ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA
SCUOLA DI INGEGNERIA E ARCHITETTURA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

APPLICAZIONE DEL MODELLO DI CONCORRENZA AD ATTORI NELLO SVILUPPO DI APPLICAZIONI WEB: UN CASO DI STUDIO

Tesi in
SISTEMI OPERATIVI

Relatore:
Chiar.mo Prof. Ing.
ALESSANDRO RICCI

Presentata da:
MATTEO VENTURI

Sessione II
Anno Accademico 2015 – 2016

Alla mia famiglia e a Marica

Indice

Introduzione

Il World Wide Web, come lo conosciamo, oggi era molto diverso ai suoi albori ed ha subito una lenta evoluzione fino ai giorni nostri attraversando tre ere principali. La sua evoluzione è stata influenzata principalmente da due fattori: la tecnologia e gli utenti finali.

Con il Web sono cambiati i computer, diventando più potenti e acquisendo la capacità di effettuare più operazioni simultaneamente, e gli stessi utilizzatori. Nel corso del tempo l'utente medio di queste tecnologie si avvicina sempre di più ad essere un utilizzatore che non possiede specifiche conoscenze informatiche ma che utilizza il Web ed i devices collegati ad esso per svolgere compiti mirati ai propri interessi ed alle proprie attività.

Come il Web, e grazie alla sua evoluzione, anche le azioni svolte tramite la Rete si sono evolute attraversando un percorso guidato da fattori quali la diffusione dei computers e l'accesso sempre più facilitato al Web. Nascono nuovi modi di comunicare, studiare, condividere informazioni ma anche di acquistare beni di consumo.

Nel corso del tempo anche i linguaggi di programmazione affrontano un'evoluzione spinta dalla ricerca di paradigmi sempre più espressivi e potenti e trovando una naturale continuazione della programmazione imperativa nel Object Oriented Programming (OOP). Supportati dall'hardware sempre più potente e veloce, i linguaggi di programmazione si sviluppano e da un tipo di programmazione puramente sequenziale passano ad una che riesce a sfruttare appieno le capacità concorrenti dell'hardware. In questo modo la velocità computazionale ed operativa dei calcolatori aumenta vertiginosamente e ciò per il Web vuol dire che un gran numero di richieste, ricevute da un web server, possono essere soddisfatte nello stesso momento e quindi garantire bassi tempi di latenza ed un alto throughput.

Il passaggio dalla programmazione sequenziale a quella concorrente obbliga ad affrontare difficoltà legate alla corretta gestione della concorrenza. La prima soluzione adottata

è quella di regolare gli accessi alle risorse con meccanismi di coordinazione per evitare situazioni di malfunzionamento dovuti al fatto che la medesima risorsa sia richiesta da più utilizzatori nello stesso momento. Un altro approccio è quello di cambiare radicalmente paradigma passando dal modello COOP (Concurrency Object Oriented Programming) al modello ad Attori in cui non ci sono meccanismi espliciti di regolazione della concorrenza in quanto il modello stesso permette di gestire correttamente questo tipo di programmazione.

Per quanto riguarda il Web il cambio di approccio è dettato anche da fattori quali l'affidabilità che il modello garantisce con i meccanismi offerti e la scalabilità delle risorse utilizzate facilita dalla natura flessibile e modulare di tale modello. Inoltre le alte performance offerte dalle ultime implementazioni, che fanno utilizzo dei più moderni linguaggi di programmazione, ne fanno un valido strumento applicabile in campo aziendale.

Ad oggi questa evoluzione ha portato allo sviluppo di soluzioni che impiegano un'architettura totalmente diversa rispetto quelle citate: un esempio è Node.js. Il modello di programmazione event-driven per il Web, applicato lato server, si fa sempre più spazio garantendo la sua esecuzione su qualsiasi sistema operativo ed offrendo caratteristiche di grandi performance per le operazioni I/O-bound.

Capitolo 1

L'evoluzione del web

> "Il World Wide Web ha le potenzialità per svilupparsi in un'enciclopedia universale che copra tutti i campi della conoscenza e in una biblioteca completa di corsi per la formazione."[1]

Si ritiene che la data di nascita del Web sia il 6 Agosto 1991, giorno in cui il programmatore britannico Tim Berners-Lee pubblicò il primo sito web. Pochi anni prima lo stesso Tim Berners-Lee propose al proprio supervisore presso il CERN l'idea di elaborare un software che permettesse di migliorare la comunicazione e la cooperazione tra i colleghi agevolando lo scambio di materiale scientifico in formato elettronico. Il progetto WorldWideWeb comprendeva lo sviluppo del software (browser e web server) e la definizione degli standard (HTML) e dei protocolli (HTTP) per lo scambio di documenti tra calcolatori. Il 30 Aprile 1993 il CERN pubblicò il WWW al pubblico rinunciando ad ogni diritto d'autore. Il World Wide Web era nato, ed era nato libero.

1.1 Il Web 1.0: fruizione della conoscenza e business

Si definisce la prima versione del Web tra il 1991 e il 2001. In questi anni il Web è fortemente orientato all'informazione e l'utente copre il ruolo principalmente di fruitore della conoscenza. Nel 1994 il padre del Web Tim Berners-Lee fonda il W3C in collaborazione con il CERN. Scopo di questa organizzazione non governativa ed internazionale è quello

[1] *L'enciclopedia universale libera e le risorse per l'apprendimento*, Richard Stallman

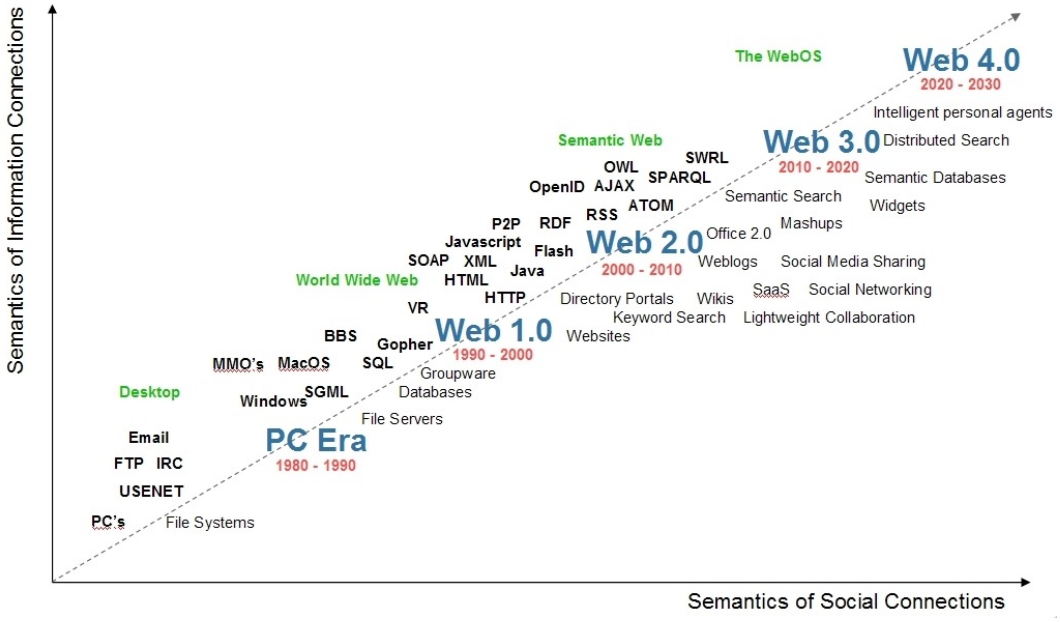


Figura 1.1: *Evoluzione del Web*, Radar Networks & Nova Spivack

di "stabilire standard tecnici per il World Wide Web inerenti sia i linguaggi di markup che i protocolli di comunicazione."[2]

Il Web si evolve acquisendo sempre più informazioni, che coprono svariate aree del sapere umano, ma queste sono poco organizzate, per lo più in forma testuale e filtrarle in modo automatico, da parte dei motori di ricerca, è complesso. I siti di questa prima fase del Web sono caratterizzati da pagine con contenuti statici e l'interazione avviene tra il sito e il singolo utente. Con il passare degli anni nascono i primi motori di ricerca che permettono agli utenti di cercare le informazioni desiderate, i portali che danno accesso ai servizi e alle informazioni delle aziende e i primi eCommerce. Da molti il Web viene visto principalmente come una nuova opportunità di business, dove le distanze si annullano e i potenziali clienti si moltiplicano, ma non perde l'idea di fondo che l'ha generato: uno strumento libero di promulgazione della conoscenza.

I siti Web non sono gli unici protagonista di questo periodo: le informazioni attraversano la Rete sfruttando servizi come l'FTP e gli utenti interagiscono tra loro tramite la posta elettronica (e-mail), i newsgroup, le chat e i forum.

[2] *World Wide Web Consortium*, Wikipedia

1.2 Il Web 2.0: creazione della conoscenza e rete sociale

"L'uomo è un'animale sociale" [3]

Si associa l'inizio della seconda Era del Web attorno al 2003 e fino al 2011.

Il carattere sociale del Web prende il sopravvento e la continua evoluzione della Rete prosegue sempre più nella direzione dell'interazione tra gli utenti che, da un ruolo passivo, passano ad avere sempre più un ruolo attivo. Ora coloro che utilizzano il Web non sono più i fruitori passivi che visitavano i siti del Web 1.0. Gli utenti del Web 2.0 diventano generatori della conoscenza ed interagiscono tra di loro attivamente tramite blog e successivamente strumenti sempre più mirati a tale scopo: i social networks.

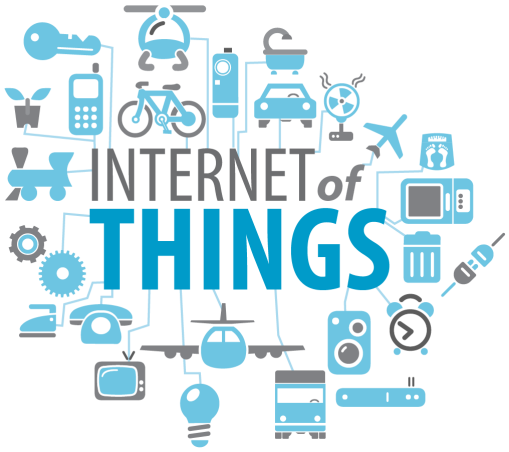
Il Web cresce e con se la conoscenza contenuta, le tecnologie impiegate e i servizi offerti. Ci si rende conto che ormai il semplice collegamento ipertestuale non è più sufficiente. Per permettere una strutturazione ordinata e rigorosa della conoscenza si necessita che "i documenti pubblicati (pagine HTML, file, immagini, e così via) siano associati ad informazioni e dati (metadati) che ne specifichino il contesto semantico in un formato adatto all'interrogazione e l'interpretazione (es. tramite motori di ricerca) e, più in generale, all'elaborazione automatica."[4]La teoria di un Web semantico inizia a prendere corpo in questo periodo con la definizione del Resource Description Framework (RDF), "strumento base proposto da W3C per la codifica, lo scambio e il riutilizzo di metadati strutturati e che consente l'interoperabilità tra applicazioni che condividono le informazioni sul Web"[5].

1.3 Il Web 3.0: personalizzato per l'utente

L'Uomo diventa il vero fulcro del Web il quale si modella e si personalizza a seconda dell'utilizzatore. Ad esempio la ricerca delle informazioni è calibrata a seconda del contesto dell'utente e i siti internet si adattano al dispositivo con cui vengono visualizzati. La tecnologia permette la fruizione del Web da quasi ogni angolo del pianeta grazie a connessioni wireless, veloci ed affidabili ed a dispositivi potenti, piccoli e portatili, come gli smarphone e i tablet. L'essere Umano raggiunge una sorta di ubiquità virtuale considerando che da

[3] *Politica*, Aristotele
[4] *Web semantico*, Wikipedia
[5] Resource Description Framework, Wikipedia

qualsiasi luogo in cui si trova può comunicare, vedere e interagire con chiunque in qualsiasi posto del pianeta.



Il Web inizia a conoscere dispositivi connessi sempre più intelligenti tanto da essere assimilati a dei veri e propri utenti senzienti. Elettrodomestici intelligenti, sveglie che suonano in anticipo in caso di traffico, smarphone che svolgono funzioni da assistente personale e tanti altri pervadono la Rete e comunicano tra loro, reperendo e producendo informazioni. Internet non appartiene più

unicamente agli essere umani ma anche alle "cose": Internet Of Things.

1.4 Web Applications e Web Services

Oggi possiamo paragonare il Web ad un grande sistema operativo distribuito in cui tutti siamo utenti, fruitori e creatori delle informazioni contenute. Il Web contiene molte applicazioni che si distinguono in due macro categorie: Web Applications e Web Services. La prima categoria include tutto ciò che concerne l'esperienza utente nel Web: dai siti internet più statici a quelli maggiormente interattivi, ai social networks, agli e-commerce, fino a vere e proprie applicazioni che sembrano installate sul PC ma in realtà sfruttano la tecnologia del Web. La seconda categoria comprende tutti i servizi adatti alla comunicazione tra macchine e che quindi un normale utente della Rete non utilizza direttamente.

1.4.1 Web Applications

Un aspetto dell'evoluzione del Web è stato sicuramente il progressivo avvicinarsi della tecnologia ad uno dei più importanti aspetti che caratterizza l'Essere Umano: la dinamicità. Come un blocco di granito nelle mani di uno scultore così il Web, giorno dopo giorno, prendeva forma, si trasformava ed evolveva verso una direzione sempre più vicina all'intel-

letto Umano. Vennero alla luce tecnologie che avevano come scopo quello di rendere più dinamiche ed interattive le pagine Web, in modo che l'utente le trovasse più utili e facili da consultare: Javascript, AJAX, CSS3 ed HTML5. Sono queste le tecnologie che oggi permettono ad una pagina di dialogare con l'utente a tal punto da offuscare i confini tra Web e PC. Vere e proprie applicazioni, che un tempo risiedevano nella memoria interna del computer, oggi sono disponibili in un ambiente condiviso e distribuito come il Web, utilizzate da più persone allo stesso tempo, con lo scopo condividere informazioni e cooperare insieme. Un esempio è la famiglia delle Web Applications di Google grazie alle quali gli utenti possono condividere fogli elettronici e lavorare nello stesso momento sullo stesso foglio. La condivisione è un filone che percorre tutto il Web dalla sua seconda versione in avanti e su questa scia nascono i social networks come Facebook e Twitter nei quali gli utenti, praticamente in tempo reale, condividono esperienze, passioni, hobby ed emozioni. Anche gli e-commerce sviluppano un importante aspetto sociale e si evolvono di conseguenza utilizzando come efficaci strumenti di marketing i commenti, i voti e le condivisioni degli utenti sui social networks. Anche il mondo dei videogames non rimane indifferente al Web ed al suo carattere sociale. Ecco perché i videogiochi multiplayer on-line prendono sempre più piede fino ad arrivare all'apice dell'interazione con titoli che creano un vero e proprio mondo virtuale parallelo a quello reale dove le persone possono avere il loro alter ego e interagire con le altre come farebbero nel mondo reale.

1.4.2 Web Services

Sono quei servizi disponibili sul Web ed utilizzati per comunicazioni machine-to-machine. Il primo approccio fu quello di replicare i tradizionali servizi basati su Remote Procedure Call sul WWW utilizzando il protocollo HTTP. Questi servizi facevano uso delle richieste POST del protocollo per scambiare la richiesta e la risposta con i rispettivi valori in formato XML. Questa tecnologia utilizzava il protocollo HTTP solo come protocollo di trasporto, senza avvalersi delle molteplici proprietà come i codici di stato, la gestione degli errori, gli

attributi dell'header per la negoziazione della comunicazione o la cache.

In seguito prese sempre più piede lo stack SOAP/WSDL come evoluzione dell' XML-RPC. Questa famiglia di web services differisce dalla precedente per una rigorosa descrizione del servizio, grazie al WSDL (Web Service Description Service), un documento XML che specifica principalmente la sintassi e la semantica dei metodi esposti. Anche questa tipologia di Web Service non utilizza tutte le facility del protocollo HTTP ma altresì ne pone al di sopra, a livello applicativo, un ulteriore, il protocollo SOAP (Simple Object Access Protocol), per gestire lo scambio di messaggi.

