messaggi; ovvero deve essere stabilito l'ordine di invio dei dati che, qualora non venisse rispettato, sia in grado di segnalare una violazione.

La definizione di un protocollo per uno specifico canale avviene mediante la definizione di una *Macchina a Stati (ASF)* in cui sono specificate le sole transizioni possibili.

Riprendendo l'esempio di Listato 5.3, affinché siano previste e conseguentemente evitate le situazioni di deadlock, è necessario modificare la definizione del canale nel modo seguente.

```
channel AdderChannel
{
  input int Addend1;
  input int Addend2;
  output int Sum;

  Start: {Addend1 -> gotAddend1_state;}
  gotAddend1_state: {Addend2 -> gotAddend2_state;}
  gotAddend2_state: {Sum -> End;}
}
```

La macchina a stati inizia con la parola chiave `Start` e termina con la parola chiave `End`. Entrambe definiscono rispettivamente lo stato iniziale e finale del protocollo. Per ogni stato è definita la corrispondente transizione a fronte della ricezione di un predefinito dato.

Nella ridefinizione di `AdderChannel` proposta, è stato specificato che inizialmente (stato `Start`) si debba necessariamente ricevere il valore di `Addend1` e contestualmente transitare verso lo stato `gotAddend1_state`. Analogamente si procede per gli stati successivi fino alla terminazione dell'automa che costituisce il protocollo del canale.

Si noti che qualunque messaggio che non rispetti il protocollo previsto solleva un eccezione che, se opportunamente gestita, permette di evitare la situazione di deadlock.

5.4 Realizzazione di Agenti

Dopo aver discusso le modalità di comunicazione tra agenti e le relative problematiche, è ora necessario approfondire gli aspetti relativi alla creazione e all'uso degli agenti.

In Axum, un agente rappresenta il concetto di attore e, in quanto tale, possiede tutte le caratteristiche definite nel Capitolo 3 relativamente agli attori. In aggiunta a quanto detto, Axum introduce anche altre particolarità, non previste dal modello, per agevolare la programmazione ed aumentare la flessibilità delle applicazioni stesse.

5.4.1 Domini

Negli esempi precedenti, per quanto riguarda la creazione degli agenti, si è sempre utilizzata la funzione `CreateInNewDomain()` senza tuttavia conoscerne il significato. Come suggerisce il nome, essa crea l'agente specificato in un nuovo dominio; il che porta a chiedersi cosa sia effettivamente un dominio.

Inizialmente, dando una definizione non completa, si può affermare che un dominio rappresenta un contenitore di agenti e viene dichiarato con la seguente sintassi.

```
domain MyDomain
{
  //Qui la dichiarazione degli agenti del dominio...
}
```

Ogni applicazione Axum deve contenere almeno un dominio che, se non specificato, viene creato autonomamente dal sistema.

Inoltre, ogni dominio possiede l'importante caratteristica di isolare dal resto del sistema le informazioni e i dati condivisi dagli agenti presenti all'interno del dominio stesso. Quest'ultima caratteristica porta ad implicazioni ed opportunità che saranno discusse in seguito.

5.4.2 Creazione di agenti e Hosting

Durante l'atto di creazione di un agente, oltre alla creazione vera e propria, à fondamentale realizzare il canale implementato dall'agente e contestualmente ottenere il riferimento allo *using end* del canale affinché questo possa essere utilizzato. Inoltre, è necessario anche associare l'agente creato ad uno specifico dominio di appartenenza.

Per ottenere quanto richiesto in modo semplice e veloce, è sufficiente far riferimento alla funzione, disponibile per ogni tipo di agente, `CreateInNewDomain()` che crea e inizializza l'agente, realizza un nuovo dominio nel quale collocare l'agente appena creato e restituisce il riferimento allo *using end* del canale implementato dall'agente.

Sebbene teoricamente questa funzione non ponga alcun tipo di problematica dal punto di vista esecutivo, essa risulta inutilizzabile nel caso in cui si vogliano raggruppare più agenti nello stesso dominio al fine di condividere informazioni e dati.

Per ovviare a questo problema, Axum introduce il concetto di *Hosting*. L'hosting è il meccanismo mediante il quale un particolare tipo di agente viene associato ad una specifica istanza di dominio.

Riferendoci nuovamente all'applicazione di Listato 5.3, è possibile specificare per l'agente `AdderAgent` l'appartenenza ad un dominio (quello corrente) e un indirizzo con un'istruzione del tipo

```
Host<AdderAgent>("myAdderAddress");
```

Così facendo, quando si vorrà creare ed istanziare un nuovo agente di tipo `AdderAgent`, sarà sufficiente specificarne l'indirizzo al quale esso è ospitato e ciò permetterà automaticamente di creare l'agente, aggiungerlo al dominio dal quale si effettua la creazione e restituire lo *using end* del canale associato. In particolare, si dovrà istanziare un agente con un istruzione del tipo

```
var adder = new AdderChannel("myAdderAddress");
```

nella quale viene specificato quale canale di comunicazione si vuole utilizzare in sostituzione del tipo di agente che si vuole istanziare [5].

In virtù del fatto che gli agenti di un'applicazione Axum comunicano tra loro tramite i canali, l'istruzione appena descritta dimostra efficacemente che per comunicare, ad un agente A non serva sapere nulla dell'agente B col quale vuole comunicare fatta eccezione per l'indirizzo dell'agente B e il canale da utilizzare (ovvero il canale implementato dall'agente B).

Si noti che l'indirizzo può essere composto da una qualunque stringa di caratteri e viene utilizzato mediante confronto. Si noti anche che la funzione `Host` utilizzata per associare un agente ad un dominio risulta essere una funzione di dominio e pertanto può essere richiamata solamente da un agente definito all'interno di un dominio [6].

5.4.3 Riferimento all'implementing end

Fino ad ora si è sempre parlato di riferimento allo *using end* di un canale. In effetti qualunque agente che voglia comunicare con un altro agente del sistema necessita di tale riferimento (unitamente all'indirizzo

[5] Questo risulta possibile in quanto ogni agente può implementare un unico canale.

[6] Prassi, quella di definire un agente all'interno di un dominio, che comunque dovrebbe essere sempre adottata.

dell'agente). All'interno del sistema pertanto i canali sono sempre visti tramite il relativo *using end*. L'unica eccezione a tale affermazione è visibile all'interno dei singoli agenti i quali possono riferirsi al proprio *implementing end*. Ogni agente infatti può riferirsi a gli *using end* di tutti i canali del sistema ma, per quanto riguarda gli *implementing end*, riesce a vedere unicamente il proprio, ovvero quello del canale che esso implementa.

All'interno di un agente pertanto nasce la necessità di far riferimento al proprio implementing end. Ciò avviene attraverso la keyword `PrimaryChannel` , il cui utilizzo è visibile nella seguente definizione di agente.

```
agent AdderAgent : channel AdderChannel
{
  public AdderAgent()
  {
    int add1 = receive(PrimaryChannel::Addend1);
    int add2 = receive(PrimaryChannel::Addend2);

    int result = add1 + add2;

    PrymaryChannel::Sum <-- result;
  }
}
```

5.4.4 Ereditarietà

Sebbene il modello ad attori non preveda la possibilità per un attore di ereditare da un altro attore, il team di sviluppatori di Axum ha liberamente scelto di introdurre questa caratteristica.

Pertanto, similmente a quanto accade nei linguaggi orientati agli oggetti, in Axum risulta possibile specificare che un agente erediti i metodi definiti in un altro agente.

Non è prevista eredità multipla, per cui ogni agente può ereditare al massimo da un solo altro agente. L'agente cosiddetto *figlio* può alternativamente richiamare i metodi dell'agente da cui eredita oppure sovrascriverli per assegnargli un diverso comportamento.

Per ereditare da un altro agente, è necessario specificare il nome dell'agente dal quale si vuole ereditare prima di inserire il nome del canale che si vuole implementare. Il seguente schema illustra e chiarisce il meccanismo dell'ereditarietà tra agenti in Axum.

```
agent Agent1 : channel Channel1
{
  public Agent1()
  {
    //...
  }

  protected Method1()
  {
    //...
  }
}

agent Agent2 : Agent1, channel Channel2
{
  public Agent2()
  {
    //...
  }

  override Method1()
  {
    //...
  }
}
```

5.5 Condivisione di informazioni

Introducendo il concetto di dominio, è stato annunciato che un'ulteriore caratteristica propria di tali elementi è quella di isolare rispetto al sistema le informazioni condivise dagli agenti definiti all'interno del dominio stesso.

Le specifiche del modello ad attori discusse nel Capitolo 3 non prevedono la possibilità per due o più attori di avere uno spazio comune in cui condividere informazioni.