L'ultima nata è la famiglia di web services RESTful. Il termine REST apparve la prima volta nel 2000 nella tesi di dottorato di Roy Fielding. Questo tipo di servizio si basa sui seguenti concetti:

- Lo stato dell'applicazione e le funzionalità sono divisi in risorse web
- Ogni risorsa è unica e indirizzabile usando sintassi universale per uso nei link ipertestuali
- Tutte le risorse sono condivise come interfaccia uniforme per il trasferimento di stato tra client e risorse, questo consiste in:
  - un insieme vincolato di operazioni ben definite
  - un insieme vincolato di contenuti, opzionalmente supportato da codice on demand
- un protocollo che è:
  - client-server
  - stateless
  - cachable
  - a livelli

Capitolo 2

Architetture Web tra passato e presente

2.1 Generazione di contenuti dinamici nelle pagine Web

2.1.1 L'interfaccia CGI

Negli anni '90 il Web forniva pagine HTML a contenuto statico e l'utente poteva solamente leggerne il contenuto. Con il tempo la richiesta di dinamicità diventava sempre più forte e si richiedeva ai web server la capacità di generare dinamicamente le pagine in base a parametri forniti o determinate condizioni. Il primo approccio a questo problema fu quello di demandare la generazione dei contenuti a processi esterni tramite l'interfaccia standard CGI (Common Gateway Interface)[1]. Questa interfaccia permetteva al web server di lanciare un processo passando i parametri necessari per l'elaborazione e di ricevere il risultato. Di per sé l'interfaccia si occupava di creare un nuovo processo che eseguiva il codice script. Questo approccio era afflitto principalmente dai seguenti problemi:

1. impossibilità di scalare in modo orizzontale poiché il processo poteva essere lanciato solo sulla macchina locale su cui risiedeva il server (comunicazione tramite STDIN e STDOUT)

2. penalità nelle performance perché ad ogni richiesta CGI creava un nuovo processo ed il relativo overhead

[1] ROBINSON, D. COAR, K.: The Common Gateway Interface (CGI) Version 1.1, RFC 3875 (Informational) (2004)

Per ovviare a queste penalizzazioni venne rilasciata una seconda versione dell'interfaccia denominata *FastCGI* che cambiava la modalità di comunicazione tra web server e script da STDIN e STDOUT in comunicazioni tramite protocollo TCP. Questo approccio permetteva di mettere in esecuzione i processi su una macchina diversa da quella su cui si eseguiva il web server. Invece per rispondere al problema rilevato al punto 2 si pensò di eseguire un processo a lungo termine e, per ogni richiesta, di creare un thread che eseguiva il comando desiderato.

2.1.2 I moduli server

Un'alternativa a CGI furono i moduli server. Questi moduli estendevano dinamicamente il comportamento di un web server, come fossero dei plug-in, integrando interpreti script che potevano essere eseguiti dallo stesso thread o processo. In questo modo si aumentavano le performance, non essendoci la necessità di creare un nuovo processo o thread come facevano CGI e FastCGI, ma tale modello ancora rendeva difficoltoso il disaccoppiamento tra la macchina server e la macchina di back-end.

2.1.3 Container web application

Per alcuni linguaggi che vengono eseguiti da una virtual machine, come Java, l'approccio utilizzato da CGI, di creare per ogni richiesta un processo, non è sfruttabile in quanto il tempo di creazione di un processo ed il tempo di avvio della stessa macchina virtuale sarebbero troppo onerosi. Per questi motivi furono stilate le specifiche per un nuovo approccio: le Java Servlet. Questo approccio consiste principalmente in un container, comunicante con il web server attraverso un protocollo dedicato (Apache JServ Protocol[2]), e che gestisce le richieste che gli giungono smistandole ad un pool di thread già precedentemente istanziati. Con questo approccio le performance non subiscono degradi e si favorisce il disaccoppiamento tra macchina web server e macchina back-end.

[2]SHACHOR, Gal MILSTEIN, Dan: The Apache Tomcat Connector - AJP Protocol Reference, , Apache Software Foundation (2000)

2.2 Gestione e strategie di utilizzo delle risorse hardware nelle architetture server

Con l'aumentare degli utenti del Web cresce anche la necessità di gestire simultaneamente più connessioni allo stesso tempo in modo da accontentare tutti gli utilizzatori. Un po' come un piccolo negozio che giorno dopo giorno accresce la propria clientela e si accorge che necessita di adattarsi al nuovo afflusso di clienti.

I parametri per misurare le performance di un server sono:

- il tempo di riposta ad una richiesta in millisecondi
- la quantità di dati scambiata in Mbps
- il numero di richieste servite al secondo
- il numero di connessioni concorrenti

Inoltre è utile considerare anche parametri legati all'hardware del server quali:

- utilizzo della memoria
- utilizzo della CPU
- numero di socket o file aperti
- numero di Threads o processi

Lo scopo è quello di utilizzare il minimo delle risorse disponibili per poter gestire quante più richieste in parallelo e nel modo più veloce possibile. Per arrivare a questo traguardo esistono svariate tecniche ed approcci utilizzabili a seconda delle necessità.

2.2.1 Modelli Input/Output: modalità bloccante e non bloccante

Queste due modalità di accesso istruiscono il sistema operativo circa l'accesso ai dispositivi di I/O. La modalità bloccante impedisce all'operazione di terminare fin quando la lettura o la scrittura dei dati non sarà completamente terminata e quindi restituire l'eventuale informazione richiesta all'operazione. Al contrario la modalità non bloccante permette all'operazione di ritornare al chiamante notificando lo stato della chiamata. Questo

comportamento presuppone che ad un certo punto il thread o processo attenda l'esito della chiamata recuperando le eventuali informazioni.

2.2.2 Modelli Input/Output: modalità sincrona ed asincrona

Queste due modalità sono utilizzate per descrivere il flusso di controllo durante le operazioni di I/O. Una chiamata sincrona sottintende che il controllo non tornerà al chiamante fin tanto che l'operazione non si sarà conclusa. Invece una chiamata asincrona permetterà al chiamante, invocata l'operazione di I/O, di riavere immediatamente il controllo per poter effettuare altre operazioni.

2.2.3 Strategie di Input/Output

Le quattro modalità sopra descritte possono essere utilizzate a coppie per ottenere comportamenti atti ad espletare al meglio le onerose operazioni di scrittura o lettura di informazioni.

Bloccante - Sincrona

Questa è la modalità "classica" con la quale i processi o Threads interagiscono con i dispositivi di I/O. Con questa modalità l'operazione risulta essere unica, bloccando l'applicazione finché l'operazione non è terminata e le informazioni non sono state copiate dal kernel space allo user space(nel caso di lettura). La CPU non è utilizzata durante l'attesa e questo è un buon momento per lo scheduler del sistema operativo di assegnarla ad un altro processo.

Non bloccante - Sincrona

In questa modalità il chiamante ottiene immediatamente il controllo subito dopo aver effettuato la chiamata. Solitamente il dispositivo di I/O non è pronto a fornire i dati, o a produrli, pertanto il chiamante deve attendere finché l'operazione sul dispositivo non è terminata. Questa attesa, denominata busy-wait, è dispendiosa ed inefficiente. Una volta che l'operazione di I/O risulta terminata, ed eventualmente i dati sono stati copiati dal kernel space allo user space, il chiamante può continuare la propria esecuzione.

Non bloccante - Asincrona

Questa modalità possiede buone performance in caso di estrema concorrenza di I/O. Il chiamante effettua un'operazione di I/O la quale ritorna immediatamente permettendo

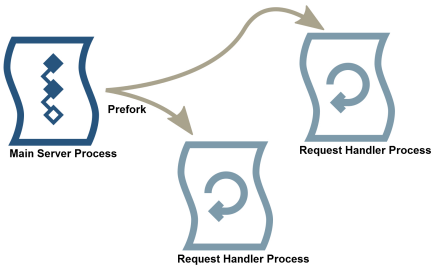
al processo o thread di proseguire la propria esecuzione. Quando l'operazione di I/O sarà terminata allora verrà generato un evento oppure invocata una chiamata di callback. Con questa modalità non ci sono attese inefficienti e l'intera operazione viene spostata lato kernel. Ciò significa che l'applicazione può eseguire in parallelo durante un'operazione di I/O.

2.3 Principali architetture server

Sono due le principali architetture server tutt'ora utilizzate: la prima è basata sui processi/thread, la seconda sugli eventi. La bontà di queste architetture è stata confermata nell'ultimo decennio anche grazie l'evoluzione tecnologica che ha portato i web server verso l'utilizzo di sistemi hardware multi-CPU ed ottimizzati per supportare un alto livello di concorrenza. Dal punto di vista storico l'architettura ad eventi è la più "giovane" tra le due perché il suo utilizzo è strettamente legato alla capacità del sistema operativo di effettuare chiamate di I/O asincrone/non bloccanti ed anche alle performance del meccanismo di notifica degli eventi.

2.3.1 Architetture server basate sui processi/thread

Questi tipi di architetture associa ogni richiesta di connessione ad un processo/thread separato detto "worker".



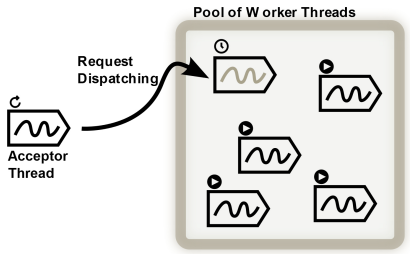
(a) Architettura multi processo che utilizza il prefork

Viene da se che molteplici richieste sono servite da molteplici worker in parallelo, quindi ogni singolo worker utilizzerà le risorse I/O in modo sincrono e bloccante. Questa modalità è supportata dalla quasi totalità dei linguaggi di programmazione, ne segue un modello di programmazione molto rigido, sequenziale e quindi "lineare" nel comportamento. Questo approccio ha il vantaggio di offrire un'astrazione mentale semplice poiché ogni richiesta ri-

sulta isolata nascondendo la concorrenza. Le richieste vengono eseguite concorrentemente grazie all'esecuzione di molteplici thread allo stesso tempo.

L'approccio "connessione per processo" è stato storicamente il primo ad essere utilizzato, ad esempio dal CERN per il suo web server. Per via della natura dei processi questi sono isolati e le chiamate non condividono memoria. La creazione di un processo è onerosa in termini di risorse per cui i server spesso implementano una strategia chiamata "preforking". Il server, in fase di avvio, crea preventivamente un numero definito di processi che andranno a gestire le chiamate in arrivo al server. Di solito il socket è condiviso tra i processi ed ognuno di questi rimane in attesa di una nuova richiesta, la gestisce e poi torna ad attendere la richiesta successiva.

Il passo evolutivo successivo di questi tipi di architetture fu l'utilizzo dei thread al posto dei processi. Il modello che ne risultava seguiva la stessa struttura del modello che utilizzava i processi ma con notevoli vantaggi. I thread sono unità lavorative più "leggere", la loro creazione e distruzione è meno costosa in termini di risorse, condividono lo stesso spazio di memoria e le stesse variabili (es. cache condivisa). Questi motivi portano a preferire i thread, soprattutto per l'utilizzo in situazioni ad alta concorrenza dove l'utilizzo dei processi, a causa delle loro dimensioni, rendeva difficoltosa la gestione delle risorse.



(b) Architettura multi thread che utilizza un pool

Nella pratica questo modello trova impiego[3] in un utilizzo in cui un thread, con funzione di dispatcher, accetta le richieste provenienti dal socket e le deposita in una coda. Un pool di thread, preventivamente creati, si occupa di prelevare richieste da tale coda e di gestirle. Nel caso in cui la coda, che ha dimensione finita e decisa in fase di inizializzazione, viene totalmente riempita allora le richieste verranno scartate. Questo metodo possiede un approccio limitativo verso la concorrenza ma previene il sovraccaricamento, definendo il numero dei thread all'interno del pool a priori, e garantisce una latenza più predicibile.

Le due strategie qui sopra introdotte posseggono uno svantaggio notevole in casi di

[3]STEVENS, W. Richard; FENNER, Bill RUDOFF, Andrew M.: Unix Network Programming, Volume 1: The Sockets Networking API (3rd Edition), Addison-Wesley Professional (2003)

pieno carico: il frequente cambiamento di contesto da parte dello scheduler causa un notevole overhead e perdita di tempo CPU se confrontato con un approccio a singolo processo o thread. Per questo motivo è importante trovare un compromesso tra scalabilità e performance, stabilendo accuratamente il numero di massimo di thread in modo: un basso numero di thread penalizza la concorrenza ma aumenta le performance, un alto numero di thread favorisce la scalabilità dell'architettura penalizzando le performance di ogni singolo thread.

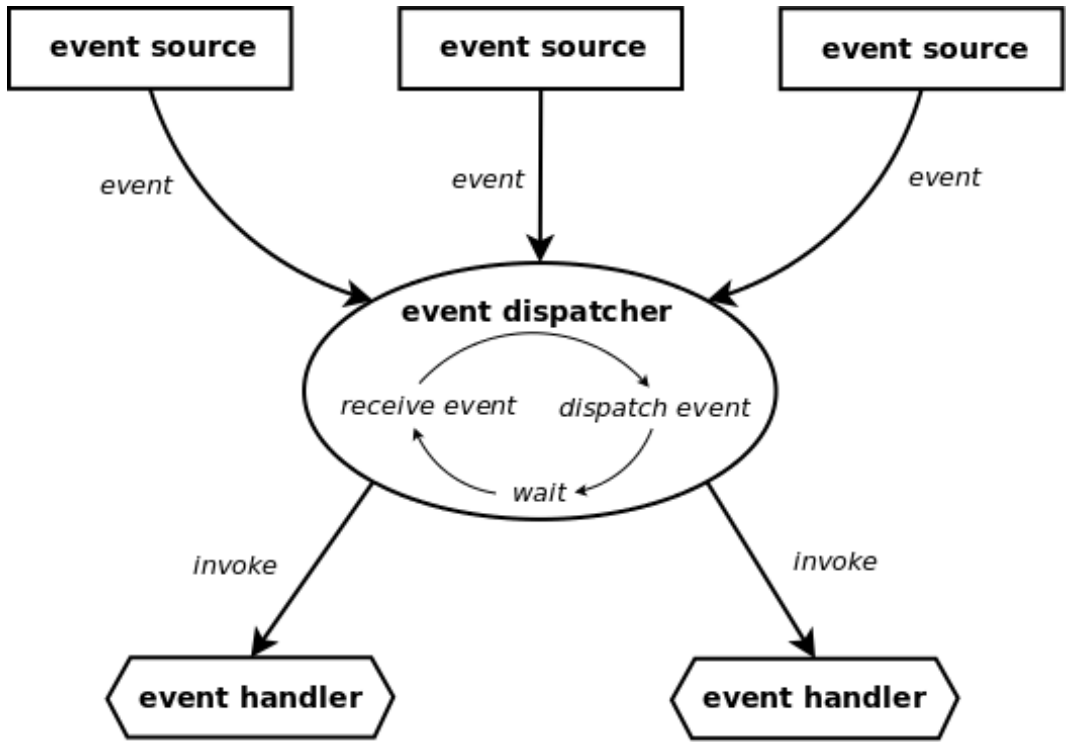
2.3.2 Architetture server basate sugli eventi

Un valido approccio alternativo alle classiche operazioni I/O sincrone e bloccanti è quello che utilizza gli eventi. Il modello più comune prevede un unico thread per più connessioni. I nuovi eventi vengono accodati dal sistema in una coda di eventi, il thread compie una chiamata di sistema che preleva un evento dalla coda, lo processa ed infine torna a prelevare eventi o, in mancanza, rimanere in attesa di nuovi. In base al tipo di evento ricevuto il thread esegue il gestore appropriato ed al termine può eseguire un metodo chiamato callback. La callback può essere decisa a priori oppure passata al thread assieme l'evento stesso. Questo approccio rende il flusso di controllo più difficile da eseguire, soprattutto per quanto riguarda il debugging, poiché non c'è una sequenza di operazioni da eseguire ma una cascata di chiamate asincrone e callbacks.

2.3.2.1 Pattern Reactor

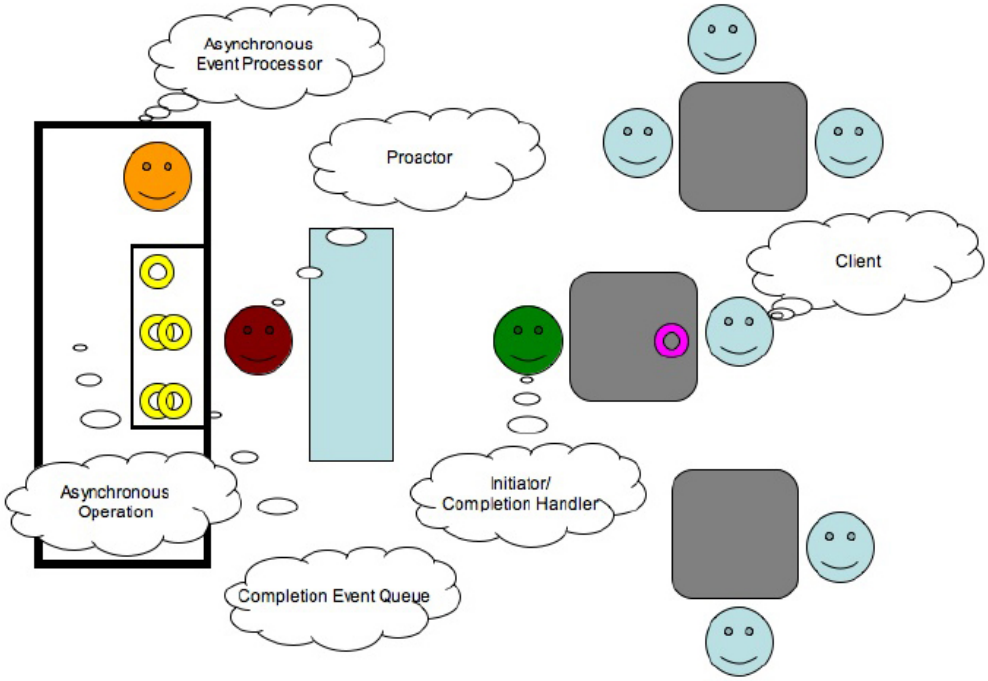
Un pattern molto utilizzato in ambienti event-driven è il pattern Reactor[4] che mira a gestire in modo non bloccante gli eventi del sistema. Con questo pattern un singolo thread, il **Dispatcher**, raccoglie le richieste in ingresso in modo sincrono/non bloccante, subito dopo l'accettazione della richiesta viene invocato il gestore, o il metodo di callback, appropriato. Esiste una versione multi-thread di questo modello in cui più gestori d'evento vengono messi in esecuzione su più thread. Un modello di questo tipo permette di sfruttare appieno le architetture multi-core, di contro necessita di tecniche di sincronizzazione per l'utilizzo delle risorse condivise.

[4] **Reactor** - An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events Douglas C. Schmidt - Department of Computer Science Washington University, St. Louis, MO



2.3.2.2 Pattern Proactor

Differentemente dal pattern precedente, il pattern Proactor può essere interpretato come la versione asincrona/non bloccante del Reactor. Questo pattern, rispetto il pattern Reactor in versione multi-thread, elimina la complessità della sincronizzazione e l'overhead dovuto al cambio di contesto. Possiamo descrivere questo pattern utilizzando come analogia un ristorante. In un ristorante ci sono più clienti (**Client**), ciascuno comunica la propria ordinazione (**Event**). Il cameriere (**Initiator**) ha come compito quello di ricevere un'ordinazione e di riferirla al supervisore (**Proactor**). Questi, una volta informato dell'ordine, comunica con la cucina riferendo al cuoco(**Asynchronous Event Processor**) l'ordine e torna ad attendere l'ordinazione successiva proveniente dal cameriere. Quest'ultimo raccoglie le ordinazioni di altri clienti. Quando una pietanza (**Event result**) è pronta, il cuoco avvisa il supervisore dei camerieri il quale prende la pietanza e la affida al cameriere, il quale a sua volta la consegna al cliente.


Asynchronous
Event Processor
Proactor
Client
Asynchronous
Operation
Initiator/
Completion Handler
Completion Event Queue

Capitolo 3

Concorrenza e Thread

Un web server possiede un comportamento dinamico tale per cui ad ogni richiesta viene generata come risultato una risposta. Il server, per generare una risposta corretta, esegue un flusso di controllo che si basa su una logica applicativa: la Business Logic. Solitamente la Business Logic di una applicazione Web di medio/alta complessità è formata da due macro parti principali:

- una parte puramente computazionale in cui il server utilizza in modo intensivo la CPU.
- una parte di interazione con componenti di I/O quali database, web services esterni, file system, ecc...

Nel caso di un web server sequenziale sarebbe possibile solamente una di queste operazioni alla volta, bloccando l'esecuzione di ogni altra richiesta. Inoltre il server sprecherebbe tempo prezioso rimanendo in attesa del risultato di una di queste azioni. Per minimizzare i tempi di risposta del server (latenza) è necessario parallelizzare tutte quelle operazioni che sono indipendenti dalle altre in modo da sfruttare le potenzialità delle moderne piattaforme multi-core e permettere l'utilizzo dell'applicazione web a più utenti allo stesso momento evitando inutili attese durante le operazioni e massimizzando lo throughput.

Parallelizzare le richieste non è l'unica forma di concorrenza che un server può attuare. Infatti per ogni singola richiesta può essere necessario svolgere in modo parallelo diverse azioni indipendenti le une dalle altre. Ad esempio immaginiamo che l'esecuzione di una richiesta consista nel reperire informazioni da diverse fonti esterne al server quali database

e web services. Infine i dati vengono verificati, elaborati ed aggregati per poi essere inviati in risposta al client. Ogni singola fase di reperimento delle informazioni, così come la loro verifica, è indipendente dalle altre. Per questo motivo l'esecuzione parallela porta sicuramente un notevole beneficio prestazionale.

L'utilizzo condiviso delle risorse non è l'unico problema da affrontare negli scenari concorrenti. Un'altra difficoltà si incontra quando più Threads necessitano di cooperare per compiere un determinato compito. Riprendendo l'esempio precedente immaginiamo che l'informazione reperita da un Thread sia ritenuta valida solo se anche l'altro Thread riesce a portare a termine il proprio lavoro. In caso positivo entrambe le informazioni dovranno essere unite per poi formare la risposta da fornire al client. Altresì, nel caso in cui uno dei due Thread non riesca a portare a termine il proprio compito, verrà inviato al client un segnale di errore. Questo scenario porta alla luce un secondo aspetto da tener in considerazione: la cooperazione.

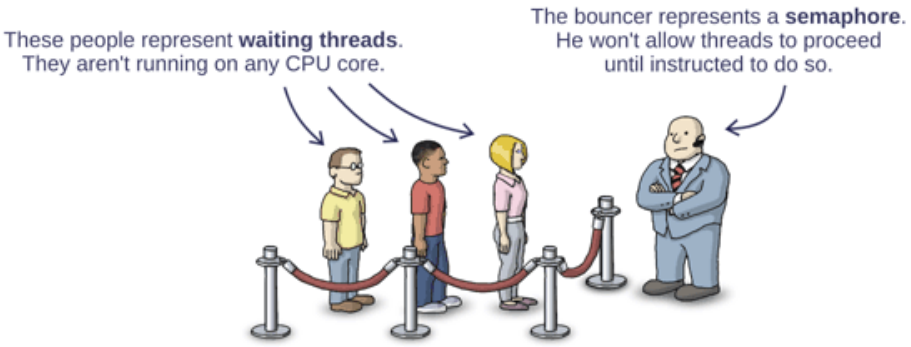
La programmazione a thread è oggi la forma di programmazione più utilizzata. Evoluzione dei processi, i Threads sono il principale modello per sfruttare il parallelismo nei sistemi operativi e nelle architetture hardware. D'altro canto, la programmazione concorrente che sfrutta i Threads risulta difficile, incline agli errori e difficilmente distribuibile.

Si può riassumere l'essenza di un Thread come un flusso di controllo sequenziale che condivide con gli altri Threads lo stesso spazio di memoria, diversamente dai processi. Questo significa che Threads indipendenti condividono variabili e stati in modo concorrente. In questo modo è facile che questi competino per leggere e/o scrivere simultaneamente sullo stesso spazio di memoria andando in contro a quella che viene definita come una race condition. Ad esempio consideriamo il caso in cui due Thread accedano alla stessa variabile condivisa. Entrambi assegnano un valore alla stessa variabile quindi, come è facile immaginare, il valore effettivamente memorizzato all'interno della variabile sarà quello dell'ultimo Thread che vi ha scritto. Questo esempio mostra come sia necessario un meccanismo di sincronizzazione per garantire il corretto funzionamento in ogni situazione di utilizzo ed in ogni momento.

3.1 Meccanismi di sincronizzazione: i locks

Le primitive utilizzate per la sincronizzazione sono i lock, i quali controllano l'accesso a sezioni critiche del codice. Esistono diversi tipi di meccanismi di lock, ognuno con un

diverso comportamento. I semafori sono una tipologia di lock che garantisce l'accesso alla sezione desiderata ad un numero massimo prestabilito di Threads. Quando il numero massimo viene raggiunto allora il semaforo blocca l'accesso alla sezione controllata. Quando si imposta il limite massimo a 1 allora si ottiene un semaforo ad accesso esclusivo chiamato Mutex.

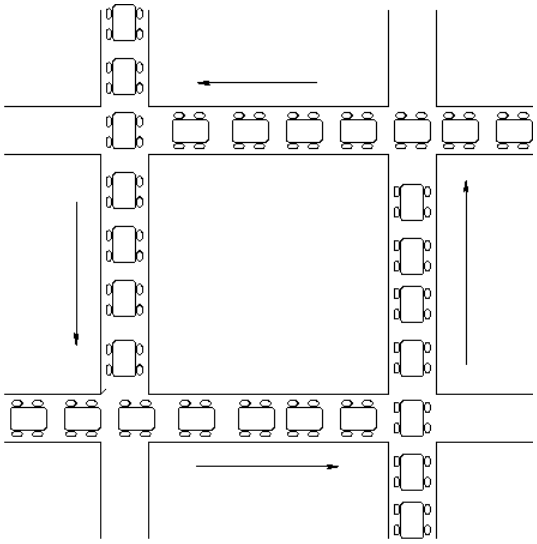


3.2 Le conseguenze dei lock

Esistono scenari in cui lock errati causano il malfunzionamento se non addirittura il blocco totale dell'intera applicazione. Prendiamo il semplice caso di due Threads che hanno necessità di acquisire il lock esclusivo simultaneo su due risorse distinte per poter compiere il proprio lavoro. Può accadere che entrambi i Threads rimangano bloccati in attesa del rilascio del lock altrui: questa situazione si verifica qualora ciascuno dei Thread riesca ad acquisire il lock su una risorsa ma non sull'altra perché nel frattempo è già stata occupata (T1 lock su R1 ma non su R2, T2 lock su R2 ma non su R1) In questa tipologia di scenario, denominata deadlock, entrambi i Threads rimangono bloccati senza speranza di sblocco: T1 attenderà all'infinito lo sblocco della risorsa R2 e T2 attenderà all'infinito lo sblocco della risorsa R1.

Questo scenario è conosciuto come "Il problema dei filosofi a cena", un esempio che concerne la sincronizzazione e viene descritto da Edsger Wybe Dijkstra nel 1965. In questo esempio 5 filosofi siedono attorno una tavola rotonda. Sulla tavola vi sono 5 piatti e 5 forchette. Le attività dei filosofi attorno la tavola consistono in pensare e mangiare. Quando un filosofo pensa non fa uso delle forchette perché non necessita di mangiare, altresì quando termina di pensare ed incomincia a mangiare ha bisogno di due forchette: quella di destra e quella di sinistra. Ogni volta che il filosofo tenta di mangiare procede

ad acquisire una forchetta alla volta: nel caso in cui riesca ad impossessarsi di entrambe allora mangia per un certo periodo di tempo ed infine rilascia entrambe le forchette e torna a pensare. Questo esempio porta a valutare le situazioni di deadlock che si possono creare qualora, per una malaugurata coincidenza, ciascun filosofo prenda la forchetta alla propria destra e rimanga in attesa della forchetta sinistra. Un'altra situazione possibile, che si evince da questo esempio, è quella di starvation in cui un particolare filosofo non riesca mai ad impossessarsi di entrambe le forchette e quindi muoia letteralmente di fame.



Esiste un'altra tipologia di scenario in cui due Threads tentano reciprocamente di acquisire due risorse in sequenza e necessitano di averle entrambe. Ognuno dei due thread riesce ad acquisirne una ma quando tenta di ottenere l'altra risorsa la richiesta viene respinta in quanto è già stata presa dall'altro Thread. Per cui la prima risorsa viene rilasciata ed il procedimento ricomincia. Se entrambi i Threads effettuano queste operazioni sempre nello stesso ordine e momento allora si ottiene una situazione di livelock nella quale i Threads non sono bloccati come nel deadlock ma non riescono ad acquisire le risorse necessarie e quindi a svolgere i loro compiti.

I casi di livelock sono risolvibili grazie a strategie atte a intervallare l'acquisizione delle risorse in caso di collisioni di richieste: ad esempio tentando nuovamente l'acquisizione delle risorse non immediatamente ma dopo un intervallo di attesa casuale. Per quanto riguarda i casi di deadlock, questi sono molto difficili da prevedere a causa del loro non determinismo. La probabilità di incorrere in queste casistiche aumenta al diminuire della grandezza dei componenti software regolati dal meccanismo di lock utilizzati all'interno dell'applicativo. Inoltre l'utilizzo massiccio dei meccanismi di sincronizzazione incrementa l'overhead globale costringendo lo scheduler a regolare frequentemente gli accessi. D'altro canto serializzare il flusso di controllo in piccoli componenti aumenta lo throughput permettendo a diversi Threads di lavorare in parallelo e quindi di sfruttare l'hardware multi-core a disposizione.

Dal punto di vista pratico esiste anche un ulteriore pericolo. Nel caso di grandi pezzi di codice, o addirittura framework interi, la modifica di una parte o l'aggiunta di una nuova funzionalità può inficiare sul comportamento atteso ed introdurre casi di deadlock, livelock o, più in generale, starvation. Anche quando si ha a che fare con componenti software esterni, dei quali non si possiede il codice sorgente, risulta impossibile essere certi del corretto funzionamento della gestione generale della sincronizzazione.

3.3 Concorrenza tramite Threads nei web servers

I web servers che utilizzano i Threads associano ad ogni singola richiesta un Thread il quale si occupa di gestire la richiesta associatagli. Questo semplice modello non richiede che ci sia coordinazione fra i Threads poiché ogni richiesta è indipendente dalle altre. Nel caso di operazioni che sfruttano intensamente la CPU questa soluzione risulta adatta in quanto, utilizzando i Threads, sfrutta il parallelismo. Diversamente, nel caso di operazioni che utilizzano I/O, il meccanismo di sincronizzazione dell'accesso alle risorse in comune è mascherato dal server che gestisce concorrentemente tutte le richieste. Questo approccio riesce ad incrementare lo throughput del server ma non velocizza l'esecuzione della singola richiesta. Per cui si rende necessario parallelizzare le operazioni all'interno della singola richiesta. Per fare ciò le operazioni devono essere indipendenti tra loro e la loro parallelizzazione non deve creare casi di deadlock o livelock visti in precedenza. Inoltre, per alti livelli di concorrenza, non si può trascurare l'overhead introdotto dal continuo cambio di contesto causato dall'utilizzo dei Threads nella parallelizzazione.

Tenendo conto di queste considerazioni ed a seconda dello scopo che si vuole raggiungere, possono essere messe in campo modelli di programmazione differenti.

Capitolo 4

Concorrenza e Attori

Il modello ad Attori può essere considerato, per certi versi, l'evoluzione del COOP (Concurrency Object Oriented Programming) poiché in primo luogo un Attore è un Oggetto ed in quanto tale incorpora dati e comportamento. Esistono diverse implementazioni del modello ad Attori ed alcune si differenziano per certi aspetti dalle caratteristiche base del modello originale che andremo ad analizzare a breve.

4.1 Il modello e le sue implementazioni

Nato nel 1973 da un lavoro di Carl Hewitt[1] questo modello si basa sul concetto che tutto sia un attore. L'attore è un'entità computazionale indipendente e concorrente che incorpora un proprio stato, inaccessibile dall'esterno, e comportamento. Queste caratteristiche derivano direttamente dall'Object Oriented Programming, modello sul quale gli Attori si fondano. Un attore può comunicare con altri attori unicamente tramite scambi di messaggi, creare attori a sua volta e decidere quale comportamento adottare per gestire il prossimo messaggio ricevuto. Queste tre caratteristiche sono il valore aggiunto che il modello ad Attori porta con se cambiando totalmente il paradigma di programmazione.

Inviare un messaggio è un'operazione atomica e asincrona: ciò significa che appena l'Attore invia il messaggio può continuare la propria esecuzione. Questo comportamento è molto diverso da una chiamata di metodo di un oggetto, la quale non termina finché il

[1] Carl Hewitt; Peter Bishop; Richard Steiger (1973). "A Universal Modular Actor Formalism for Artificial Intelligence". IJCAI.

metodo non ha terminato il proprio codice ed il flusso di controllo non è tornato all'oggetto che ha invocato il metodo. Un'altra implicazione di questo meccanismo di comunicazione è la gestione della risposta, se ne esiste una. Nel modello ad oggetti, l'eventuale valore di ritorno, risultato del metodo invocato, viene ricevuto insieme al flusso di controllo quando il metodo termina la propria esecuzione. Nel modello ad Attori il flusso di controllo e gli eventuali dati di ritorno seguono due cicli di vita differenti. Nel caso del flusso di controllo, questo torna immediatamente al chiamante. Al contrario, gli eventuali dati, risultato della computazione oppure informazioni provenienti dall'altro Attore come risposta al primo messaggio, vengono consegnati tramite un messaggio separato. Questo meccanismo è paragonabile alla gestione dei risultati nei modelli event-driven nei quali, per notificare al chiamante i dati, frutto dell'esecuzione del compito svolto, o più semplicemente la terminazione dell'operazione, il chiamato invoca una funzione di callback sul chiamante, denominata in questo modo appunto perché invocata al termine della gestione dell'evento.