Effettivamente si tratta di una scelta molto oculata poiché, in tal modo, si impone che qualunque condivisione di informazione tra attori debba essere effettuata sempre mediante il meccanismo di scambio asincrono di messaggi.

Nella pratica comune tuttavia, sebbene lo scambio asincrono di messaggi resti un eccellente metodo di comunicazione, il vincolo che i messaggi scambiati debbano essere sempre immutabili e serializzabili potrebbe portare a rallentamenti e conseguenti inefficienze.

Si pensi ad esempio al caso in cui due attori abbiano la necessità di condividere un dato che semplicemente funga da flag per qualcosa. Se tali attori sono e saranno sempre situati sullo stesso nodo della rete e se il dato in condivisione viene richiesto spesso durante la computazione di entrambi gli attori, si intuisce immediatamente che in un caso del genere condividere in una sorta di piccola shared memory il dato, sarebbe molto più efficiente piuttosto che utilizzare messaggi asincroni; sovraccaricando conseguentemente l'attore che dovrebbe contenere contenere l'informazione necessaria ad entrambi gli attori.

In effetti, Axum, per ampliare la possibilità di utilizzo, introduce la possibilità di condividere informazioni e dati tra i vari agenti di un'applicazione. Questo tuttavia può avvenire unicamente all'interno di un dominio e con precisi accorgimenti.

Per condividere informazioni nel contesto di un dominio è sufficiente dichiarare i dati a livello di dominio, con una sintassi del tipo

```
domain myDomain
{
  private int mySharedData1;
  private bool mySharedData2;

  //Qui la dichiarazione degli agenti del dominio...
}
```

In questo modo, nel corpo degli agenti definiti all'interno del dominio sarà possibile accedere ai dati condivisi, in modo del tutto sicuro, attraverso la keyword `parent`; un esempio di utilizzo è mostrato nel seguente frammento di codice.

```
if(parent.mySharedData2 == true)
{
  //...
}
```

5.5.1 Semantica Reader/Writer per Agenti

Seppur limitata al solo livello di dominio, la possibilità di condividere informazioni tra agenti può portare a tutte le problematiche relative alla gestione degli spazi condivisi della programmazione multithread; cosa che il modello ad attori intende assolutamente evitare. Pertanto, definire solamente quali informazioni si vogliono condividere non è sufficiente per poterle utilizzare.

In Axum, qualora si voglia fare riferimento ad un dato condiviso a livello di dominio, è necessario definire, per ogni agente, in che modo tale agente avrà l'accesso al dato. In particolare, è possibile specificare che un agente possa accedere ai dati condivisi in sola lettura mediante la keyword `reader`, viceversa se si vuole specificare che un agente possa accedere ai dati condivisi sia in lettura che in scrittura, deve essere utilizzata la keyword `writer`.

```
domain myDomain
{
  private int mySharedData1;
  private bool mySharedData2;

  reader agent Agent1 : channel Channel1
  {
    //Accesso in sola lettura ai dati condivisi
  }

  writer agent Agent2 : channel Channel2
  {
    //Accesso in lettura/scrittura ai dati condivisi
  }
}
```

Per garantire un accesso sicuro ai dati condivisi ed evitare inconsistenza, le specifiche di Axum prevedono che un attore dichiarato di tipo `writer` abbia l'accesso in mutua esclusione ai dati condivisi. Questo implica due considerazioni:

- la prima, ovvia, è che nel momento in cui un agente *writer* sta leggendo o scrivendo un dato condiviso, esso risulta essere l'unico in grado di accedere al dato. Tutti gli altri agenti dovranno attenderne il rilascio.

- la seconda, meno ovvia, è che due o più attori di uno stesso dominio dichiarati di tipo `writer` non potranno mai essere eseguiti parallelamente. Nel caso estremo in cui un applicazione preveda solo agenti di tipo *writer*, l'applicazione si ridurrebbe ad applicazione a singolo thread.

Nonostante la seconda considerazione, tuttavia, la pratica insegna che un applicazione ben progettata dovrebbe possedere la massimo un unico agente di tipo `writer` per ogni dominio e teoricamente, un numero illimitato di agenti di tipo `reader`.

Concludendo, si noti che, se un un agente non è dichiarato né di tipo `reader` né di tipo `writer`, allora questo agente non potrà accedere né il lettura né in scrittura ad eventuali dati condivisi limitandosi ad effettuare computazioni su dati immutabili e totalmente serializzabili ottenuti col meccanismo di scambio asincrono di messaggi.

5.6 Modificatori per dati locali

Quando si istanzia un tipo di dato, in Axum è possibile aggiungere un modificatore per definire alcune modalità di accesso.

In particolare, sono previste tre diverse forme per la memorizzazione locale di dati:

- **const** - accetta un unica operazione di scrittura e non permette ulteriori modifiche[7];

- **async** - dopo la prima operazione di scrittura, accetta un infinito numero di modifiche tramite altre operazioni di scrittura;

- **sync** - dopo la prima operazione di scrittura, blocca qualunque altro tentativo di sovrascrittura finché il dato non viene letto; dopo la lettura il dato viene cancellato e l'istanza torna ad essere disponibile per una nuova scrittura.

[7] È fondamentale non confondere questo modificatore con quelli comunemente utilizzati nei linguaggi OOP e non solo per la definizione di costanti. In questo caso, infatti l'operazione di scrittura per l'inizializzazione del dato non deve necessariamente essere fatta contestualmente alla creazione dell'istanza; cosa che invece accade per le costanti degli altri linguaggi.

5.7 Metodi asincroni

Sebbene nel modello ad attori siano teoricamente assenti tutte le problematiche relative ai fenomeni potenzialmente bloccanti tipici della programmazione multithread, le varie implementazioni non sempre implementano il modello senza alterarlo e ciò comporta la necessità di inserire particolari accorgimenti affinché siano comunque rispettate le proprietà semantiche.

In Axum, diverse sono le possibili fonti di blocchi, la maggior parte dovuta alla necessità di interagire con l'ambiente .NET Framework.

In particolare, sono fonte di blocchi:

- Uso diretto o indiretto di istruzioni di I/O sincrone, come ad esempio `Console.ReadLine()`;
- Uso diretto o indiretto di istruzioni .NET per la gestione della sincronizzazione, come ad esempio l'istruzione `Monitor.Enter()` per utilizzare un monitor;
- Espressioni di tipo `receive`;
- ... e tante altre.

Nonostante i blocchi rappresentino una delle maggiori fonti di irritazione per progettisti, sviluppatori e utilizzatori, essi sono e resteranno una realtà.

Fortunatamente per la maggior parte delle operazioni potenzialmente bloccanti, in Axum esiste la possibilità di trasformarle in operazioni non bloccanti.

Quando si intende utilizzare funzionalità che possano essere causa di operazioni bloccanti, è sufficiente incapsularle all'interno di un metodo etichettato come `asynchronous`.

In questo modo, è possibile concentrarsi unicamente sulla logica dell'applicazione senza preoccuparsi dei meccanismi per rendere asincrone tali operazioni; sarà il compilatore Axum a prendersi carico di ciò.

Infatti, in fase di compilazione, tutte le istruzioni contenute in un metodo asincrono che non siano già asincrone, vengono trasformate secondo il modello .NET APM (*Asynchronous Programming Model*).

Per maggiore chiarezza, si consideri il seguente frammento di codice in cui è abbozzato il metodo `ReadFile()` ipotizzando che esso possa essere utilizzato per conoscere il numero di caratteri contenuti nel file.

```
private asynchronous void ReadFile(string path)
{
  Stream stream = new Stream(...);

  int numRead = stream.Read(...):

  while(numRead > 0)
  {
    numRead = stream.Read();
  }
}
```

Nell'esempio si fa uso della funzione `Read()` definita per l'elemento `Stream` della libreria `System.IO`. Tale funzione può portare a situazioni potenzialmente bloccanti e per questo motivo è assolutamente necessario incapsularla, e quindi utilizzarla, all'interno di un metodo asincrono. In questo modo, il compilatore Axum la sostituirà con la rispettiva funzione APM e la potenziale situazione di blocco sarà evitata.

5.7.1 Performance e regole d'uso

Usare metodi asincroni permette di ridurre significativamente il costo dovuto al meccanismo di scambio asincrono di messaggi.

Inoltre, in questo modo risulta possibile effettuare tutte le operazioni di I/O senza preoccuparsi di ottenere situazioni di blocco. Naturalmente, l'uso spropositato di tali metodi produce l'effetto contrario: definire un metodo asincrono che non contenga funzioni bloccanti risulta inutilmente costoso in termini di performance.

Di norma, in Axum, tutti i metodi sono di tipo sincrono per default; qualora si voglia rendere un metodo asincrono è necessario dichiararlo esplicitamente. Gli unici "metodi" che sono asincroni per default sono i costruttori degli agenti.