La comunicazione tra attori avviene in modalità asincrona e non usa alcun intermediario. Ogni attore possiede una "mailbox", identificata univocamente, grazie alla quale riceve ed invia i messaggi. Si precisa che un Attore, per poter inviare un messaggio ad un altro Attore, ne deve conoscere il nome, un po' come accade se noi volessimo inviare una e-mail ad un nostro conoscente: questa operazione sarebbe impossibile senza conoscere l'indirizzo della casella si posta elettronica del destinatario. Inoltre un Attore è in grado di ricevere messaggi sulla propria casella di posta anche se viene spostato. Questo è ciò che succede a noi stessi quando lavoriamo con le nostre caselle di posta elettronica. Infatti non siamo obbligati a rimanere a casa per ricevere la posta ma possiamo spostarci dove vogliamo e continuare ad essere in grado di ricevere e-mail a patto di riuscire a collegarsi alla casella. Il fatto che un Attore riesca a ricevere messaggi anche cambiando dislocazione spaziale è garantito dal fatto che ogni Attore è identificato in modo univoco in modo indipendentemente dalla propria posizione.

Da notare che un attore può inviare a se stesso un messaggio, il quale verrà preso in carico come se fosse un messaggio proveniente da un altro Attore e quindi gestito. Il tempo che un messaggio impiega a giungere nella casella dell'attore destinatario è indefinito, non c'è garanzia sull'ordine di arrivo dei messaggi e addirittura non c'è garanzia di arrivo dei messaggi. Per cui può avvenire che un messaggio, partito prima di un altro, arrivi dopo questo nella mailbox del ricevente o non arrivi affatto. Questo comportamento è assimilabile ai pacchetti di dati che viaggiano attraverso la Rete. Infatti non è predicibile

l'ordine di arrivo dei pacchetti di dati poiché ognuno di loro può percorrere strade diverse per giungere alla destinazione. Addirittura i pacchetti possono non giungere mai, in questi casi esistono meccanismi di recupero dell'informazione o di re-invio del dato.

Un Attore processa i messaggi arrivati nella propria mailbox in modo sequenziale e senza rischi di incorrere in casi di race condition in quanto le operazioni di accodamento di un messaggio e prelevamento sono atomiche e lo stato interno dell'Attore è inaccessibile dall'esterno.

La prima implementazione del modello ad Attori fu il popolare linguaggio Erlang. Successivamente altri linguaggi incorporarono questo modello ma non nativamente, bensì attraverso librerie esterne che offrivano le funzionalità degli attori basandosi sull'ormai convenzionale multi-threading. Nel corso del tempo molteplici furono le implementazioni di questo modello ed alcune offrivano piccole modifiche all'architettura originale.

Quando un Attore invia un messaggio compie un'operazione non bloccante. Secondo il modello, non esiste garanzia che il messaggio arrivi a destinazione (es. l'attore non è più disponibile, problemi di comunicazione via rete, ecc...). Numerose implementazioni gestiscono questi casi e forniscono meccanismi per rispedire messaggi non confermati dal ricevente, ad esempio tramite timeout o conferme di ricezione. Inoltre non esiste garanzia di ordine negli arrivi ma molte implementazioni forniscono la garanzia che due messaggi, inviati dallo stesso attore ad un altro attore, arrivino nello stesso ordine in cui sono stati spediti.

Uno dei punti di forza del modello ad Attori è la facilità con cui si presta ad essere impiegato nei sistemi distribuiti grazie alla propria natura flessibile, concorrente e reattiva agli errori. Quando un Attore genera un altro Attore si crea una gerarchia a livelli. Nel caso in cui un attore vada in crash allora viene inviato un messaggio al padre, o supervisore, il quale può reagire in vari modi a seconda dell'implementazione come riavviare l'Attore, fermare altri Attori oppure inviando un messaggio di errore al proprio supervisore.

Poiché lo stato di un Attore non è accessibile a nessun altra entità se non a lui stesso e le operazioni di gestione della propria casella di posta sono atomiche, allora questi presupposti impediscono tutti quei casi di race condition che potevano verificarsi nel modello a Threads come deadlock, livelock e starvation. Seppur il modello scongiuri queste problematiche ciò non esclude che un cattivo utilizzo dell'architettura possa far scaturire situazioni di stallo. Ad esempio immaginiamo che un attore necessiti di aspettare un messaggio proveniente da un altro attore prima di compiere un determinato compito, ed

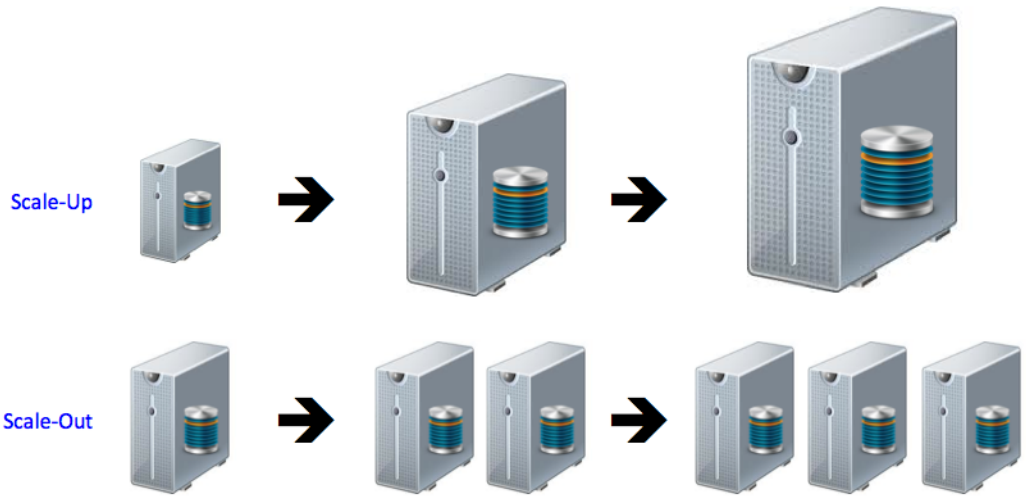
immaginiamo che l'altro attore abbia lo stesso comportamento. Questa mutua attesa genera un ciclo bloccante, un caso di deadlock, ed il risultato è che i due attori attenderanno all'infinito il messaggio dell'altro. In questo caso i modi per evitare il blocco sono due: utilizzare i timeout in modo da reagire se l'attore destinatario del messaggio non risponde entro un tempo prestabilito, oppure correggere il design dell'applicazione in modo appropriato. D'altro canto questa situazione può accadere anche se un messaggio non arriva mai al destinatario. In questo caso è molto importante l'implementazione del modello che si è scelto di utilizzare per far in modo di evitare il blocco del ricevente.

4.2 Perché scegliere il modello ad Attori e differenze con il modello a Threads

Quando un sistema software è altamente concorrente e fattori come l'affidabilità, la scalabilità e la tolleranza ai guasti diventano requisiti imprescindibili allora la programmazione ad Attori fornisce numerosi pregi rispetto al più "classico" approccio a Threads.

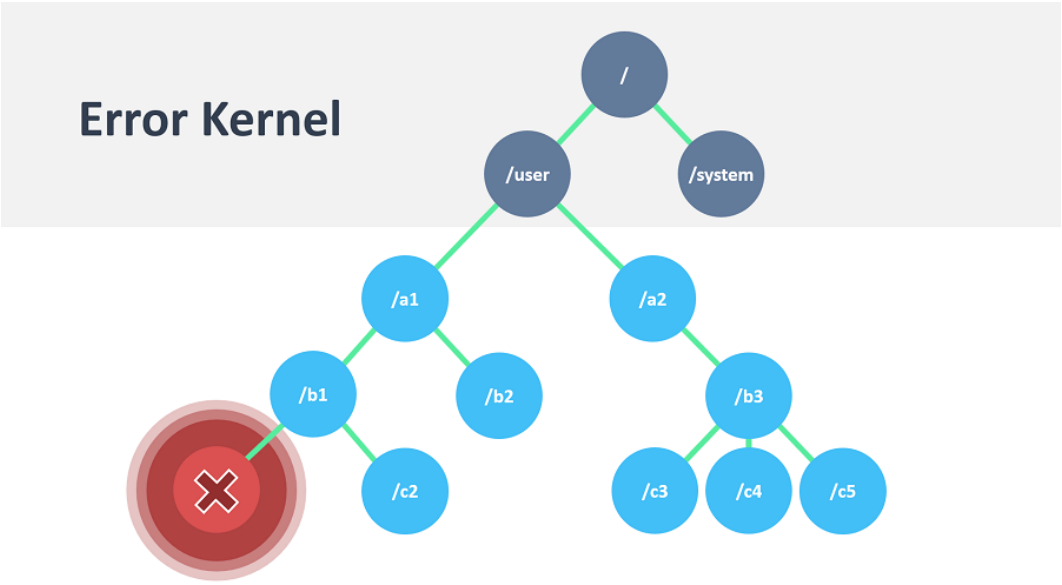
Una delle prime caratteristiche che differenziano i due modelli sono l'utilizzo delle risorse. Un Attore impiega meno risorse rispetto ad un Thread, inoltre la creazione e distruzione di un Attore introduce un overhead considerevolmente minore rispetto al corrispondente Thread.

La scalabilità è un altro fattore di rilievo. Un sistema sviluppato ad Attori può facilmente fare uso di più o meno risorse fisiche a seconda del carico a cui il sistema è soggetto. Ad esempio, in caso di forte stress, alcuni Attori possono essere spostati su macchine dedicate e continuare il proprio lavoro sfruttando le nuove risorse a disposizione. In questo modo si ottiene un sistema dinamico, che utilizza effettivamente solo le risorse di cui necessita. Ad esempio un sistema così costituito potrà inizialmente utilizzare un ambiente relativamente modesto per quanto riguarda potenza di calcolo e storage per poi ampliarsi al bisogno in modo totalmente trasparente, e senza interruzione di servizio (scale-out). Diversa sarebbe la questione per un sistema sviluppato a Threads per il quale si renderebbe necessario cambiare totalmente ambiente, solitamente con una interruzione di servizio, in favore di una macchina più potente e con maggiori risorse a disposizione



(scale-up).

La facilità di spostamento degli Attori incide anche sull'affidabilità di questi sistemi. Nel caso in cui parte di un sistema di questo tipo incorresse in un qualche genere di malfunzionamento allora sarebbe facile per l'Attore supervisore reagire all'inconveniente, ad esempio, creando dei nuovi Attori, su una qualsiasi macchina a disposizione, e garantendo la continuità di funzionamento del sistema. Al contrario, in un sistema sviluppato a Threads, si renderebbe necessario avviare un complesso meccanismo di recovery con lo scopo di ricreare la stessa situazione che era presente prima del crash del server interessato.



4.3 L'implementazione del modello by Akka

Esistono varie implementazioni del modello ad Attori, ognuna con caratteristiche differenti, alcune ricalcano fedelmente il modello, altre lo rielaborano sotto alcuni aspetti.

Ho scelto di approfondire lo studio del modello ad Attori impiegando la libreria Akka.Net perché è un progetto open source nato nel 2009, sviluppato da una folta community, costantemente aggiornato e in versione stabile. Inoltre questa libreria promette alti livelli di performance: 50 milioni si messaggi al secondo (su singola macchina) e un utilizzo di 1 GB per circa 2,5 milioni di Attori.

Essendo un'implementazione del modello ad Attori questa libreria offre tutte le caratteristiche proprie del modello:

- un'astrazione ad alto livello per la concorrenza ed il parallelismo
- un modello di programmazione event-driven altamente performante, non bloccante ed asincrono
- l'applicazione della semantica "let-it-crash" propria del modello ad attori con relativo recupero delle situazioni di crash
- la fedele applicazione del modello di comunicazione proprio degli Attori, con comunicazione unicamente tramite scambio di messaggi in modalità asincrona.

4.3.1 Creazione ed utilizzo di un Attore

Akka.Net poggia sul framework .Net ed in particolare sul linguaggio C# il quale è Object Oriented. Per questo motivo, in base al modello ad Oggetti in cui ogni cosa è un Oggetto, non solo gli Attori sono Oggetti ma anche i messaggi che gli Attori si scambiano lo sono. Di seguito si mostra un semplice programma che mostra come creare un Attore ed utilizzarlo.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using Akka;
using Akka.Actor;

namespace ConsoleApplication11
{
    public class Greet
    {
        public Greet(string who)
        {
            Who = who;
        }
        public string Who { get;private set; }
    }

    public class GreetingActor : ReceiveActor
    {
        public GreetingActor()
        {
            Receive<Greet>(greet =>
               Console.WriteLine("Hello {0}", greet.Who));
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            // Create a new actor system (a container for your actors)
            var system = ActorSystem.Create("MySystem");

            // Create your actor and get a reference to it.
            // This will be an "ActorRef", which is not a
            // reference to the actual actor instance
            // but rather a client or proxy to it.
            var greeter = system.ActorOf<GreetingActor>("greeter");

            // Send a message to the actor
            greeter.Tell(new Greet("World"));

            // This prevents the app from exiting
            // before the async work is done
            Console.ReadLine();
        }
    }
}
```

Nell'esempio sopra riportato sono state create due classi: Greet e GreetingActor. La prima è il messaggio che verrà inviato all'Attore, la seconda è l'Attore stesso. Per quanto riguarda il messaggio è da notare che il suo stato interno non è modificabile dall'esterno se non alla sua creazione.

In Akka esistono due tipi di Attori:

- UntypedActor: Attore "senza tipo" nel senso che riceve messaggi non tipizzati a priori e deve incorporare una logica applicativa esplicita basata sul tipo di messaggio

ricevuto

- ReceiveActor: Attore più strutturato che, per tipo di messaggio, definisce il comportamento da adottare

Nel costruttore di ogni Attore, che estende la classe base ReceiveActor, vengono definiti i comportamenti che l'Attore deve assumere al sopraggiungere di un tipo di messaggio.

```
Receive<Greet>(greet =>
                    Console.WriteLine("Hello {0}", greet.Who));
```

Questo metodo definisce il comportamento che il GreetingActor dovrà assumere quando elaborerà un messaggio di tipo Greet. In questo particolare esempio il metodo è tipizzato con il tipo di messaggio Greet, e come parametro di input è stata data una lambda expression che descrive il comportamento da adottare, cioè la stampa della frase di saluto riportando la parola contenuta nel messaggio.

Una volta creato l'Attore ed il messaggio passiamo all'utilizzo di questi.

```
var system = ActorSystem.Create("MySystem");
```

Invocando il metodo statico Create sulla classe ActorSystem si crea un nuovo ActorSystem, cioè un nuovo container per gli Attori. L'ActorSystem è un componente centrale nell'implementazione di Akka in quanto fornisce tutte le primitive di base per poter lavorare con gli Attori come ad esempio la loro creazione e la configurazione e gestione del sistema degli Attori.

```
var greeter = system.ActorOf<GreetingActor>("greeter");
```

Invocando il metodo ActorOf sull'istanza creata dell'ActorSystem e specificando il tipo di Attore che si vuole creare e l'identificativo da assegnargli si ottiene il riferimento all'Attore creato. L'implementazione di Akka utilizza un pattern di tipo Proxy per la creazione degli Attori: questo significa che, quando si crea un attore o quando lo si richiama, non si otterrà mai l'oggetto richiesto ma piuttosto un'interfaccia di tipo IActorRef che permette di invocare i metodi comuni a tutti gli Attori.

```
greeter.Tell(new Greet("World"));
```

Invocando il metodo Tell su un Attore si compie un invio asincrono del messaggio dato in input al metodo stesso. In questo esempio si invoca il metodo Tell sul GreetingActor

appena creato inviandogli, in modo asincrono, un nuovo messaggio contenente la parola "World". Ci aspettiamo che l'Attore, non appena processi questo messaggio, scriva a console l'intera frase "Hello World".

4.3.2 Affidabilità di recapito dei messaggi

Nel modello ad Attori la dislocazione spaziale delle entità è completamente trasparente. Questa affermazione rimane vera per l'implementazione di Akka in cui la modalità di invio di un messaggio ad un Attore remoto è identica anche per un Attore locale. Naturalmente i passaggi che portano alla consegna di un messaggio ad un Attore che si trova su una macchina diversa da quella del mittente sono diversi rispetto alla modalità di consegna locale. La consegna di un messaggio ad un destinatario remoto richiede controlli aggiuntivi più stringenti sul messaggio (es. la dimensione), la sua serializzazione ed invio sulla rete.

L'implementazione che Akka ha svolto per quanto riguarda la comunicazione tra Attori si basa su due regole delle quali la prima è l'applicazione della modalità at-most-once delivery, cioè nessuna garanzia di consegna dei messaggi.

L'utilizzo di questa modalità si trova in molte implementazioni del modello. La scelta fatta da Akka è quella più semplice, performante e meglio gestibile poiché non richiede ulteriori logiche e mantenimento di stati di invio una volta inviato il messaggio. Le altre due possibili modalità di gestione della comunicazione, at-least-once e exactly-once, sono rispettivamente una più onerosa dell'altra. La modalità at-least-once richiede che almeno una copia del messaggio raggiunga il destinatario e per ottenere questo comportamento si deve impiegare un vero e proprio protocollo di comunicazione con ACK di consegna e gestione di re-invio in caso di non recapito. La modalità exactly-once risulta ancora più onerosa in quanto, oltre ad implementare il protocollo del meccanismo precedente, necessita di applicare una strategia di filtraggio dei messaggi per eliminare i duplicati. In ogni modo, la flessibilità del modello ad Attori permette al bisogno di implementare i due protocolli di comunicazione appena descritti utilizzando le primitive offerte dal modello. Akka ha deciso questo approccio in modo da garantire le massime performance lasciando agli utilizzatori, qualora lo desiderino, la facoltà di implementare i protocolli di comunicazione più adatti alle proprie esigenze.

4.3.3 Ordine di arrivo dei messaggi

La seconda regola impiegata da Akka per la comunicazione tra Attori è il mantenimento dell'ordine dei messaggi per coppia di interlocutori. Questa regola impone che i messaggi inviati da un Attore A1 ad un Attore A2 arrivino nel medesimo ordine in cui sono stati inviati. Questo comportamento non vieta che se l'Attore A2 stia ricevendo nello stesso momento messaggi da un'Attore A3 allora i messaggi di A1 e A3 non possano giungere a A2 intervallati tra loro. Qualora un comportamento FIFO come quello appena descritto non sia ritenuto adatto è possibile utilizzare un altro tipo di Mailbox implementata da Akka: UnboundedPriorityMailbox. Questo tipo di casella di posta applica una logica arbitraria basata sulla priorità dei messaggi che giungono nella casella. In questo modo, attribuendo diverse priorità ai messaggi, questi vengono elaborati con un ordine diverso rispetto il loro arrivo. D'altro canto, l'utilizzo di questo tipo di Mailbox può causare situazioni di starvation in quanto i messaggi meno prioritari rimangono sempre in fondo la coda e potrebbero essere mai presi in carico dall'Attore nel caso in cui arrivassero continuamente messaggi di priorità maggiore.

4.3.4 Problematiche e pattern di creazione

Un caso particolare ed interessante da sottolineare è la creazione di un Attore. In Akka la creazione di un Attore può essere vista come l'invio di un messaggio. Infatti quando un Attore crea un altro Attore, il padre ottiene subito un riferimento all'Attore figlio (interfaccia IActorRef) ma non è garantito che tale Attore sia già stato creato. Ciò significa che la creazione di un Attore è un'operazione asincrona non bloccante, come l'invio di un messaggio. Per tanto esiste la possibilità di incorrere in uno scenario di questo tipo:

1. A1 crea A2
2. A1 invia un messaggio ad A3 contenente il riferimento ad A2
3. A3 riceve il messaggio di A1
4. A3 invia un messaggio ad A2
5. A2 ancora non esiste! Il messaggio finisce tra le DeadLetters

Per evitare di incorrere in problematiche di questo tipo esistono due alternative:

1. A1 invia un messaggio ad A3 istruendolo sulla creazione di A2. In questo modo A3 crea A2 e successivamente gli invia un messaggio. In questo modo, poiché l'ordine dei messaggi tra due interlocutori viene garantito, A2 riceverà sicuramente il messaggio di A3.

2. A1 crea A2 e gli invia immediatamente un messaggio di echo. A2, una volta creato, riceve il messaggio di A1 e risponde all'echo. A1 riceve l'echo di A2 e, sicuro della sua esistenza, invia il suo riferimento ad A3. A3 riceve il messaggio di A1 e contatta a sua volta A2.

Il primo di questi due punti è sicuramente il più semplice ma ci sono motivi per cui si renda necessario intraprendere la strada indicata dal secondo punto: ad esempio A1 potrebbe essere il supervisore di A2 e A3 e non si vuole modificare la gerarchia esistente.

L'utilizzo del pattern al punto 2, unito con una Mailbox prioritaria che assegna massima priorità ai messaggi di echo, potrebbe fornire un valido meccanismo per affrontare scenari in cui si necessita creare Attori e successivamente passarne il riferimento.

4.3.5 Nati per essere distribuiti

"Everything in Akka is designed to work in a distributed setting"

Come tutte le implementazioni del modello ad Attori anche Akka fornisce delle metodologie per la comunicazione remota degli attori. In Akka è disarmante la facilità con cui si è in grado di instaurare una comunicazione remota tra due Attori.

Di seguito un esempio:

Server:

```
var config = ConfigurationFactory.ParseString(@"
akka {
    actor {
        provider = ""Akka.Remote.RemoteActorRefProvider, Akka.Remote""
    }

    remote {
        helios.tcp {
            port = 8080
            hostname = localhost
        }
    }
}
");

using (ActorSystem system = ActorSystem.Create("MyServer", config))
{
    system.ActorOf<GreetingActor>("greeter");
    Console.ReadKey();
}
```

Client:

```
var config = ConfigurationFactory.ParseString(@"
akka {
    actor {
        provider = ""Akka.Remote.RemoteActorRefProvider, Akka.Remote""
    }
    remote {
        helios.tcp {
            port = 8090
            hostname = localhost
        }
    }
}
");

using(var system = ActorSystem.Create("MyClient", config))
{
    //get a reference to the remote actor
    var greeter = system
        .ActorSelection("akka.tcp://MyServer@localhost:8080/user/greeter");
    //send a message to the remote actor
    greeter.Tell(new Greet { Who = "Roger" });

    Console.ReadLine();
}
```

Come si può evincere dal codice sopra riportato, è sufficiente conoscere l'identificativo di un attore per poter comunicare con lui ed inviargli messaggi, qualsiasi sia la sua localizzazione. Nell'esempio sopra riportato sono stati creati due ActorSystem sulla stessa macchina fisica che sono in ascolto su porte diverse (8090 per il container utilizzato e 8080 per il container utilizzante).

```
var greeter = system
        .ActorSelection("akka.tcp://MyServer@localhost:8080/user/greeter");
```

Con il metodo ActorSelection, invocato sull'ActorSystem locale, è possibile ottenere il riferimento all'Attore remoto specificando un ActorPath valido.

L'ActorPath è una stringa che identifica in modo univoco un Attore. Questa è formata da diverse parti:

- Protocol: il protocollo di rete utilizzato per la comunicazione
- ActorSystem: il nome dell'ActorSystem remoto
- Address: indirizzo di rete (con porta) dell'ActorSystem
- Path: percorso relativo dell'Attore

4.4 Conclusioni

Come abbiamo analizzato, il modello ad Attori offre by-design aspetti che eliminano i principali problemi della programmazione concorrente quali starvation, deadlock e livelock, problemi che nel modello a Thread sono ben presenti e quindi da affrontare.

Inoltre il modello ad Attori è notevolmente più adatto ad avere una conformazione distribuita: infatti grazie alla comunicazione a messaggi un Attore non è legato alla posizione in cui è stato creato ma è in grado di spostarsi spazialmente continuando a ricevere i messaggi nella propria mailbox.

L'implementazione di Akka differisce in alcuni punti dal modello standard. Queste differenze sono dettate dalle filosofie adottate:

- let it crash: semplicità di gestione del ciclo di vita degli Attori. Se un Attore incorre in un errore si gestisce l'accaduto informando il supervisore dell'Attore stesso
- fire-and-forget: semplicità di gestione della comunicazione tra gli Attori. Non esiste garanzia di consegna dei messaggi (at-most-once delivery)

- design once, deploy any way you wish: per Akka l'implementazione del meccanismo di comunicazione tra gli Attori non deve basarsi sul contesto particolare in cui si trovano. La comunicazione tra Attori può avvenire all'interno della stessa macchina fisica così come tra i due punti più lontani del pianeta. Per questi motivi vengono promossi i concetti di "distribuited by default" e "context awareness"

Capitolo 5

Pattern di comunicazione “Router” per il modello ad Attori

Quando un sistema applicativo, basato sul modello ad Attori, si evolve spesso aumenta la complessità intrinseca della logica applicata. Nel modello ad Attori la comunicazione è parte fondamentale dell'architettura e quindi la sua complessità cresce di pari passo con quella dell'applicazione che ne fa uso: ecco perché il pericolo di “spaghetti coding” è sempre presente anche in un modello come questo.

Precedentemente abbiamo visto che un Attore è assimilabile, dal punto di vista interno, ad un Oggetto. Per tanto molti Pattern che si applicano al modello Object Oriented possono essere utilizzati, talvolta con qualche modifica, all'interno degli Attori.

Per quanto riguarda la comunicazione, il modello ad Attori necessita di Pattern applicabili al message passing in quanto è lo scambio di messaggi l'unico metodo di comunicazione utilizzato. Un Pattern già implementato da Akka è il Router.

In Akka un Router è uno speciale Attore il cui scopo è quello di re-direzionare i messaggi verso altri Attori chiamati Routees. Esistono diverse implementazioni del Router, ognuno dei quali applica diverse strategie. Esistono due tipologie di Routers:

- Pools: sono Routers i quali creano loro stessi gli Attori Routees. In questo modo il loro supervisore sarà il Router che li ha creati e quindi lui sarà il primo a ricevere le eventuali notifiche di crash degli Attori figli.

- Groups: sono Routers che utilizzano Attori affidati dall'esterno. In questo modo il Router non è il supervisore ma solo l'utilizzatore degli Attori workers.

Per impostazione predefinita i Routers Pools utilizzano la strategia “Escalate” quando ricevono una eccezione. Questa strategia fa in modo che l’eccezione salga fino l’Attore padre del Router il quale deciderà se riavviare il Router e tutti i Routees. I Routers Groups sono semplici utilizzatori degli Attori assegnati dall’esterno e in caso di anomalie sarà il loro supervisore ad essere notificato e non il Router che rimarrà all’oscuro dell’accaduto.

Di seguito verranno analizzate alcune strategie di routing che Akka offre già implementate e pronte per essere utilizzate.

5.1 RoundRobin

I Routers che implementano questa strategia inoltrano i messaggi agli Attori Routees utilizzando l’ordinamento RoundRobin che non utilizza alcun sistema di priorità e garantisce semplicità ed assenza di starvation.

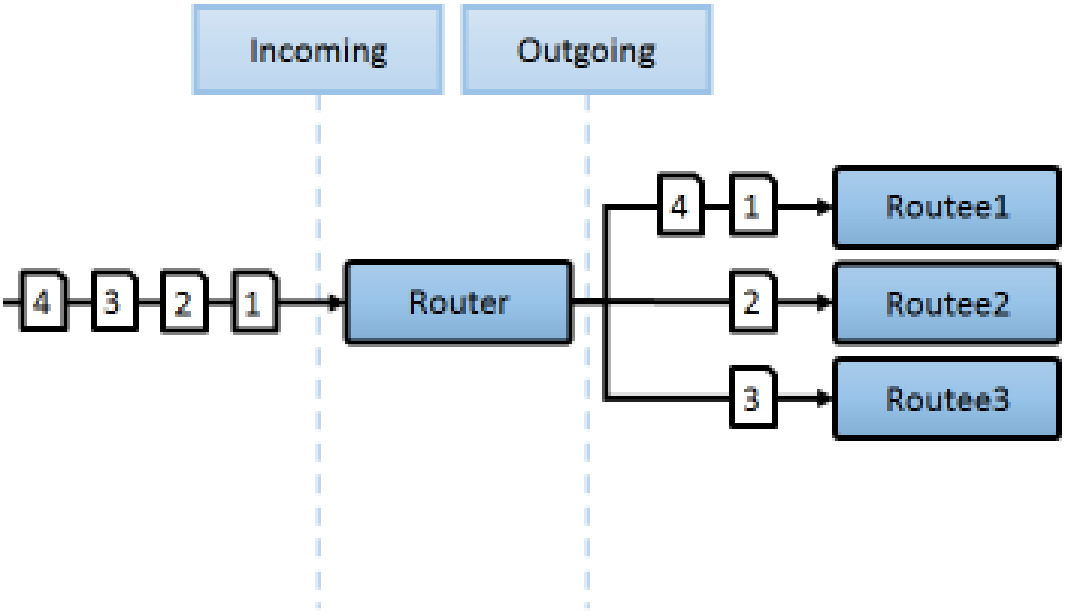


Figura 5.1: Router che utilizza strategia RoundRobin

Per questa strategia esistono due modalità di creazione del Router: RoundRobinPool e RoundRobinGroup.

Per creare un Router di tipo Pool allora il codice è il seguente:

```
var router =
```

```
system.ActorOf(Props.Create<Worker>()
.WithRouter(new RoundRobinPool(5)), "some-pool");
```

Mentre per creare un Router di tipo Group l’implementazione risulta la seguente:

```
var workers = new [] {
"/user/workers/w1", "/user/workers/w3", "/user/workers/w3" ù
};
var router =
system.ActorOf(Props.Empty
.WithRouter(new RoundRobinGroup(workers)), "some-group");
```

5.2 Broadcast

I Routers che implementano questa strategia inoltrano gli stessi messaggi a tutti gli Attori.

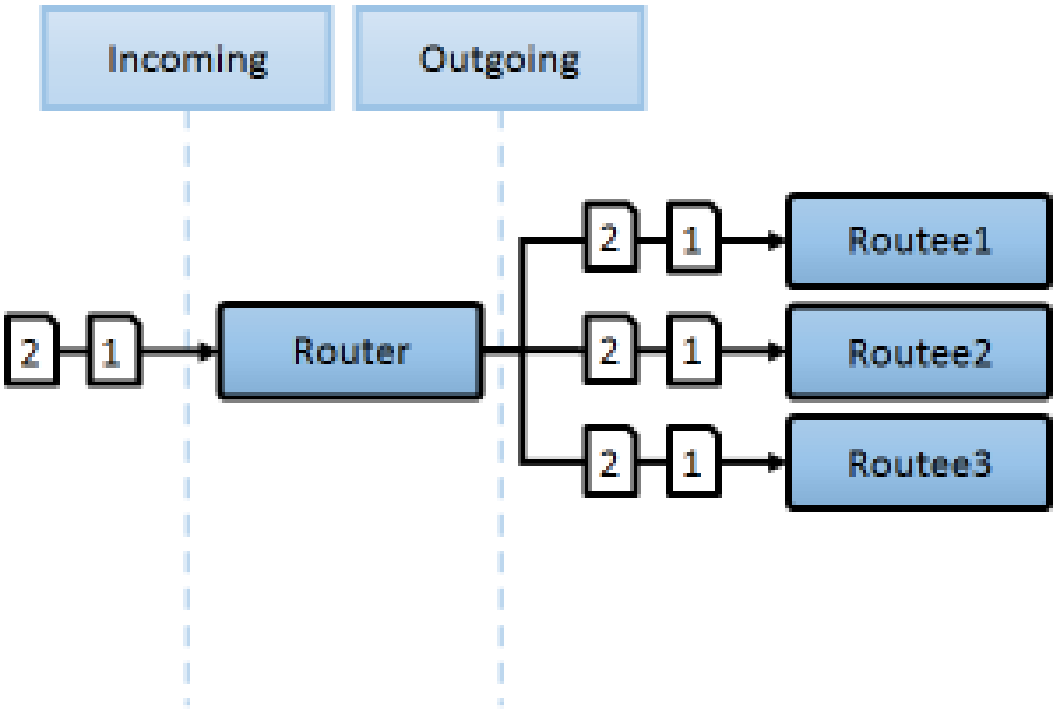


Figura 5.2: Router che utilizza strategia Broadcast

Per questa strategia esistono due modalità di creazione del Router: BroadcastPool e BroadcastGroup.

Per creare un Router di tipo Pool allora il codice è il seguente:

```
var router =
system.ActorOf(Props.Create<Worker>()
.WithRouter(new BroadcastPool(5)), "some-pool");
```

Mentre per creare un Router di tipo Group l’implementazione risulta la seguente:

```
var workers = new [] {
"/user/workers/w1", "/user/workers/w3", "/user/workers/w3" ù
};
var router =
system.ActorOf(Props.Empty
.WithRouter(new BroadcastGroup(workers)), "some-group");
```

5.3 Random

I Routers che implementano questa strategia inoltrano i messaggi agli Attori con ordine casuale.

Per questa strategia esistono due modalità di creazione del Router: RandomPool e RandomGroup.

Per creare un Router di tipo Pool allora il codice è il seguente:

```
var router =
system.ActorOf(Props.Create<Worker>()
.WithRouter(new RandomPool(5)), "some-pool");
```

Mentre per creare un Router di tipo Group l’implementazione risulta la seguente:

```
var workers = new [] {
"/user/workers/w1", "/user/workers/w3", "/user/workers/w3"
};
var router =
system.ActorOf(Props.Empty.
WithRouter(new RandomGroup(workers)), "some-group");
```

5.4 ConsistentHashing

I Routers che implementano questa strategia utilizzano un algoritmo di hashing consistente per selezionare l’Attore al quale inoltrare il messaggio. Lo scopo di questa strategia è quella di inviare i messaggi con la stessa chiave sempre allo stesso Attore.

Questo tipo di strategia è molto utile quando si ha a che fare con una situazione assimilabile a Domain Driven Design in cui ogni messaggio risulta essere un’azione applicata ad un dominio di appartenenza.

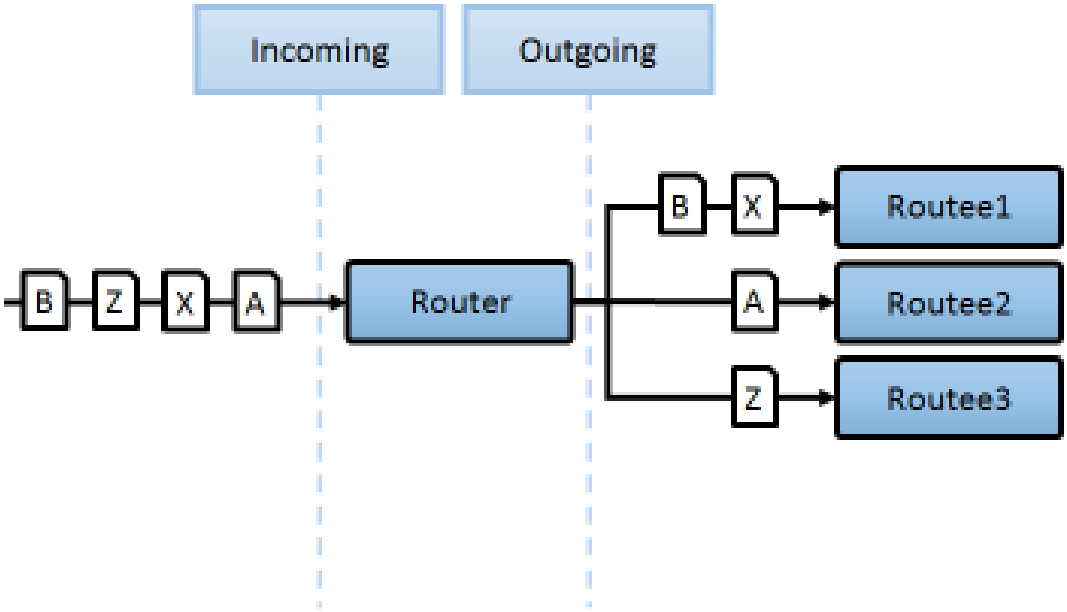


Figura 5.3: Router che utilizza strategia ConsistentHashing

Ad esempio immaginiamo che un Router debba processare messaggi che operano azioni sui clienti dell’applicativo. Ogni messaggio contiene l’id del cliente, il tipo di comando da applicare ed altre informazioni utili per espletare il comando. L’utilizzo dell’hashing consistente permette al Router di inoltrare quei messaggi applicabili ad un particolare cliente sempre al medesimo Attore. In questo modo è possibile evitare situazioni di race conditions tra gli Attori che applicano i comandi dei relativi clienti. Immaginiamo infatti di applicare un Round Robin Router al problema analizzato al posto di un Consistent Hashing Router come si vede in Figura 7. La strategia Round Robin avrebbe assegnato i messaggi ai tre Attori in modo sequenziale e quindi i messaggi del cliente 123 sarebbero stati processati da Attori diversi. Poiché non è predicibile a priori quanto tempo può impiegare un messaggio a giungere a destinazione, e tanto meno non c’è garanzia di consegna, allora i comandi “Cancel” e “Create” potrebbero essere eseguiti indistintamente uno primo dell’altro causando un comportamento non prevedibile e addirittura errato nel caso in cui venisse eseguito prima il comando “Cancel” del comando “Create”.

Per creare un Consistent Hashing Router, definendo il metodo per l’hash mapping, il codice è il seguente:

```
var chp = new ConsistentHashingPool(5).WithHashMapping(o => {
if (o is IHasCustomKey) return ((IHasCustomKey)o).Key;
```

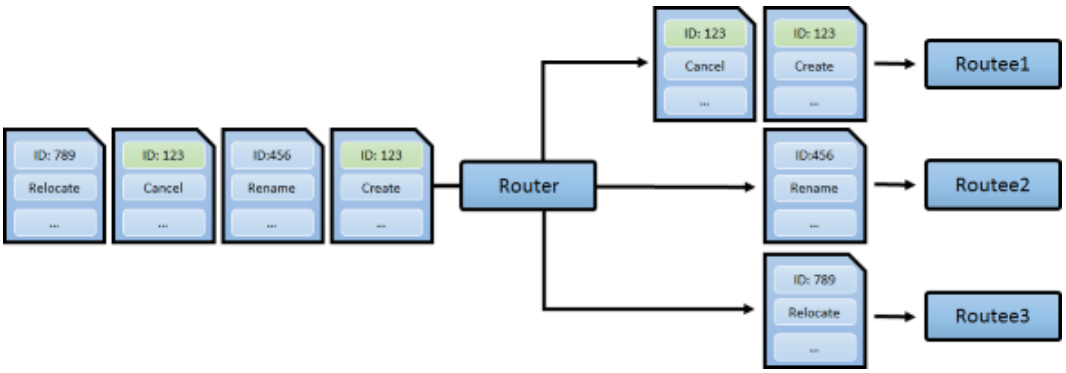


Figura 5.4: Esempio di strategia ConsistentHashing

```
    return null;
});
var router = system.ActorOf(Props.Create<Worker>()
.WithRouter(chp), "some-pool");
```

Il codice appena visto non è l’unico modo per implementare questa strategia ma permette totale trasparenza verso il mittente il quale è ignaro della strategia applicata al proprio messaggio. Il secondo modo è quello di far estendere al messaggio l’interfaccia IConsistentHashable e quindi di implementare la property ConsistentHashKey in modo che restituisca l’hash desiderato. Per fare in modo che Akka applichi questa tecnica bisogna omettere il delegato hash mapping visto precedentemente. Questa tecnica obbliga il mittente a costruire il messaggio in una certa maniera e quindi lo fa partecipe della specifica implementazione.

Per creare un Router di tipo Group l’implementazione risulta la seguente:

```
var workers = new [] {
"/user/workers/w1", "/user/workers/w3", "/user/workers/w3"
};
var router =
system.ActorOf(Props.Empty.WithRouter(new ConsistentHashingGroup(workers)), "some-group");
```

5.5 TailChopping

I Routers che implementano questa strategia inoltrano un messaggio ad un Attore scelto in modo casuale e poi attendono la risposta per una certa quantità di tempo. Se l’Attore non risponde entro il lasso di tempo stabilito allora il Router ripete l’azione di inviare il

messaggio ad un altro Attore casuale e di attendere. Il Router termina la propria operazione quando un qualsiasi Attore risponde al messaggio inviato e quindi lo inoltra al mittente originale. Se non viene ricevuta alcuna risposta entro un ritardo stabilito allora il Router invia al supervisore una segnalazione di fallimento.

Lo scopo di questa strategia è quello di ridurre la latenza delle operazioni eseguendo operazioni ridondanti su molteplici Attori nel tentativo che uno di questi sia più veloce di quello precedente. Ad esempio immaginiamo che lo stesso tipo di Attore “worker” sia presente su diversi nodi della rete. La strategia TailChopping è utile per individuare il nodo che risponde in tempo utile e quindi con meno carico di lavoro.

Per questa strategia esistono due modalità di creazione del Router: TailChoppingPool e TailChoppingGroup.

Per creare un Router di tipo Pool allora il codice è il seguente:

```
var within = TimeSpan.FromSeconds(10);
var interval = TimeSpan.FromMilliseconds(20);
var router =
system.ActorOf(Props.Create<Worker>()
.WithRouter(new TailChoppingPool(5, within, interval)), "some-pool");
```

Mentre per creare un Router di tipo Group l’implementazione risulta la seguente:

```
var workers = new [] {
"/user/workers/w1", "/user/workers/w3", "/user/workers/w3"
};
var within = TimeSpan.FromSeconds(10);
var interval = TimeSpan.FromMilliseconds(20);
var router =
system.ActorOf(Props.Empty
.WithRouter(new TailChoppingGroup(workers, within, interval)),
"some-group");
```

Il parametro “within” indica il tempo totale che il Router attenderà per ricevere una risposta da un qualsiasi Attore prima di generare un messaggio di fallimento ed inoltrarlo al proprio supervisore. Il parametro “interval” specifica quanto tempo il Router attenderà prima di effettuare l’invio del messaggio ad un altro Attore.

5.6 ScatterGatherFirstCompleted

Questa strategia di routing riprende l’idea di quella precedentemente analizzata con delle differenze. In questa modalità il Router invia simultaneamente lo stesso messaggio a tutti

i Routees e la prima risposta che riceve la inoltra al mittente originale, scartando tutte le successive risposte ottenute. Questo approccio si mostra utile in quei scenari in cui non importata da chi provenga la risposta al messaggio: ad esempio ottenere un nodo on-line qualsiasi.

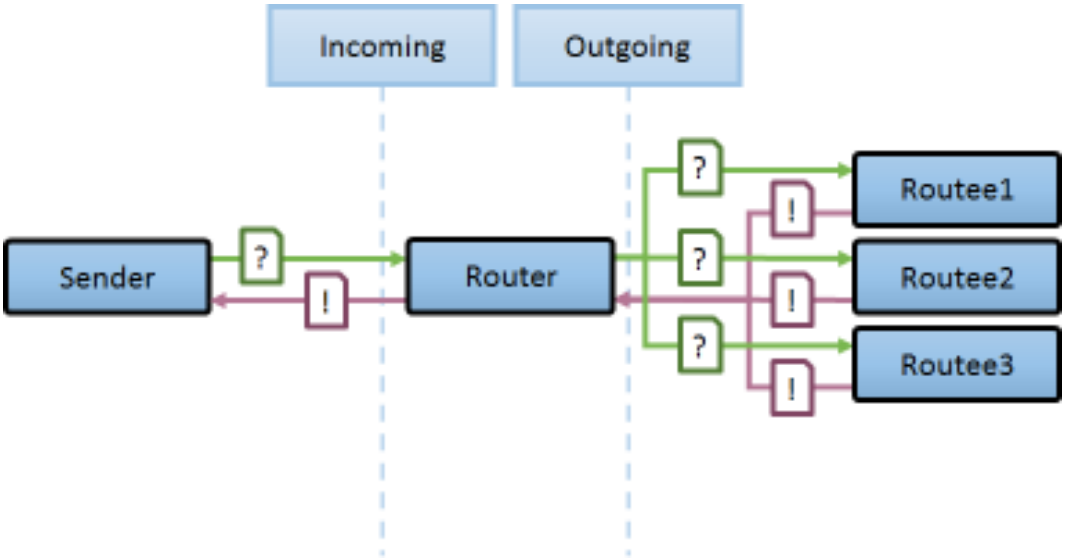


Figura 5.5: Router che utilizza strategia ScatterGatherFirstCompleted

Per questa strategia esistono due modalità di creazione del Router: ScatterGatherFirst-CompletedPool e ScatterGatherFirstCompletedRouter.

Per creare un Router di tipo Pool allora il codice è il seguente:

```
var within = TimeSpan.FromSeconds(10);
var router =
system.ActorOf(Props.Create<Worker>()
.WithRouter(new ScatterGatherFirstCompletedPool(5, within)),
"some-pool");
```

Mentre per creare un Router di tipo Group l’implementazione risulta la seguente:

```
var workers = new [] {
"/user/workers/w1", "/user/workers/w3", "/user/workers/w3"
};
var within = TimeSpan.FromSeconds(10);
var router =
system.ActorOf(Props.Empty
.WithRouter(new ScatterGatherFirstCompletedGroup(workers, within)),
"some-group");
```

Il parametro “within” indica il tempo totale che il Router attenderà per ricevere una risposta da un qualsiasi Attore prima di generare un messaggio di fallimento ed inoltrarlo al proprio supervisore.

5.7 SmallestMailbox

Questa strategia di routing applica un bilanciamento tra i Routees associati al Router e tenta di inoltrare un messaggio all’Attore con meno carico all’interno del pool. Il Router sceglie l’Attore a cui inoltrare il messaggio basandosi via via sui seguenti passi:

1. sceglie un Attore che non stia processando alcun messaggio (idle) e che abbia la mail box vuota
2. sceglie un’Attore che abbia la mail box vuota
3. sceglie l’Attore con il minor numero di messaggi pendenti
4. sceglie un Attore qualsiasi (gli Attori remoti hanno priorità minore)

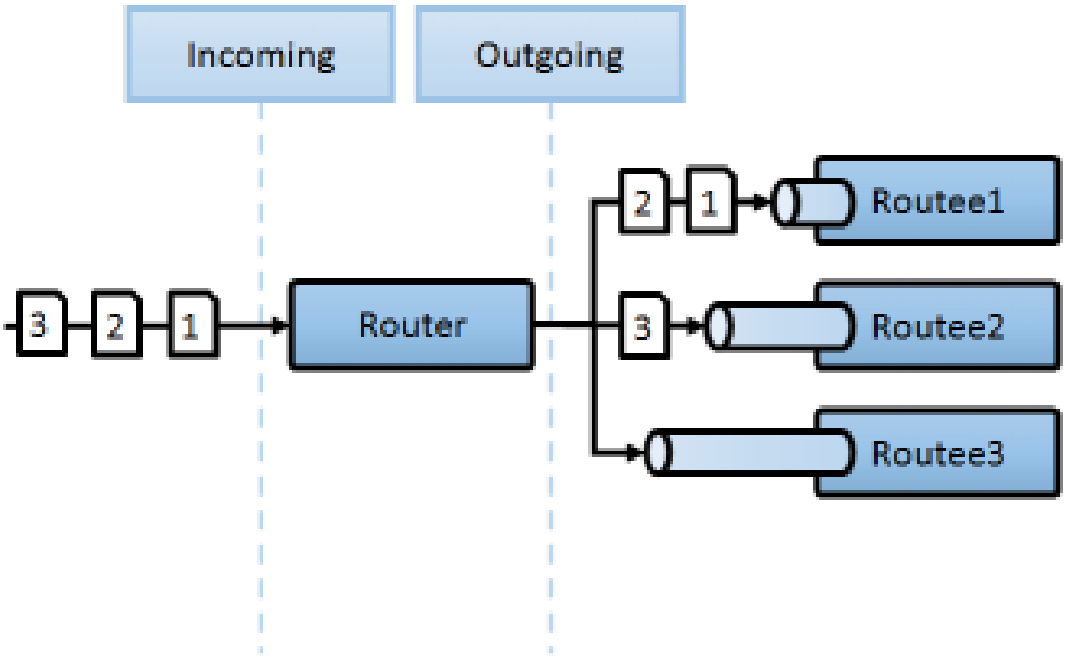


Figura 5.6: Router che utilizza strategia SmallestMailbox

Per questa strategia esiste la sola modalità Pool: SmallestMailboxPool.

Per creare un Router di tipo Pool allora il codice è il seguente:

```
var router =
system.ActorOf(Props.Create<Worker>()
.WithRouter(new SmallestMailboxPool(5)), "some-pool");
```

5.8 Considerazioni generali

Ogni strategia di routing vista fino ad ora può essere implementata via codice (come in tutti gli esempi visti) o tramite un file di configurazione successivamente caricato con il comando FromConfig.Instance.

Un Router di tipo Pool è in grado di gestire dinamicamente il proprio set di Attori incrementando o decrementando il numero di questi al bisogno. Questo comportamento è possibile aggiungendo al file di configurazione del Router la sezione “resizer”. In questa sezione è possibile specificare i seguenti parametri per ottenere il comportamento desiderato:

- enabled: abilita il resizer del pool di Attori.
- lower-bound: minimo numero di Routees che devono rimanere attivi.
- upper-bound: massimo numero di Routees che si possono creare.
- messages-per-resize: numero di messaggi da smistare dopo il quale il Router valuta l’aggiustamento del numero dei Routees.
- rampup-rate: percentuale di incremento della dimensione del pool (20% di default).
- backoff-rate: percentuale di decremento della dimensione del pool (10% di default).
- pressure-threshold: valore utilizzato per decidere se incrementare il pool.
  - 0 (zero) - tutti gli Attori sono occupati e non ci sono messaggi nella mailbox
  - 1 (uno) - tutti gli Attori sono occupati e c’è almeno un messaggio nella mailbox
  - N - tutti gli Attori sono occupati e ci sono N messaggi nella mailbox (con N > 1)

- backoff-threshold: valore utilizzato per decidere se decrementare il pool. Il valore di default è 0.3 e significa che il pool di Routees verrà decrementato se meno del 30% degli Attori saranno occupati.

Capitolo 6

Approccio event-driven alla programmazione Web

Negli ultimi anni un “nuovo“ approccio alla programmazione Web (in particolar modo lato server) sta comparendo sulla scena: la programmazione event-driven. Non a caso la parola “nuovo” è stata posta tra virgolette nella frase precedente. Possiamo individuare un primordiale abbozzo di questo modello verso la metà degli anni ’70 con l’uscita di una pubblicazione intitolata “Structured Design” di Glenford J. Myers, Wayne P. Stevens, and Larry Constantine uscita nel 1974. Negli anni successivi, e fino la fine del decennio, svariati libri sull’argomento vengono pubblicati ma l’archetipo del modello event-driven lo si ritrova proprio nella pubblicazione di Myers, Stevens e Constantine (Figura 6.1) in cui i tre scrivono che “Una transazione inizia quando un qualsiasi elemento come informazione, controllo, segnale, evento, o cambio di stato è inviato al processo che esegue il Transaction Center”. Inoltre viene chiarito il ruolo di un Transaction Center elencando le azioni che deve svolgere:

- ricezione della transazione in un formato grezzo
- analisi della transazione per determinarne il tipo specifico
- smistamento del tipo di transazione individuato
- completamento del processo di ogni transazione

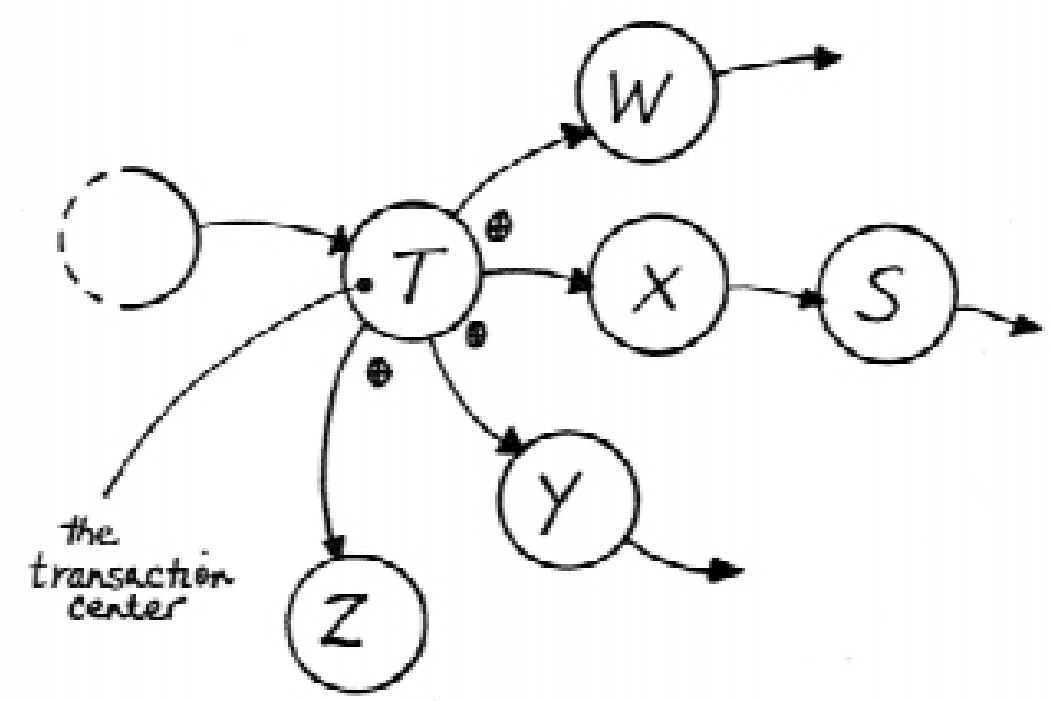


Figura 6.1: Transaction Center estratto da "Structured Design" di Glenford J. Myers, Wayne P. Stevens, and Larry Constantine

A posteriori potremmo individuare nelle definizioni espresse in "Structured Design" quello che in seguito venne individuato come un pattern: Handlers pattern come lo definisce Stephen Ferg. Questo pattern è composto da tre elementi:

- stream di elementi informativi: "eventi" ("transazioni" in "Structured Design")
- un dispatcher ("Transaction Center" in "Structured Design")
- set di Handlers (gestori degli eventi)

Stephen Ferg riprende la descrizione fatta circa il Transaction Center e descrive il Dispatcher come un componente il cui compito è quello di prendere ogni evento che arriva, analizzarlo per determinare a quale tipo di evento appartiene e quindi inviarlo al gestore appropriato. Il Dispatcher è un componente che rimane in ascolto degli eventi che giungono da uno stream, ecco quindi che una volta smistato un evento il Dispatcher deve tornare a prelevare il successivo evento dallo stream di input. In "Event-Driven Programming: Introduction, Tutorial, History" Stephen Ferg introduce l'utilizzo di questo pattern nell'ambito

della programmazione di GUI (Graphical User Interface) ed infatti è proprio in questo ambito che la programmazione event-driven trova terreno fertile: perché le gestione delle interfacce grafiche implica intrinsecamente la gestione degli eventi generati dall'utente utilizzatore.

Non è strano che, a partire dal Web 2.0 in cui compaiono le prime forme di pagine dinamiche, ritroviamo lo stesso pattern applicato lato client in quanto le interfacce grafiche iniziano a popolare anche i siti Internet. La gestione degli eventi grafici in una pagina Web è possibile grazie al linguaggio Javascript che permette di introdurre comportamento dinamico in risposta alle azioni dell'utente. Ad esempio, al click del pulsante di conferma di una form è possibile mostrare un messaggio di avviso, oppure avviare una procedura di validazione delle informazioni inserite ed eventualmente bloccare l'invio dei dati mostrando all'utente gli errori o la mancanza di dati obbligatori. Altra importante azione implementabile è quella di caricare informazioni extra senza bisogno di ricaricare l'intera pagina. In questo caso, all'avvento di uno specifico evento come può essere il click su una particolare area della pagina oppure lo scadere di un timeout, tramite una chiamata AJAX è possibile reperire le informazioni presso il server e, in modo asincrono, mostrarle quando giungono al client.

Anche lato server, ed in generale nella programmazione algoritmica, si fa strada un tipo di programmazione event-driven utile per risolvere in modo più agevole alcuni tipi di problemi. Ad esempio è frequente l'utilizzo del pattern Observer in cui alcune entità si registrano come ascoltatori presso un'altra entità generatrice di eventi. Quando questa entità genera un evento allora tutti gli ascoltatori registrati , o solamente alcuni a seconda della logica applicativa, verranno notificati.

In generale, la programmazione event-driven ed i pattern applicabili con essa, venivano utilizzati per risolvere problemi di programmazione grafica (GUI e pagine Web) e algoritmica, ma i modelli che la facevano da padrone, per quanto riguardava la gestione delle richieste effettuate a un Web Server, erano quelli multi processo e poi multi thread.

6.1 Panoramica sulle gestioni delle richieste client

Come abbiamo visto nei capitoli precedenti, i fattori principali che hanno guidato l'evoluzione della programmazione Web sono stati principalmente due: la scalabilità e la semplicità di programmazione. Il primo fattore è stato dettato dal continuo e sproporzionato

aumento degli utilizzatori del Web, siano essi essere umani o macchine (comunicazioni Machine2Machine e InternetOfThings), che ha portato alla continua ricerca di potenziamento dell'hardware ma soprattutto di sempre più efficienti approcci alla gestione delle richieste. Il secondo fattore deriva da una crescente complessità della programmazione dovuta alla concorrenza di esecuzione delle richieste client e della loro gestione.

Nel tempo, man mano che il numero di richieste che un server doveva gestire al secondo aumentava, si palesava l'inefficienza del modello adottato e si cercava un'alternativa migliore: ciò avvenne con la gestione a processi, evolutasi poi in favore di una gestione a thread, e successivamente avvenne anche per quella a thread. Immaginiamo un Web server che debba soddisfare 10.000 richieste al secondo e che per ogni richiesta crei un Thread. Per quanto esiguo possa essere l'impatto della creazione di un Thread, alla lunga questo approccio è destinato a collassare. Le soluzioni che si possono adottare sono molteplici: a partire dalla creazione di un prefissato pool di Thread e del loro riutilizzo (strategia adottata anche con i processi ma che penalizza lo throughput all'aumentare del carico di lavoro) fino alla dispiegazione di molteplici macchine che cooperano in parallelo assieme ad un bilanciatore di carico che assegni loro in modo equo le richieste provenienti dai client. L'utilizzo di queste metodologie, anche abbinate insieme, risulta efficace in quanto impedisce il collasso del sistema e garantisce buone performance di latenza e throughput ma che dire dell'efficienza? L'overhead introdotto dalla concorrenza gestita tramite Threads diventerebbe non più trascurabile e risulterebbe essere un terribile spreco di risorse.

6.2 L'approccio event-driven: Node.js

Nel 2009 fa la sua comparsa la prima release di Node.js: un runtime environment cross-platform, open-source (licenza MIT), dedicato allo sviluppo di Web Applications server-side. Node.js fornisce un'architettura event-driven che fa uso di API I/O non bloccanti specifiche per ottimizzare scalabilità e throughput nelle applicazioni Web real-time.

Le applicazioni eseguite in questo tipo di ambiente sono scritte in Javascript ed il codice viene eseguito dal motore Google V8. All'interno di un'applicazione Node.js è possibile utilizzare librerie proprie o provenienti dalla folta comunità cresciuta attorno a questo progetto. La gestione delle librerie come la loro distribuzione, installazione e rimozione è facilitata dall'utilizzo del package manager npm.

Node.js offre un ambiente veramente facile da utilizzare e, con poche righe di codice, è possibile costruire una vera e propria applicazione. Node.js è eseguibile su qualsiasi sistema operativo poiché utilizza come linguaggio di interfacciamento con l'OS e di esecuzione del motore Javascript i comuni linguaggi C e C++ mentre la vera e propria logica applicativa utilizza Javascript. Una volta installato l'environment è facile creare un esempio come il classico "Hello World" senza ulteriori installazioni o difficili configurazioni: basta aprire un file di testo e scrivere il seguente codice:

```
var http = require('http');

http.createServer(function (request, response) {
  response.writeHead(200, {'Content-Type': 'text/plain'});
  response.end('Hello World\n');
}).listen(8124);

console.log('Server running at http://127.0.0.1:8124/');
```

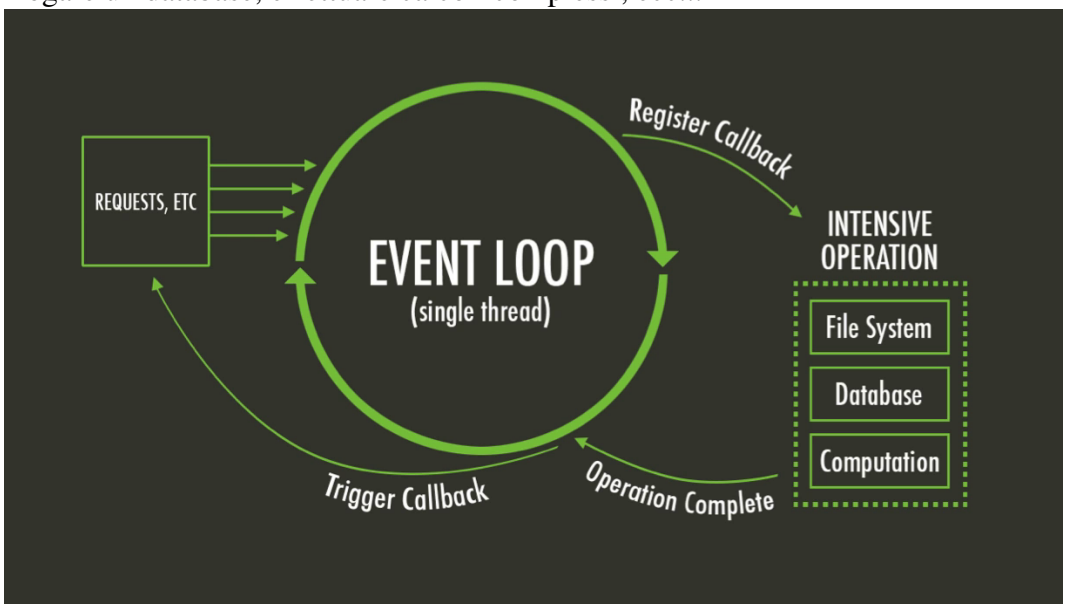
e poi eseguirlo aprendo una shell e digitando il comando "node":

```
> node example.js
Server running at http://127.0.0.1:8124/
```

Quando, dal nostro browser, apriremo la pagina all'indirizzo indicato allora Node.js prenderà in carico la nostra richiesta, eseguirà il codice contenuto all'interno dell'handler e tornerà in ascolto sulla porta specificata per gestire la prossima richiesta.

In questo esempio si può notare come l'esecuzione del codice che porta alla soddisfazione della richiesta non sia concorrente. Se il codice contenuto nell'handler impiegasse 3 secondi per terminare la propria esecuzione e questo server ricevesse 100 richieste simultanee allora l'ultima richiesta della coda sarebbe soddisfatta dopo 300 secondi: un tempo inammissibile. Ecco perché non bisogna mai impiegare Node.js per effettuare operazioni CPU-bound. Se analizziamo l'esempio notiamo che manca una parte importante dell'architettura event-driven: lo smistamento della richiesta verso un handler. Proprio il delegare

all'handler il lavoro da svolgere permette all'Event Loop di non bloccarsi e quindi di tornare immediatamente a prendere in carico la prossima richiesta. Sarà compito dell'handler effettuare l'operazione richiesta come ad esempio leggere/scrivere in un file su filesystem, interrogare un database, effettuare calcoli complessi, ecc...



6.3 Gestione della concorrenza nel modello event-driven

Di primo acchito l'utilizzo di Node.js sembra essere un ritorno al passato con una gestione single thread delle richieste. Questa considerazione nasce dal fatto che si è abituati a un tipo di programmazione in cui ogni richiesta è gestita separatemene da un thread o processo che si occuperà di svolgere ogni tipo di operazione, anche molto onerosa dal punto di vista computazionale o addirittura bloccante. In Node.js il thread che accetta le richieste(Event Loop) non è quello che poi andrà a gestirle ma il suo compito consiste principalmente in:

- accettare la richiesta
- compire una logica preliminare atta a decidere quale handler chiamare e quale callback di ritorno registrare
- ricevere i risultati delle callback
- inviare la risposta al client

Tutte queste operazioni non devono mai essere operazioni computazionalmente onerose e soprattutto bloccanti. Ad esempio, se in una richiesta viene chiesto di interrogare il database per ottenere il numero di notifiche di un utente, l'Event Loop non deve effettuare la query bloccante e rimanere in attesa del risultato(Algoritmo 1); facendo cosi non potrebbe accettare altre richieste perché occupato ad attendere il risultato dal database. Piuttosto L'Event Loop dovrebbe inviare la query al database registrando la callback che questo dovrà invocare una volta che i dati saranno pronti. A quel punto l'Event Loop eseguirebbe la callback e invierebbe il risultato al client. Anche questa ultima operazione non deve essere onerosa: il dato che il database ritorna dovrebbe essere già il risultato finale o comunque si dovrebbe richiedere veramente poco sforzo per ottenerlo(Algoritmo 2).

Algoritmo 1 Interrogazione del DB bloccante

```
function getNotifications(id) {
   var user = db.query(id);
   return user.notifications;
}
var query = url.parse(req.url,true).query;
var n = getNotifications(query.id);
res.setHeader('Content-Type', 'application/json');
res.send(JSON.stringify({ notifications: n }));
```

Algoritmo 2 Interrogazione del DB non bloccante