Per evitare di fare un uso sovrabbondante di metodi asincroni, è bene far riferimento a tre semplici regole. Un metodo va dichiarato asincrono se e solo se:

1. contiene almeno una chiamata alla primitiva `receive`;
2. contiene almeno una chiamata di una funzione che possiede una corrispondente variante nell'APM;
3. effettua una chiamata ad un altro metodo asincrono.

In relazione alla "regola n.2" appena citata, va specificato che non tutte le funzioni richiamabili in Axum hanno una variante nell'APM. È il caso, ad esempio, della già citata funzione `Monitor.Enter()`. In questi casi, sarebbe opportuno evitare l'uso di tali funzioni e pensare ad una metodologia alternativa attraverso l'uso del pattern Empty/Full realizzato mediante l'uso dei modificatori sui dati locali.

5.8 DataFlow Network

Nelle sezioni precedenti, introducendo e descrivendo il linguaggio Axum, si è sempre fatto riferimento al modello di comunicazione basato sul meccanismo di scambio asincrono di messaggi tra agenti. Tuttavia, tale modello (chiamato in Axum *ControlFlow Communication*) non è l'unico approccio alla comunicazione utilizzabile in Axum.

Nel modello ControlFlow, dove gli agenti si scambiano messaggi asincroni, la comunicazione è gestita tramite la logica dell'applicazione, lo stato dei vari agenti o in base ai messaggi ricevuti.

A tale approccio, si contrappone il modello denominato *DataFlow Communication*. Nel DataFlow, l'esecuzione dell'applicazione è gestita unicamente in base alla disponibilità di dati all'ingresso della cosiddetta DataFlow Network e la computazione è realizzata in base a come i dati si muovono sulla relativa Network.

Generalmente, i due modelli di comunicazione e gestione della logica di applicazione vengono utilizzati in modo combinato per aumentare e migliorare le performance.

5.8.1 Nodi di interazione

Nelle DataFlow Network di Axum, i dati sono inviati e ricevuti attraverso i cosiddetti *nodi di interazione* (*interaction point*) chiamati alternativamente *source*, quando il nodo identifica un punto in cui viene generato e spedito un messaggio , oppure *target*, quando il nodo identifica un punto al quale viene destinato e ricevuto un messaggio. Ovviamente, un nodo può essere contemporaneamente sia *source* che *target*.

I nodi di interazione sono alla base della composizione delle DataFlow Network in quanto consentono la composizione dell'interazione tra i vari agenti del sistema.

Ogni messaggio, fluttuando attraverso la rete e passando da un nodo all'altro, subisce una trasformazione secondo quanto specificato per raggiungere poi la destinazione e produrre così il risultato.

In una DataFlow Network, i nodi possono essere rappresentati sia mediante metodi (o funzioni) definiti ad hoc per descrivere una particolare computazione, sia attraverso istanze di componenti predefiniti. In Axum, esistono diversi tipi predefiniti per rappresentare un nodo di interazione, cinque tra i quali risultano essere quelli maggiormente utilizzati:

- `OrderedInteractionPoint` - costituisce un semplice buffer per dati che preserva l'ordine di arrivo dei messaggi; può rappresentare sia un nodo source sia un nodo target.
- `ImmediateValue` - rappresenta un nodo che fornisce sempre un predefinito valore costante.
- `WriteOnceInteractionPoint` - inizialmente vuoto, rappresenta un nodo che accetta un'unica scrittura di dato dopo la quale il valore scritto non può più essere modificato; è utilizzabile sia come source sia come target.
- `Future` - costituisce una variante del WriteOnce, fornendo un valore che non può essere sovrascritto ottenuto mediante una specifica computazione anziché attraverso un altro nodo.
- `SingleItemInteractionPoint` - costituisce un nodo di interazione che accetta un unico valore per volta; qualora sia presente un valore non ancora trasmesso e un altro valore sia in coda al nodo, quest'ultimo valore viene rifiutato finché il valore attualmente presente sul nodo non viene trasmesso.

ESEMPIO

Si presenta ora un primo esempio di applicazione realizzata mediante il meccanismo di DataFlow Network. L'applicazione proposta permette di calcolare gli elementi della sequenza di Fibonacci. In particolare, essa restituisce sulla shell a caratteri la sequenza di Fibonacci dal 42esimo elemento fino al primo (zero escluso).

Listato 5.5: *Prima applicazione con DataFlow Network*

```
using System;
using Microsf.Axum;
using System.Concurency.Messagging;

agent MainAgent : channel Microsoft.Axum.Application
{
```

```
  int numElements = 42;
  int counter = numElements*2;

  public MainAgent()
  {
    var numbers = new OrderedInteractionPoint<int>();

    numbers ==> Fibonacci ==> ProcessResult;

    for(int i=0;i<counter;i++)
      numbers <-- numElements-i;
  }

  function int Fibonacci(int n)
  {
    if(n <= 1) return n;
    return Fibonacci(n-1) + Fibonacci(n-2);
  }

  void ProcessResult(int n)
  {
    Console.WriteLine(n);

    if(--counter == 0)
      PrimaryChannel::ExitCode <-- 0;
  }
}
```

Analizzando l'applicazione proposta dal Listato 5.5, tralasciando i metodi `Fibonacci()` e `ProcessResult()` che rappresentano rispettivamente l'algoritmo (descritto in termini ricorsivi) per il calcolo e la metodologia di visualizzazione dei risultati, si ponga l'attenzione sul costruttore dell'agente `MainAgent`.

La prima istruzione, permette la creazione di un nodo di interazione sulla rete.

```
var numbers = new OrderedInteractionPoint<int>();
```

Viene creata un'istanza dell'elemento `OrderedInteractionPoint` che, in particolare, permette di ottenere un nodo, sia source che target, che preserva l'ordine di arrivo dei messaggi. Tale nodo risulta necessario poiché rappresenterà lo *start point* della logica dell'applicazione.

Dopo la creazione del nodo, viene descritta la DataFlow Network secondo il metodo di pipeline, che ne costituisce la forma più semplice.

```
numbers ==> Fibonacci ==> ProcessResult;
```

Attraverso l'operatore di forwarding (descritto successivamente), infatti, viene specificato che tutti i messaggi in arrivo sul nodo `numbers` dovranno essere inoltrati sul nodo `Fibonacci` (costituito dalla funzione opportunamente definita), il quale a sua volta, dopo aver completato l'elaborazione, inoltrerà il risultato al nodo successivo di `ProcessResult`.

Una volta creato il nodo di partenza e descritta la Network, è possibile attivare la computazione.

```
for(int i=0;i<counter;i++)
  numbers <-- numElements-i;
```

Si noti come sia sufficiente inviare al nodo *start point* solamente i messaggi contenenti ciascuno l'indice dell'elemento della sequenza di Fibonacci da calcolare. Tutta la successiva logica di gestione di tali dati è effettuata tramite la DataFlow Network.

Concludendo, si presti attenzione alle seguenti precisazioni:

1. Il metodo `Fibonacci()` è stato definito mediante la keyword `function`. Questo implica che le istruzioni poste all'interno del metodo non possano modificare nessun dato al di fuori di quelli definiti all'interno del metodo; ciò garantisce quindi l'assenza di qualunque genere di possibile *side-effect* durante l'esecuzione. L'aver definito il metodo come `function`, con la conseguente garanzia di non avere side-effect, consente ad Axum di eseguire parallelamente a runtime più chiamate al metodo, permettendo cioè di ridurre notevolmente i tempi di esecuzione e quindi di aumentare l'efficienza dell'applicazione.

2. Definendo la Network mediante una struttura a pipeline, la quale garantisce che in ogni nodo della rete i messaggi arrivino e partano ordinati, risulta possibile eseguire tutte le chiamate ai vari nodi della rete in modo completamente parallelo.

5.8.2 Operatori

Come si evince dall'esempio di Listato 5.5, la Network è stata definita facendo uso di specifici operatori (nel caso dell'esempio si tratta dell'ope-

ratore di forward). Un tale operatore è caratterizzato da una definizione binaria, che richiede un'espressione sorgente come operando sinistro ed un'espressione destinazione come operando destro.

Gli operatori per le Network di Axum possono essere suddivisi in tre categorie: One to One, Many to One, One to Many.

Operatori "One to One"

Fanno parte della categoria One to One gli operatori *Forward* e *Forward Once*.

- **Forward** (`==>`) : Inoltra ogni messaggio prodotto dal operando sorgente al operando destinazione.
- **Forward Once** (`-->`) : Inoltra il primo messaggio prodotto dal operando sorgente al operando destinazione, quindi disconnette la Network in quel punto.

Come si è già visto, l'operatore di forward viene utilizzato per creare pipeline; inoltre, può essere usato anche per gestire architetture event-driven, come ad esempio le GUI. Il seguente frammento di codice illustra un possibile utilizzo.

```
channel GUIChannel
{
  input MouseEvent Click;
  input Key Keypress;
  input Canvas Paint;
}

agent GUIHandler : channel GUIChannel
{
  public GUIHandler()
  {
    PrimaryChannel::Click ==> ClickHandler;
    PrimaryChannel::KeyPress ==> KeypressHandler;
    PrimaryChannel::Paint ==> PaintHandler;
  }

  void ClickHandler(MouseEvent event)
  {
    //...
  }
```

```
  void KeypressHandler(Key key)
  {
    //...
  }

  void PaintHandler(Canvas cvs)
  {
    //...
  }
}
```

Il codice illustra come sia possibile, attraverso l'operatore di forward, associare un ben preciso comportamento a fronte di un messaggio ricevuto su una specifica porta. In particolare, le porte del canale implementato rappresentano il verificarsi di eventi sulla GUI. Quando si verifica un evento (ad esempio la pressione di un tasto sulla tastiera) viene inviato un messaggio sulla relativa porta (`Keypress` in questo caso) che attraverso la rete specifica viene inoltrato al metodo corrispondente che computerà in base a ciò che è specificato all'interno.

Operatori "Many to One"

Fanno parte della categoria Many to One gli operatori *Multiplex* e *Combine*.

- **Multiplex** (`>>-`) : Invia i messaggi provenienti da un vettore di nodi sorgente ad un singolo nodo destinazione.
- **Combine** (`&>-`) : Unisce tutti i messaggi provenienti da più nodi sorgente e li invia mediante un vettore al nodo destinazione.

Operatori "One to Many"

Fanno parte della categoria One to Many gli operatori *Forward* e *Forward Once*.

- **Broadcast** (`-<<`) : Inoltra il messaggio proveniente dal nodo sorgente al vettore specificante tutti i nodi destinazione.
- **Alternate** (`-<:`) : Inoltra il messaggio proveniente dal nodo sorgente al vettore di nodi destinazione utilizzando un ordine di tipo round-robin.

ESEMPIO

Affinché sia possibile avere una visione più chiara relativamente all'uso degli operatori nelle DataFlow Network, si propone ora una reingegnerizzazione dell'applicazione di Listato 5.3.

Listato 5.6: *Esempio d'uso degli operatori per le DataFlow Network*

```
using System;
using System.Concurrency;
using Microsoft.Axum;

channel AdderChannel
{
  input int Addend1;
  input int Addend2;
  output int Sum;
}

agent AdderAgent : channel AdderChannel
{
  public AdderAgent()
  {
    var ipJoin = new OrderedInteractionPoint<int>();

    {PrimaryChannel::Addend1 ==> getNumber,
      PrimaryChannel::Addend2 ==> getNumber}
    &>- ipJoin -<: {GetSum, GetSum} >>- PrimaryChannel::Sum;
  }

  private int getNumber(int n)
  {
    return n;
  }

  private function int getSum(int[] nums)
  {
    return nums[0] + nums[1];
  }
}

agent MainAgent : channel Microsoft.Axum.Application
{
  public MainAgent()
```

```
  {
    String[] args = receive(PrimaryChannel::CommandLine);

    var adder = AdderAgent.CreateInNewDomain();

    adder::Addend1 <-- Convert.ToInt32(args[0]);
    adder::Addend2 <-- Convert.ToInt32(args[1]);

    int sum = receive(adder::Sum);

    Console.WriteLine("{0} + {1} = {2}", args[0], args[1], sum);

    PrimaryChannel::ExitCode <-- 0;
  }
}
```

Il primo nodo della rete è composto da un vettore di due elementi, ciascuno dei quali inoltra il proprio valore al metodo `getNumber()`. Quindi, i valori vengono combinati nel nodo `ipJoin` e trasferiti poi ad un nuovo array formato da una tupla di metodi `GetSum()` in grado di elaborarli. Infine, il valore finale viene inviato alla porta `Sum`.

5.9 Axum e il .NET Framework

Axum nasce con l'idea di ampliare le potenzialità offerte attraverso la piattaforma .NET e, per tanto, trae profonda ispirazione da quanto già esiste in ambito .NET Framework. In particolar modo, Axum eredita molte caratteristiche, soprattutto per quanto riguarda la sintassi e gli elementi base, dal linguaggio principale di .NET, ovvero C#. Questo rende Axum un linguaggio molto evoluto poiché coniuga le caratteristiche importanti e consolidate di un linguaggio ad ampia diffusione come C# e le innovative funzionalità orientate alla concorrenza. Tra le altre, Axum eredita da C# le seguenti caratteristiche:

- Tutte le espressioni presenti in C# 3.5, comprese le Lambda Expression e la libreria LINQ per le query sui dati;
- Tutte i costrutti presenti in C# 3.5, compresa l'opzione yield per gli enumeratori;
- La sintassi di dichiarazione di metodi, campi e funzioni;

- I delegates;

Viceversa, non sono presenti tutti quegli aspetti propri dell'object oriented come classi, interfacce, strutture, campi statici e proprietà, che sono invece stati sostituiti dai componenti che implementano il modello ad attori.

5.9.1 Lo stato dell'arte

Attualmente (anno 2012) Axum risulta essere un progetto dei Microsft DevLabs dichiarato concluso. Sulla pagina web ufficiale del progetto[8] è disponibile per il download un installer contenente le librerie per Axum e il relativo compilatore. Esiste, ad oggi, una versione compatibile con l'IDE Microsoft Visual Studio 2008 e un'altra compatibile con l'IDE Microsoft Visual Studio 2010. Inoltre risultano disponibili una guida dettagliata [8], da cui sono tratti gli esempi del presente capitolo, e un documento con le specifiche del linguaggio [9].

5.9.2 Integrazione con .NET Framework 4.5

Attualmente in versione beta, la nuova release della piattaforma .NET Framework introdurrà, tra le altre innovazioni, anche notevoli features per quanto riguarda l'aspetto di Parallel Computing. Le nuove caratteristiche saranno orientate al miglioramento delle performance e ad un considerevole nuovo supporto alla programmazione asincrona.

In particolare, è stato annunciato [10] che saranno integrate in .NET Framework 4.5 alcune delle caratteristiche introdotte in Axum, tra le quali, quelle di maggior risalto risultano essere i metodi asincroni e le dataflow network.

Tuttavia, al momento, non si hanno certezze sul come tali caratteristiche saranno introdotte, se cioè sarà mantenuto l'approccio utilizzato in Axum oppure se, come sembra più probabile, tali caratteristiche saranno si introdotte ma rivisitate per farle aderire al paradigma ad oggetti che resta essere ancora il cuore della programmazione commerciale.

[8] `http://www.microsoft.com/en-us/download/details.aspx?id=21024`

Capitolo 6

Analisi critica su AXUM

Nonostante Microsoft abbia attualmente dichiarato concluso il progetto relativo ad Axum (si veda il paragrafo 5.9.1), non è da escludere che in futuro il linguaggio Axum possa essere riconsiderato in relazione alla scelta di un successore all'attuale linguaggio object oriented C# in relazione alle sempre maggiori richieste in campo di *parallel programming*.

Naturalmente, una candidatura di Axum in tal senso, può essere presa in considerazione unicamente se il linguaggio potrà costituire una valida alternativa alle altre implementazioni del modello ad attori e se potrà quindi delineare in qualche modo il futuro della programmazione concorrente.

Attualmente, vista la vastità di linguaggi disponibili specificatamente ideati e progettati per la programmazione concorrente, risulta difficile stabilire un possibile futuro per Axum.

In ogni caso, comunque, è opinione diffusa che completamente o in parte, il linguaggio Axum e i meccanismi da esso introdotti saranno sicuramente integrati in tutte le tecnologie Microsoft che abbiano interesse a gestire aspetti concorrenti.

Dal punto di vista commerciale, Axum potrà avere successo, e quindi riscontrare un crescente impiego per la realizzazione di applicazioni, unicamente se i gradi di flessibilità e di semplicità offerti siano tali da garantire alte performance nelle applicazioni e bassi costi di produzione. Inoltre, l'integrazione tra Axum e C# potrebbe giocare un ruolo fondamentale qualora si voglia utilizzare Axum per riscrivere parti di applicazioni esistenti per migliorarne certi aspetti pur mantenendo un impostazione generale object oriented.

Dal punto di vista didattico/accademico, invece, Axum potrà essere considerato come linguaggio esempio per la programmazione concorrente

tanto più quanto l'implementazione del modello ad attori proposta da Axum sia fedele alle specifiche teoriche del modello stesso.

Perciò, analizzare in modo critico alcuni aspetti di Axum, risulta fondamentale per tentare di rispondere in parte alla domanda relativa al futuro di Axum.

6.1 Axum e il Modello ad Attori

Spesso accade che la definizione di un modello teorico non consideri aspetti pratici che tuttavia devono essere considerati qualora si voglia implementare il modello in un linguaggio di programmazione. Inoltre, spesso accade anche che ogni linguaggio che implementa un preciso modello decida liberamente di non seguire esattamente quanto previsto dal modello introducendo così alcune varianti.

Il grado di divergenza di un linguaggio di programmazione rispetto al modello teorico costituisce un indice per la valutazione del linguaggio; indice che teoricamente dovrebbe essere pari a zero. Tuttavia, per quanto si cerchi di rispettare le specifiche imposte dal modello, inevitabilmente ogni implementazione introduce elementi che portano a differenze, talvolta notevoli.

Il linguaggio Axum non fa eccezione e diversi sono gli aspetti che sono stati liberamente modificati rispetto a quanto previsto nel modello ad attori. Nei paragrafi seguenti saranno analizzati tali aspetti.

6.1.1 Agenti e mailbox

Il primissimo aspetto che emerge è rappresentato dal nome scelto per identificare gli attori.

In Axum, il concetto di attore è rappresentato da un'entità chiamata `agent`. Per quanto nelle guide ufficiali del linguaggio [8] venga specificato che un `agent` rappresenta un attore, questo non è del tutto vero o, perlomeno, non è completamente corretto. Il modello ad attori originale identifica un attore come un'entità che:

1. possiede un proprio flusso d'esecuzione;
2. è costituita da un proprio stato interno non accessibile dall'esterno;
3. scambia messaggi con gli altri attori in modo asincrono;
4. possiede una mailbox mediante la quale ricevere messaggi;

5. a fronte della ricezione di un messaggio può rispondere con un altro messaggio, oppure modificando il proprio stato interno, oppure infine creando un nuovo attore.

Un `agent` concorda quasi in tutto con la definizione di attore introdotta dal modello ad eccezione del fatto che non possiede una mailbox esplicita mediante la quale ricevere i messaggi.

In Axum i messaggi destinati ad un attore raggiungono le porte poste all'*implementing end* del canale implementato dall'attore e quindi vengono letti mediante la chiamata alla primitiva `receive`. Non è tuttavia prevista una struttura unica alla quale far riferimento per leggere i messaggi senza necessariamente conoscere a priori il "tipo" del messaggio.

Sull'*implementing end* del canale, infatti, possono coesistere più porte, ciascuna delle quali destinata a ricevere un preciso tipo di messaggio. Per leggere un messaggio è quindi necessario far riferimento ad una precisa porta e quindi non risulta corretto affermare che *gli* `agent` *restano in attesa di messaggi*, bensì è più corretto affermare che *gli* `agent` *restano in attesa di specifici messaggi*.

La scelta di non introdurre una mailbox esplicita riduce sicuramente la complessità nella gestione dello scambio di messaggi, in quanto i controlli sulla correttezza dei messaggi ricevuti si semplificano molto. Tuttavia, in questo modo, viene ridimensionata l'idea di generalità che il modello associa agli attori. Infatti, se nel modello ad attori, un attore può ricevere qualunque messaggio, in Axum un `agent` può ricevere solo i messaggi previsti dalle porte del proprio canale.

6.1.2 Canali e messaggi

L'aspetto relativo all'assenza di una mailbox esplicita può essere aggirato imponendo che su ogni porta dell'*implementing end* del canale di un `agent` debbano essere depositati i vari dati del messaggio e che quindi l'insieme di tali porte costituisca l'accesso alla "mailbox", la quale comunque resterebbe vincolata a ricevere un solo tipo di messaggio.

Tuttavia, anche questa soluzione non risulterebbe appropriata poiché in questo modo verrebbe meno l'idea di messaggio atomico prevista dal modello ad attori. In Axum, il fatto che un messaggio debba essere atomico (anche qualora i dati del messaggio vengano trasmessi in tempi diversi) è garantito dalla possibilità di specificare protocolli sui canali oppure dalla possibilità di utilizzare i dati `schema`.

In questo modo però, torna ad essere evidente la discrepanza esistente tra l'implementazione degli attori in Axum e il modello teorico originale.

In definitiva, in Axum, i messaggi potrebbero anche non essere atomici qualora non si presti particolare attenzione affinché questo sia garantito; compito che peraltro è demandato completamente al programmatore.

Inoltre, un altro aspetto presente in Axum ma non previsto dal modello ad attori è il concetto di canale. Ogni `agent` può implementare un unico e ben preciso `channel` mediante il quale può ricevere messaggi ed inviare le risposte.

Nel modello ad attori, non è presente nessuna definizione circa eventuali canali di comunicazione tra attori. In effetti, secondo il modello, ad un attore risulta sufficiente conoscere l'indirizzo dell'attore al quale desidera inviare un messaggio; tale indirizzo costituisce l'unico vincolo necessario al fine di instaurare una comunicazione tra attori.

In Axum, invece, l'indirizzo di un `agent` non permette di raggiungere direttamente l'agente, bensì permette di ottenere il riferimento al canale da esso implementato.

Questo approccio isola maggiormente gli `agent` tra loro ma, in definitiva, non essendo (giustamente) prevista la possibilità di cambiare dinamicamente l'implementazione del canale, il fatto di utilizzare un intermediario per la comunicazione (il `channel` appunto), potrebbe essere considerato superfluo.

6.1.3 Primitiva `receive` esplicita

Un'ulteriore aspetto non previsto dal modello ad attori ma presente in Axum è rappresentato dalla presenza di una primitiva esplicita per la ricezione dei messaggi.

Nel modello ad attori, una volta che un messaggio ha raggiunto la mailbox di un attore si ha la certezza che sarà letto senza necessariamente prevedere una primitiva (bloccante) per attendere e processare i messaggi ricevuti all'interno del behaviour dell'attore stesso.

Nel modello, infatti, è un loop implicito che si occupa della gestione della mailbox di un attore, associando ad uno specifico *method handler* ogni messaggio ricevuto in base alla richiesta in esso presente.

In Axum è invece presente una primitiva `receive` esplicita mediante la quale attendere e processare i messaggi ricevuti.

Sebbene ciò non aderisca alle specifiche del modello, mediante una primitiva esplicita e bloccante è relativamente più semplice gestire la sincronizzazione e la coordinazione tra `agent`.

6.1.4 L'approccio mediante DataFlow Network

Se la presenza della primitiva `receive` esplicita discosti notevolmente il linguaggio Axum dal modello ad attori, la possibilità offerta da Axum di scrivere applicazioni secondo l'approccio delle DataFlow Network consente di aderire decisamente molto meglio a quanto previsto dal modello.

Attraverso gli operatori per le DataFlow Network, infatti, è possibile evitare di ricorrere alla primitiva `receive`, inoltrando direttamente i messaggi ricevuti a specifici handler. L'esempio seguente descrive dettagliatamente questo meccanismo.

```
schema MsgA {
  //...
}

schema MsgB
{
  //...
}

channel MessageChannel
{
  input MsgA MessageA;
  input MsgB MessageB;
}

agent GenericAgent : channel MessageChannel
{
  public GenericAgent()
  {
    PrimaryChannel::MessageA ==> HandlerForMessageA;
    PrimaryChannel::MessageB ==> HandlerForMessageB;
  }

  void HandlerForMessageA()
  {
    //...
  }

  void HandlerForMessageB()
  {
    //...
```

```
  }
}
```

Dall'esempio proposto si nota come il canale definisca i tipi di messaggi accettati, definiti a loro volta mediante tipi `schema`, mentre la ricezione di tali messaggi da parte dell'agente sia effettuata senza utilizzare la primitiva `receive`. Nel costruttore dell'agente, si specifica infatti a quale handler deve essere inoltrato ogni messaggio in base alla porta sulla quale questo giunge. Sarà poi l'handler a gestire il messaggio e incapsulando in questo modo l'idea di behaviour relativo alla ricezione di un messaggio.

Mediante l'approccio alle DataFlow Network, è possibile scrive codice decisamente pulito e chiaro avvicinandosi molto a quanto previsto dal modello originale. Tale approccio rappresenta probabilmente l'aspetto più innovativo di Axum, nonché una delle idee che maggiormente concilia la logica dello scambio di messaggi del modello ad attori al modo in cui tale logica viene inserita nelle varie implementazioni.

6.2 Flessibilità nello scrivere applicazioni

Sebbene non tutte le caratteristiche del linguaggio Axum siano completamente aderenti al modello teorico ad attori, le diverse opportunità offerte consentono maggiore flessibilità nella scrittura di applicazioni.

Una stessa applicazione, infatti, può essere strutturata diversamente in base a quale "tecnica" di programmazione si intende utilizzare. In particolare, può essere prediletto l'approccio attraverso l'uso dei canali nei quali concentrare un maggiore controllo di flusso, oppure può essere scelto l'approccio basato sulle DataFlow network in cui sono i dati a indirizzare il flusso d'esecuzione. Inoltre, alternativamente è disponibile un approccio mediante domini e elementi condivisi sfruttando la semantica reader/writer per gli agenti.

A dimostrazione di quanto detto sarà ora presentata un'applicazione sviluppata secondo diverse "tecniche"[1]. In particolare sarà considerata la nota applicazione *PingPong* proposta da Erlang[2] nella quale sono definiti due attori (rispettivamente *Ping* e *Pong*) che si scambiano messaggi in modo ordinato.

[1] L'applicazione proposta, nelle sue varianti, è tratta dagli articoli [21], [22], [23] di Matthew Podwysocki

[2] Concurrent Programming, 3.2 Message Passing (`http://erlang.org/doc/getting_started/conc_prog.html`)

6.2.1 Implementazione orientata ai canali

Costituisce il metodo più semplice ma meno performante per realizzare applicazioni. La logica di comunicazione è interamente gestita dai canali mediante le specifiche dei protocolli. La ricezione dei dati avviene sempre mediante `receive` esplicite e nella maggioranza dei casi risulta fondamentale prevedere nei canali porte per trasmettere l'informazione relativa alla terminazione di un preciso agente.

Listato 6.1: *Applicazione PingPong - Approccio channel-based*

```
using System;
using System.Concurrency;
using System.Concurrency.Messagging;
using System.Collection.Generic;
using Microsoft.Axum;

public channel PingPongStatus
{
  input int HowMany;
  output int Done;

  Start : {HowMany, Done -> End;}
}

public channel PingPong
{
  input int HowMany;
  output int Done;
  input Signal Ping;
  output Signal Pong;
}

public agent Program : channel Application
{
  public Program()
  {
    var args = receive(PrimaryChannel::CommandLine);
    var iters = 10;
    var pingAg = Ping.CreateInNewDomain();
    pingAg::HowMany <-- iters;
    receive(pingAg::Done);
    PrimaryChannel::Done <-- Signal.Value;
  }
```

```
}

public agent Ping : channel PingPongStatus
{
  public Ping
  {
    var iters = receive(PrimaryChannel::HowMany);
    var PongAgent = Pong.CreateInNewDomain();
    pongAg::HowMany <-- iters;

    for(int i=0; i<iters; i++)
    {
      pongAg::Ping <-- Signal.Value;
      receive(pongAg::Pong);
      Console.WriteLine("Ping received Pong");
    }

    int pongIters = receive(pongAg::Done);
    PrymaryChannel::Done <-- 0;
  }
}

public agent Pong : channel PingPong
{
  public Pong()
  {
    var iters = receive(PrimaryChannel::HowMany);

    int i = 0;
    for(; i<iters; i++)
    {
      receive(PrimaryChannel::Ping);
      Console.WriteLine("Pong received Ping");
      PrimaryChannel::Pong <-- Signal.Value;
    }

    PrimaryChannel::Done <-- i;
  }
}
```

Descrivendo brevemente l'applicazione, si noti che l'agente `Program` crea l'agente `Ping` e invia sulla sua porta `Done` il numero di iterazioni desiderate, quindi resta in attesa della segnalazione circa la terminazio-

ne. L'agente `Ping`, ricevuta l'informazione sul numero di iterazioni, crea l'agente `Pong` e avvia lo scambio di messaggi con esso. L'agente `Pong`, ricevuto dall'agente `Ping` il numero di iterazioni, esegue la propria computazione scambiando messaggi che risulteranno sincronizzati e intervallati. Per ogni interazione sarà stampato a video il seguente messaggio

```
>Pong received Ping
>Ping received Pong
```

in cui la prima riga è inviata dall'agente `Pong` mentre la seconda è inviata dall'agente `Ping`.

6.2.2 Implementazione orientata al DataFlow

La logica dell'applicazione è gestita dalla network descritta all'interno di ciascun `agent`, la quale definisce come i dati debbano essere scambiati. Le performance aumentano in quanto diminuisce il numero di `receive` esplicite necessarie. La computazione all'interno di ogni agente è demandata, generalmente, a specifiche funzioni le quali rappresentano nodi della rete. In questo modo, il grado di manutenibilità e riusabilità dell'applicazione vede un notevole aumento.

Listato 6.2: *Applicazione PingPong - Approccio mediante DataFlow*

```
using System;
using System.Concurrency;
using System.Concurrency.Messaging;
using Microsoft.Axum;

public channel PingPongStatus
{
    input int HowMany : Signal;
    output Signal Done;

    Start: { HowMany -> End; }
}

public channel PingPong
{
    input Signal Done;
    input Signal Ping;
    output Signal Pong;
}
```

```
public agent Program : channel Application
{
    public Program()
    {
        var pingAg = Ping.CreateInNewDomain();
        pingAg::HowMany <-- 10;
        pingAg::Done ==> Done;
    }
}

public agent Ping : channel PingPongStatus
{
    public Ping()
    {
        PrimaryChannel::HowMany ==> Process;
    }

    private Signal Process(int iters)
    {
        var pongAg = Pong.CreateInNewDomain();

        for (int i = 0; i < iters; i++)
        {
            pongAg::Ping <-- Signal.Value;
            receive(pongAg::Pong);
            Console.WriteLine("ping received pong");
        }

        pongAg::Done <-- Signal.Value;
        return Signal.Value;
    }
}

public agent Pong : channel PingPong
{
    public Pong()
    {
        while (true) receive
        {
          from PrimaryChannel::Done :
            return;

          from PrimaryChannel::Ping :
            Console.WriteLine("pong received ping");
```

```
            PrimaryChannel::Pong <-- Signal.Value;
            break;
        }
    }
}
```

Rispetto all'implementazione orientata ai canali, qui non è necessario trasmettere all'agente `Pong` il numero di iterazioni che si intendono eseguire. Quest'ultimo agente, infatti, gestisce lo scambio di messaggi mediante attraverso una particolare definizione delle primitiva `receive` che permette di specificare su quali porte attendere contemporaneamente dei messaggi.

6.2.3 Implementazione orientata ai domini

Sfruttando la semantica reader/writer per gli agenti e introducendo specifici domini mediante i quali condividere nodi di interazione, questa implementazione estende quella orientata alle DataFlow Network.

Nonostante questa implementazione sia molto performante, essa necessita spesso della definizione di regioni di codice `unsafe` in quanto il compilatore Axum, in presenza di operazioni che potenzialmente potrebbero modificare lo stato interno degli agenti, non distingue tra stato immutabile e non.

Tuttavia, sebbene l'uso spropositato di codice `unsafe` sia decisamente sconsigliabile, un approccio in cui solo piccolissime porzioni di codice siano `unsafe` potrebbe garantire massime performance, evitando comunque bug e malfunzionamenti.

Listato 6.3: *Applicazione PingPong - Approccio domain-based*

```
using System;
using System.Concurrency;
using Microsoft.Axum;
using System.Concurrency.Messaging;

public channel PingPongStatus
{
  input int HowMany : Signal;
    output Signal Done;

    Start: { HowMany -> End; }
}
```

```
public channel PingPong
{
    input Signal Done;
    input Signal Ping;
    output Signal Pong;
}

public agent Program : channel Application
{
    public Program()
    {
        var pingAg = PingPongDomain.Ping.CreateInNewDomain();
        pingAg::HowMany <-- 10;
        pingAg::Done ==> Done;
    }
}

public domain PingPongDomain
{
    public PingPongDomain()
    {
        Host<Pong>("Pong");
    }

    OrderedInteractionPoint<Signal> ping =
              new OrderedInteractionPoint<Signal>();
    OrderedInteractionPoint<Signal> pong =
              new OrderedInteractionPoint<Signal>();

    public agent Ping : channel PingPongStatus
    {
        public Ping()
        {
            PrimaryChannel::HowMany ==> Process;
        }

        private Signal Process(int iters)
        {
            var pongAg = DefaultCommunicationProvider
                                .Connect<PingPong>("Pong");

            unsafe
            {
```

```
            for (int i = 0; i < iters; i++)
            {
                ping <-- Signal.Value;
                receive(pong);
                Console.WriteLine("ping received pong");
            }
        }

        pongAg::Done <-- Signal.Value;
        return Signal.Value;
    }
  }

  public agent Pong : channel PingPong
  {
    public Pong()
    {
      unsafe
      {
        while (true) receive
        {
          from PrimaryChannel::Done :
            return;

          from ping :
            Console.WriteLine("pong received ping");
            pong <-- Signal.Value;
            break;
        }
      }
    }
  }
}
```

Diversamente da Erlang (e altre implementazioni del modello ad attori), Axum consente la condivisione di informazioni tramite i domini.

Tuttavia, questo potrebbe introdurre errori nella computazione a causa del non completo isolamento delle informazioni.

In ogni caso, qualora si decida di utilizzare tale approccio, è bene ricordare che mediante l'uso di agenti di tipo reader/writer è possibile garantire la sicurezza dell'applicazione, eccetto per alcune porzioni di codice per le quali bisogna ricorrere all'uso di codice `unsafe` che deve

pertanto comprendere il minor numero di righe di codice possibile al fine di avere un maggior controllo su di esso.

6.3 Integrazione tra Axum e C#

Il linguaggio Axum è stato ideato in modo da garantire un'integrazione bidirezionale con il Linguaggio C#. Risulta infatti possibile inserire classi e oggetti all'interno di applicazioni in Axum e, viceversa, è possibile scrivere pezzi di codice Axum in applicazioni scritte in C#.

L'unico vincolo, qualunque "tipo" di integrazione sia stato scelto, risulta essere quello di compilare l'applicazione attraverso il compilatore di Axum, il quale estende il compilatore per C# aggiungendo le funzionalità di Axum.

La possibilità di integrare codice Axum in applicazioni C# appare oggi come un punto decisamente a favore poiché, in questo modo, risulta possibile sfruttare tutte le ampie caratteristiche del linguaggio C# ed estenderlo con l'approccio alla concorrenza offerto da Axum.

Attraverso un'oculata scelta di quali porzioni di applicazione realizzare in Axum è possibile incrementare notevolmente le performance della applicazioni basate sul .NET Framework.

6.4 Applicazioni concorrenti distribuite

Il concetto di dominio rappresenta un'estensione al modello ad attori originale introdotta inizialmente in Axum con lo scopo di consentire a determinati agenti l'uso di un piccolo e controllato spazio in cui condividere dati.

Sebbene tale uso dei domini semplifichi spesso la realizzazione di applicazioni basate su Axum, la vera utilità dei domini emerge quando ci si propone l'obiettivo di realizzare applicazioni distribuite.

Il linguaggio Axum possiede decisamente un buon supporto per le applicazioni distribuite; infatti, per come sono stati definiti i meccanismi per la comunicazione tra agenti all'interno di un dominio, essi possono comunicare tra loro localmente oppure in remoto senza che nessuna modifica debba essere fatta alla logica dell'applicazione.

Il mapping tra i web services e Axum è immediato: i domini rappresentano i servizi di applicazioni service oriented (SOA), gli agenti sono responsabili della gestione dei protocolli di comunicazione e i dati *schema* definiscono il payload dei dati.

Per raggiungere un agente all'interno di un dominio è sufficiente conoscere il relativo indirizzo e questo vale sia in uno scenario locale sia in uno remoto, con l'unica differenza che per quello locale viene utilizzato un indirizzo di default trasparente al programmatore mentre in quello remoto va specificato il vero indirizzo.

Attraverso l'interfaccia IHost è possibile assegnare un indirizzo ad un agente (si veda a tal proposito anche il paragrafo 5.4.2) all'interno di un dominio.

L'esempio che segue realizza un server Axum-based che fornisce servizi attraverso domini.

```
channel SimpleChannel
{
  input string Msg1;
  output string Msg2;
}

domain ServiceDomain
{
  agent ServiceAgent : channel SimpleChannel
  {
    public ServiceAgent()
    {
      //...
    }
  }
}

agent Server : channel Microsoft.Axum.Application
{
  public Server()
  {
    var host = new WcfServiceHost(
          new NetTcpBinding(
           SecurityMode.None, false));

    host.Host<ServiceDomain.ServiceAgent>
          ("net.tcp://localhost/Service1");
  }
}
```

Ogni volta che un client si connette all'indirizzo `net.tcp://localhost/`

`Service1` viene creata una nuova istanza di `ServiceAgent` e contestualmente aggiunta alle istanze di `ServiceDomain`.

La connessione di un client al server proposto è effettuabile mediante il seguente codice.

```
var provider = new WcfCommunicationProvider(
       new NetTcpBinding(SecurityMode.None, false));

var channel = provider.Connect<SimpleChannel>
       ("net.tcp://localhost/service1");
```

Sebbene l'esempio proposto non esaurisca le possibilità di interazione di Axum con la piattaforma WCF (*Windows Communication Foundation*)[3], esso evidenzia come per Axum la gestione della concorrenza in applicazioni centralizzate oppure distribuite non cambi. Infatti, l'esempio mette in luce come sia possibile raggiungere agenti in remoto, tuttavia, nessuna logica per la gestione degli accessi concorrenti al server è stata implementata, in quanto essa è interamente gestita dai meccanismi di concorrenza. In definitiva quindi, un ulteriore aspetto positivo e degno di nota di Axum è dato dalla possibilità di combinare facilmente la logica relativa alla concorrenza e al parallelismo e la logica di distribuzione delle applicazioni.

[3]maggiori informazioni possono essere ottenute all'indirizzo `http://msdn.microsoft.com/en-us/netframework/aa663324.aspx`

Conclusioni

Descrivere l'introduzione della concorrenza come pilastro fondamentale ed imprescindibile per lo sviluppo di applicazioni in termini di *nuova rivoluzione del software* appare estremamente appropriato.

Ad oggi, non è più possibile progettare un'applicazione software accantonandone l'aspetto concorrente e tralasciando le possibilità di parallelizzazione.

Le nuove architetture hardware disponibili, le sempre maggiori richieste in termini di performance e le esigenze derivanti dalla vastità di dispositivi computazionali (computer, smartphone, tablet, ...) disponibili impongono necessariamente l'obbligo di progettare applicazioni efficienti e che sfruttino completamente le possibilità offerte.

Per questi motivi, quindi, evidenziare le lacune dell'attuale generale approccio alla concorrenza (la programmazione multithread) e conseguentemente analizzare metodi e modelli alternativi costituisce l'unico approccio mediante il quale poter ottemperare alle richieste poste ai progettisti e agli sviluppatori di software.

Il modello ad attori, nato diversi decenni fa ma solo ultimamente rivalutato e riconsiderato, è sicuramente tra gli approcci di riferimento per la gestione della concorrenza e può costituire una valida alternativa all'attuale modello orientato agli oggetti che, seppur efficiente, non consente un'ottima gestione della concorrenza e del parallelismo.

In base a tale convinzione e dopo aver introdotto e analizzato le attuali relazioni e richieste in termini di software e concorrenza, questa tesi ha considerato il modello ad attori e ne ha analizzato le caratteristiche.

È stato evidenziato come il meccanismo di scambio di messaggi asincrono alla base del modello costituisca un approccio chiaro e semplice al fine di garantire un flusso indipendente ed autonomo ai vari attori, mediante il quale eseguire la propria computazione parallelamente l'uno all'altro.

Inoltre, sono state descritte le caratteristiche che un attore deve possedere per potersi dichiarare tale. Incapsulamento, Atomicità, Fairness

e Location Transparency sono proprietà fondamentali nella definizione del modello ad attori, le quali consentono di evitare molti problemi che potrebbero insorgere con la programmazione di tipo multithread.

Dopo aver presentato alcuni esempi di come il modello ad attori sia effettivamente utilizzato per realizzare alcune tra le moderne applicazioni software, la tesi ha provveduto ad analizzare poi alcune tra le più significative implementazioni del modello ad attori. Per ognuna di esse si è dimostrato come il modello ad attori riesca a rendere più semplice e performante la realizzazione di applicazioni concorrenti.

Quindi, il caso di studio ha considerato l'implementazione del modello proposta da Microsoft: il linguaggio Axum. Dettagliatamente, sono state analizzate tutte le caratteristiche del linguaggio mediante le quali realizzare applicazioni basate sul modello ad attori.

Infine, è stata effettuata un'analisi critica sul linguaggio Axum al fine di mettere in evidenza le differenze con il modello ad attori originale ed i vantaggi introdotti.

In conclusione, quindi, si è potuto dimostrare come il modello ad attori sia effettivamente un buon modello di programmazione utilizzabile per progettare applicazioni software concorrenti e che sfruttino completamente le tecniche di parallelismo.

Inoltre, è emerso che il modello ad attori non sia un modello utilizzato unicamente per fini accademici, ma bensì sia attualmente implementato in diversi linguaggi i quali vengono utilizzati anche per lo sviluppo di applicazioni commerciali e non solo.

Il caso di studio poi, ha evidenziato infine come il linguaggio Axum sia una valida implementazione del modello ad attori e consenta contemporaneamente sia la completa flessibilità nello sviluppo di applicazioni sia la possibilità di integrazione con linguaggi ad oggetti al fine di perseguire l'obiettivo di massime performance.

Nonostante la sua non completa osservanza di tutte le specifiche imposte dal modello ad attori originale, Axum appare un linguaggio in definitiva adatto a rispondere alle crescenti esigenze per lo sviluppo di software concorrente.

La sintassi simile a quella object oriented del linguaggio C# rende il codice Axum facile da leggere e da esaminare. Relativamente immediato appare anche trascrivere in codice, applicazioni progettate per essere pienamente parallelizzabili.

Unico rammarico deriva dal fatto che Axum, ad oggi, risulta essere un progetto dichiarato concluso ma non inserito nelle piattaforme commerciali di Microsoft. Tuttavia, viste le sue potenzialità e dopo aver

ampiamente discusso gli aspetti che sarebbero da migliorare e quali invece risultano essere punti di forza, questa tesi si conclude con la speranza che il progetto Axum possa essere a breve riconsiderato, ampliato, ottimizzato e, in definitiva, commercializzato.

Ringraziamenti

Scegliere le parole giuste per ringraziare tutti coloro i quali abbiano contributo, in qualche modo, affiché io potessi raggiungere questo importante traguardo di vita non è compito facile.

Prime fra tutte, devo ringraziare le persone alle quali questa tesi è dedicata. Un ringraziamento speciale va ai miei genitori Meris e Pasquale, i quali mai hanno fatto mancare il loro sostegno morale, pratico ed economico. Sempre vigili sulla mia crescita e, anche nelle difficoltà, sempre pronti a verificare la solidità e l'adeguatezza delle scelte da me intraprese. Sempre pronti a tendermi una mano nei miei momenti di difficoltà e sempre disponibili a concedere momenti di riflessione e di dialogo. A loro devo gran parte di quello che sono e di quello che potrò diventare.

Un ulteriore ringraziamento speciale va a mio fratello Francesco, con cui ho condiviso molti momenti importanti e che mai ha fatto mancare considerazione per il percorso da me scelto nonché ammirazione e sostegno per i traguardi raggiunti.

Devo ringraziare poi le nonne Lucia e Maria le quali, pur essendo molto lontane dall'ambito professionale da me intrapreso, non hanno mai smesso di interessarsi agli obiettivi che avevo e ho tuttora intenzione di raggiungere in relazione al percorso universitario e non solo. Con vivo interesse e solidarietà, hanno sempre condiviso i momenti positivi e non e per questo sono loro molto riconoscente.

Un saluto va anche ai nonni Angelo, che non ho conosciuto, e Secondo, che ho conosciuto seppur per poco tempo; con la certezza che in questo momento sarebbero entrambi molto fieri di me.

Non posso poi dimenticare di ringraziare tutti gli altri parenti, i quali a proprio modo hanno sempre dimostrato alta considerazione nei miei

confronti e per questo sono molto fiero e grato.

Un enorme ringraziamento va poi a tutti gli amici, quelli veri, quelli a cui devo molto. A quegli amici che nei diversi momenti di difficoltà nati durante questi tre anni di università mi hanno sempre incitato a non arrendermi.

In particolare, devo ringraziare Pietro, Francesca e Lorenzo, con i quali ho avuto il privilegio di trascorrere questi anni di università e grazie ai quali poterli trasformare in un insieme di momenti gioiosi e piacevoli da affrontare. Con loro ho condiviso i successi, i traguardi intermedi e anche le difficoltà. Senza di loro, probabilmente, non sarei riuscito a raggiungere questo traguardo finale con la stessa serenità e la stessa soddisfazione. A loro devo molto, ciascuno infatti mi ha dimostrato un grado di considerazione e amicizia che è andato ben oltre ogni più rosea aspettativa.

Un vivo ringraziamento va poi a tutti i colleghi che hanno condiviso con me il percorso universitario, con i quali ho collaborato a progetti e condiviso conoscenze e idee.

Un veloce ma sincero ringraziamento è destinato a tutte le persone con le quali ho potuto condividere momenti piacevoli che hanno permesso a me di accantonare momentaneamente pensieri e stress dovuti all'università. Tra queste persone, in particolare, non posso dimenticare Silvia, la quale ha sempre dispensato buone ed appropriate parole di sostegno.

Un ulteriore ringraziamento va poi a Maurizio Conti, al quale devo profonda riconoscenza per avermi trasmesso importanti e significativi insegnamenti relativi al mestiere dell'informatico, perito o ingegnere che sia. La disponibilità dimostrata nel corso degli anni e i preziosi consigli offerti sono tali che difficilmente potranno essere adeguatamente ricompensati.

Un ringraziamento particolare va infine al Prof. Alessandro Ricci per la grande disponibilità dimostrata per la realizzazione di questa tesi, nonché per i preziosi consigli e le precise e dettagliate indicazioni fornite.

Bibliografia

[1] Gul Agha
ACTORS: a model of concurrent computation in distributed systems
1986, The MIT Press

[2] Herb Sutter, James Larus
Software and the Concurrency Revolution
2005, acm QUEUE, Vol. 3(7), pagg. 54-62

[3] Herb Sutter
The Free Lunch is over: a foundamental turn toward councurrency in software
2009, `www.gotw.ca/publications/concurrency-ddj.htm`

[4] Rajesh K. Karmani, Gul Agha
Actors
2011, `osl.cs.uiuc.edu/docs/actors/final-article.pdf`

[5] Carl Hewitt
Actor Model of Computation: Scalable Robust Information Systems
2011, Inconsistency Robutness '11, Stanford University

[6] Rajesh K. Karmani, Amin Shali, Gul Agha
Actor Frameworks for the JVM Platform: A comparative Analysis
2009, `http://osl.web.cs.illinois.edu/docs/pppj09/paper.pdf`

[7] Paul Mackay
Why has the actor model not succeeded?
1997, `www.doc.ic.ac.uk/~nd/surprise_97/journal/vol2/pjm2/`

[8] Microsoft Corporation
Axum Programmr's Guide
2009, `http://download.microsoft.com/download/B/D/5/BD51FFB2-C777-43B0-AC24-BDE3C88E231F/Axum%20Programmers%20Guide.pdf`

[9] Microsoft Corporation
Axum Language Specifications
2009, http://download.microsoft.com/download/B/D/5/BD51FFB2-C777-43B0-AC24-BDE3C88E231F/Axum%20Language%20Spec.pdf

[10] Stephen Toub
What's New For Parallelism in .NET 4.5
2011, http://blogs.msdn.com/b/pfxteam/archive/2011/09/17/10212961.aspx

[11] Ruben Vermeersch
Concurrency in Erlang & Scala: the Actor Model
2009, http://ruben.savanne.be/articles/concurrency-in-erlang-scala

[12] J. Armstrong, R. Virding, C. Wikstrom, M. Williams
Concurrent Programming in Erlang, 2^{nd} edition
1996, Prentice Hall

[13] M. Odersky, L. Spoon, B. Venners
Programming in Scala
2008, Artima Press

[14] S. Abbasi, Z. Zahid
ActorFoundry: A Tutorial Guide
http://osl.cs.uiuc.edu/af/tutorial.pdf

[15] Henry Lieberman
A preview of Act 1
1981, Massachusetts Institute of Tecnology, Artificial Intelligence Laboratory
ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-625.pdf

[16] Bill Venners
Twitter on Scala: a conversation with S. Jenson, A. Payne and R. Pointer
2009, http://www.artima.com/scalazine/articles/twitter_on_scala.html

[17] Redfin Developers' Blog
Engineer-to-Engineer Talk: How and Why Twitter uses Scala

2010, http://blog.redfin.com/devblog/2010/05/how_and_why_twitter_uses_scala.html

[18] *A First Look at DART, The Programming Language*
2011, http://www.replicator.org/content/a-first-look-at-dart-the-programming-language

[19] *Dart API reference - dart:isolate*
http://api.dartlang.org/dart_isolate.html

[20] James Larus et al.
Orleans: A Framework for Cloud Computing
2010, http://research.microsoft.com/en-us/projects/orleans/

[21] Matthew Podwysocki
Axum - Introduction ad PingPong Example
2009, http://codebetter.com/matthewpodwysocki/2009/05/12/axum-introduction-and-ping-pong-example/

[22] Matthew Podwysocki
Axum - PinPong with Dataflow Networks
2009, http://codebetter.com/matthewpodwysocki/2009/05/15/axum-ping-pong-with-dataflow-networks/

[23] Matthew Podwysocki
Axum - PingPong with ordered Interaction Points
2009, http://codebetter.com/matthewpodwysocki/2009/06/04/axum-ping-pong-with-ordered-interaction-points/