```
var query = url.parse(req.url,true).query;
db.query(id, function(user, err){
   if(err) throw err;
   res.setHeader('Content-Type', 'application/json');
   res.end(JSON.stringify({ notifications: user.notifications }));
};
```

6.4 Pro e contro dell'approccio event-driven

I vantaggi portati da questo tipo di approccio sono principalmente l'azzeramento dell'overhead dovuto al cambio di contesto, cosa che in Node.js non esiste dal momento che è single thread, e quindi globalmente le performance ne beneficiano dal momento che il sistema operativo non deve creare e distruggere Threads e gestirne l'esecuzione tramite lo scheduler. Da ciò ne deriva anche un beneficio in termini di memoria occupata, molto

minore rispetto gli approcci visti fino ad ora. Sono questi i fattori che hanno portato rapidamente al successo Node.js e grazie alle sue performance è diventato velocemente il punto di riferimento per lo sviluppo di Web Applications, sopratutto se real-time come sistemi di messaggistica e notifiche, servizi di conversioni di dati, ecc...

Queste strabilianti performance però sono limitate ad un uso specifico. Infatti Node.js dà il meglio di se quando gestisce principalmente operazioni I/O bound. Per utilizzi più general purpose si rimane ancora legati ad approcci più consolidati come quello a Thread o ad Attori. Ciò non significa però che al bisogno Node.js non possa gestire anche una programmazione multi-thread: per questo scopo esistono i Web Workers. I Web Workers sono pezzi di codice Javascript eseguiti in background parallelamente al codice Javascript principale e vengono eseguiti su un thread separato. Questo meccanismo permette all'Event Looper di Node.js di eseguire codice CPU-bound senza bloccarsi. Questo approccio rompe gli schemi dell'architettura event-driven in favore di un'architettura ibrida, mista con quella multi-thread, e tende a perdere tutti i punti di forza propri del modello a single-thread.

Approccio molto più efficace ed efficiente è quello di rimanere strettamente fedeli all'architettura originale di Node.js e di sfruttare la scalabilità. Ad esempio, se un server possiede 16 core ecco che si possono lanciare 16 istanze di Node.js e un bilanciatore di carico software a monte che si assicura che il volume di richieste sia equo tra le istanze. Con questo approccio ogni core della CPU esegue l'Event Loop di ogni istanza, quindi nessun cambio di contesto e nessun intervento dello scheduler dell'OS, moltiplicando il volume di richieste che il server è in grado di gestire. Inoltre questo sistema è perfettamente scalabile ed a seconda del volume di carico è possibile aumentare le macchine e quindi le istanze di Node.js.

Capitolo 7

Caso di studio reale di programmazione ad Attori

Nella mia esperienza lavorativa quotidiana non capita spesso di vedere progetti che utilizzano un paradigma ad Attori. Solitamente l'approccio è quello dei Threads che compiono task paralleli, vengono coordinati con l'utilizzo di monitor o semafori e storicizzano informazioni in basi di dati condivise. Riconosco che questo approccio crea notevole confusione, soprattutto in punti del codice dove non è subito chiaro l'andamento del flusso di controllo. Ho voluto proporre come caso di studio un nuovo progetto sviluppato con l'utilizzo di Threads concorrenti e di questo re-implementerò delle parti utilizzando il modello ad Attori per evidenziare pro e contro. Inoltre, nello svolgere questa comparazione, ho analizzato le attuali proposte tecnologiche che il panorama informatico offre in campo aziendale.

7.1 Scelte e requisiti tecnologici

Il progetto di studio ha come requisito l'utilizzo del framework .Net 4.5 ed è sviluppato in linguaggio C#. Più nello specifico, il progetto originale utilizza MVC 5 di Microsoft come modello di programmazione Web. Per la successiva parte di programmazione ad Attori ho scelto di utilizzare la libreria Akka nella sua versione per il framework .Net.

7.2 Introduzione al progetto

Il progetto, denominato “Take A Ticket”(abbr. TAT), ha come scopo la prenotazione concorrente di servizi on-line.

Si fa riferimento al comune esempio della fruizione dei servizi presso gli uffici postali. L’utente che desidera utilizzare i servizi offerti è obbligato a prenotare il proprio turno attraverso l’acquisizione di un ticket numerato. La particolarità del progetto di studio è la possibilità che ha l’utente di posticipare il proprio ticket ad un secondo momento senza perdere la posizione acquisita originariamente. In questo modo l’utente che “abbandona” momentaneamente la posizione verrà superato dagli altri utenti ma al suo ritorno recupererà la posizione originale. Questa modalità di assegnamento delle priorità permette a tutti gli utenti di usufruire del servizio richiesto senza penalizzare le assenze. Per non incoraggiare l’abbandono temporaneo del ticket si è introdotto il concetto di esclusione temporizzata con il quale tutti i ticket assenti,superato il periodo di timeout, perderanno la priorità acquisita.

7.3 Punti di interesse e criticità

Considerando l’ambiente fortemente concorrente in cui si intende utilizzare il software in oggetto sono emersi dalla fase di analisi del progetto 3 punti di criticità principali:

1. creazione di un nuovo ticket
2. re-inserimento di un ticket
3. estrazione del prossimo ticket

Le criticità valutate consistono nella difficoltà di mantenere l’ordine delle operazioni, a meno delle latenze di rete, e soprattutto di prevenire incoerenze nei dati. Prendiamo ad esempio il caso in cui, nell’ufficio postale citato poc’anzi, molti clienti tentino di prendere un biglietto nello stesso momento (vedi punto 1). Per evitare inutili asti si rende necessaria la presenza di un meccanismo di coordinazione che imponga un ordine e permetta ai clienti di prendere il proprio biglietto in modo ordinato e pacifico. Questa situazione quotidiana può essere riportata in un sistema software programmato a Thread in cui, senza un controllo sulla coordinazione, le entità tentino di accaparrarsi la risorsa (ticket) prima delle altre. In

questo modo ad ogni entità verrà assegnato un ticket non in base all'ordine di arrivo ma in modo non predicibile. Inoltre non è garantito che a tutte le entità venga assegnato un biglietto: potrebbe accadere che alcune non vengano mai servite (starvation).

Un altra situazione molto delicata è il re-inserimento di un ticket nella coda (vedi punto 2). In questo caso, soprattutto in situazioni di forte concorrenza, è molto probabile che l'operazione di re-inserimento di un ticket si sovrapponga temporalmente con una delle altre azioni elencate precedentemente. Ad esempio potrebbe succedere che, mentre un utente riattiva il proprio ticket, un altro ne acquisti uno nuovo. Secondo le specifiche del progetto il ticket, se rientrato prima della scadenza del timeout, mantiene la priorità acquisita altrimenti verrà messo in fondo la fila. Se l'operazione di creazione del nuovo ticket e quella di assegnamento dell'ultima posizione della coda la ticket ri-attivato dovessero sovrapporsi allora si otterrebbe un comportamento non predicibile in cui si potrebbero verificare tre situazioni:

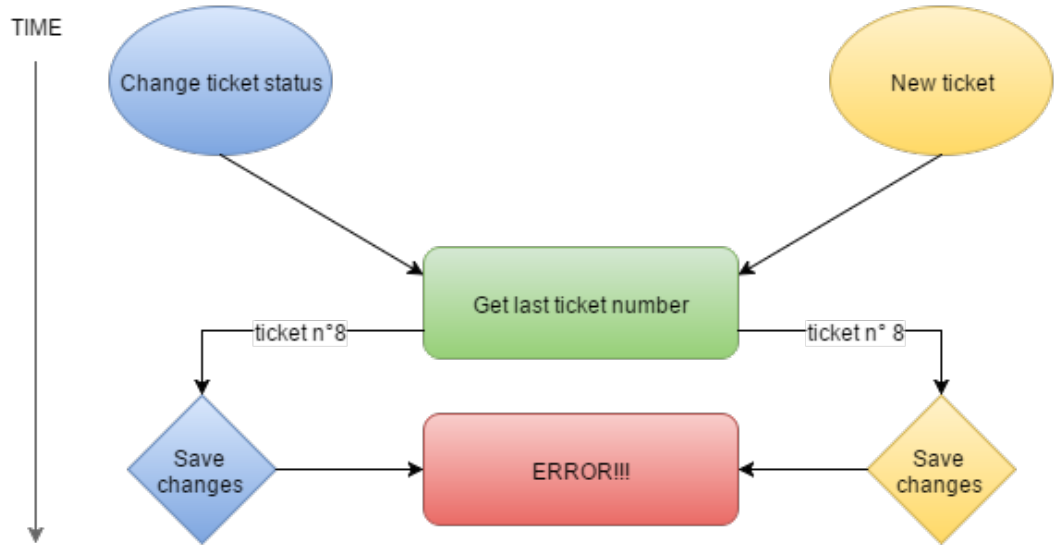
1. il nuovo ticket risulta avere priorità maggiore del ticket ri-attivato

2. il ticket ri-attivato risulta avere priorità maggiore del nuovo ticket

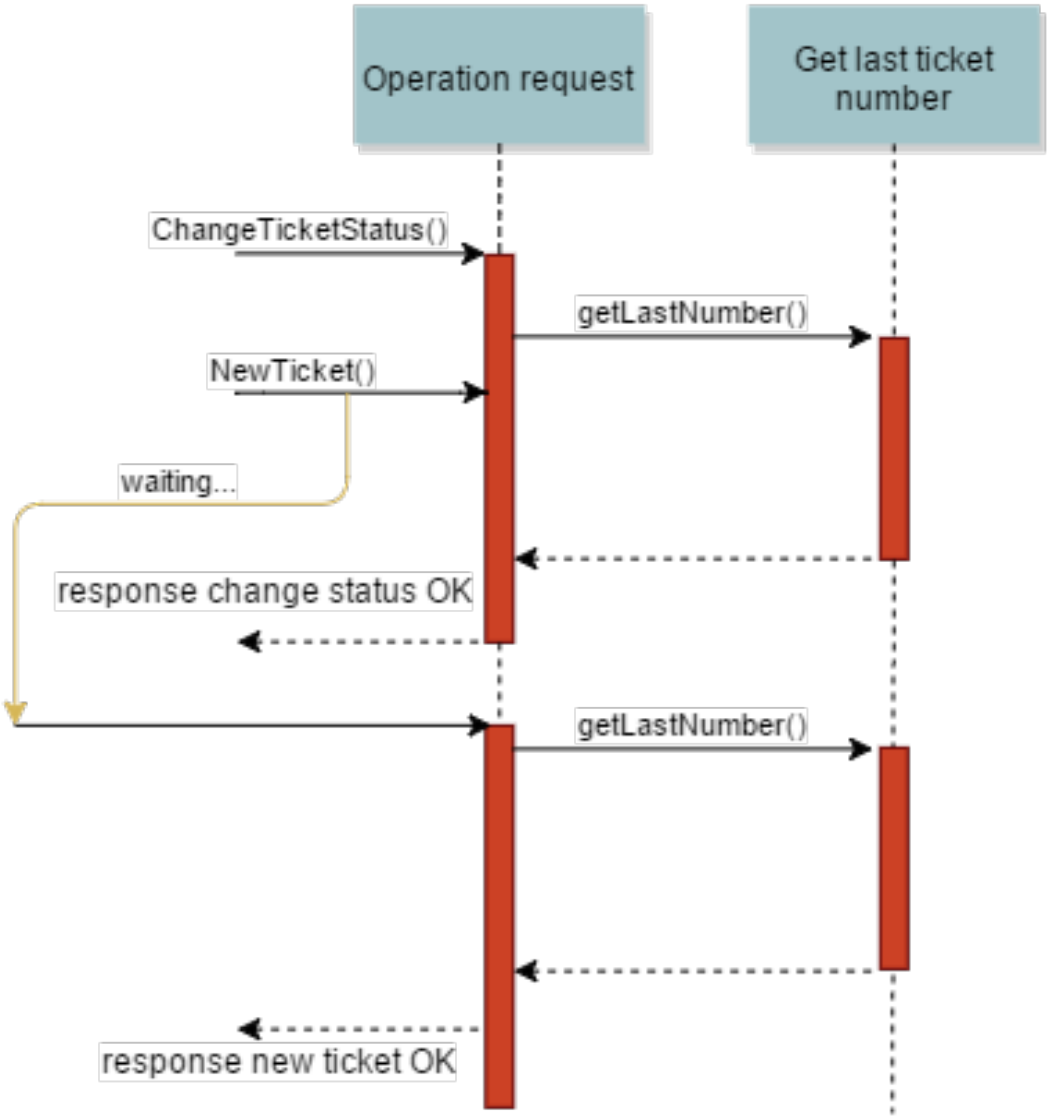
3. i due ticket hanno la stessa priorità

Mentre le prime due opzioni risultano accettabili, la terza opzione è inaccettabile per il sistema poiché non possono esistere due o più biglietti con lo stesso numero di posizione.

```
Change Ticket Status ->
...
int number = DbUtility.GetCurrentQueueLastTicketNumber(ticket.TicketQueueId, db);
ticket.PositionNumber = number + 1;
db.SaveChanges();
...
Take A New Ticket ->
...
int number = DbUtility.GetCurrentQueueLastTicketNumber(queueId, db);
Ticket model = new Ticket();
model.TicketQueueId = queueId;
model.PositionNumber = number + 1;
db.Tickets.Add(model);
db.SaveChanges();
...
```



Questo ultimo esempio chiarifica come sia necessario un sistema di coordinazione in quelle sezioni critiche del codice, come in questa appena esaminata, in cui la mutua esclusione è d'obbligo per evitare incoerenze nei dati e nei comportamenti. Ad esempio, per impedire che le azioni di assegnamento di un nuovo ticket e di cambio di stato di un ticket esistente vadano in conflitto, è necessario che queste due azioni risultino atomiche: mentre una è in esecuzione l'altra non può essere eseguita.

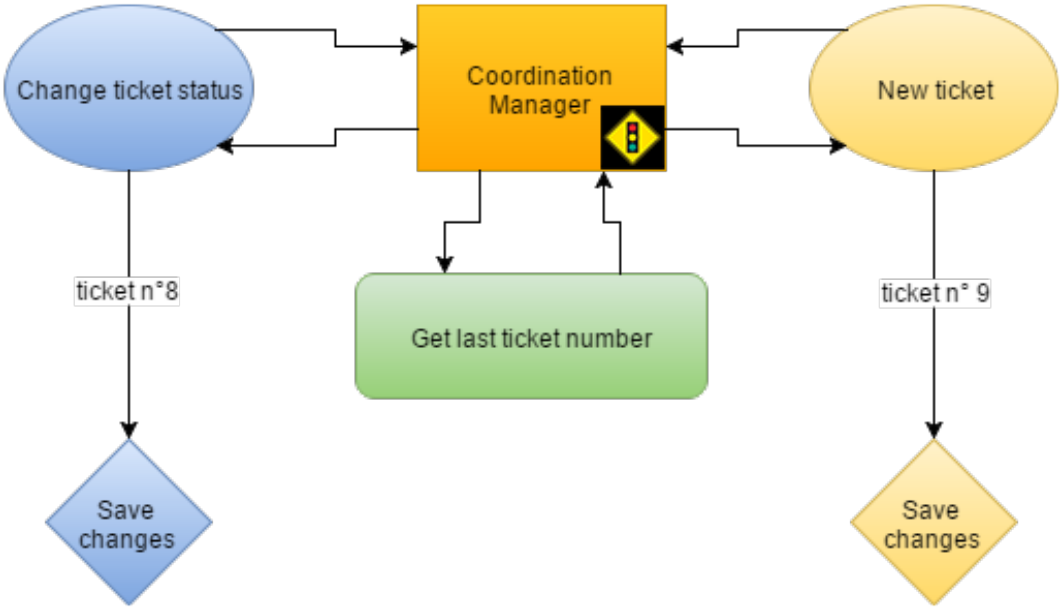


7.4 Meccanismi di coordinazione nella soluzione a Threads

L'approccio a Threads è il primo modello utilizzato e di conseguenza la soluzione adottata è quella di un componente che supervisioni l'accesso a quelle aree del codice che devono essere eseguite da un thread alla volta: il CoordinationManager. Questo oggetto non appartiene alla libreria standard di .Net ma è stato costruito basandosi sul fatto che le operazioni di creazione di un ticket, cambio stato e avanzamento della coda devono risultare atomiche non da un punto di vista globale ma solamente dal punto di vista della coda. L'utilizzo

di un Monitor avrebbe forzato l'intera applicazione a rispettare la mutua esclusione delle zone desiderate, al contrario è sufficiente che solo le operazioni sulla stessa coda siano mutuamente esclusive. Per queste ragioni è stato implementato il CoordinationManager il quale utilizza un semaforo per ogni coda.

Ogni nuova richiesta, come quella di un nuovo ticket effettuata dal Thread A, viene immediatamente messa in esecuzione. Prima che il flusso di controllo di A giunga in prossimità del codice in cui si applica la logica di creazione di un nuovo ticket, questi deve registrarsi presso il CoordinationManager per ottenere il permesso di accedervi. Se nessun Thread si trova all'interno della zona allora A viene fatto passare. Nel caso il Thread B richieda l'accesso alla zona critica in concomitanza con A e prima della sua uscita, B viene fermato e messo in uno stato di attesa. Non appena il Thread A esce dalla zona controllata dal CoordinationManager allora a B viene concesso di continuare la propria esecuzione. In questo modo le due richieste vengono eseguite sequenzialmente e senza sovrapposizioni.



In base ai test condotti questo approccio elimina definitivamente l'insorgere di situazioni indesiderate di collisione tra le operazioni ed inoltre mantiene l'ordine di registrazione dei thread presso l'entità coordinante. Il test è stato condotto creando 63 Threads, ognuno dei quali tenta di creare un nuovo ticket. Per simulare nel modo più fedele possibile una situazione reale i thread vengono fatti partire tutti nello stesso istante ed ognuno di essi incontra un ritardo casuale, tra 0 e 10.000 ms, prima di inoltrare effettivamente la richiesta. Di seguito viene riportato un log esplicativo:

```
Thread 13 is waiting for take a ticket...
Thread 13 had taked ticket 862
Thread 21 is waiting for take a ticket...
Thread 9 is waiting for take a ticket...
Thread 21 had taked ticket 863
Thread 22 is waiting for take a ticket...
Thread 9 had taked ticket 864
Thread 22 had taked ticket 865
Thread 16 is waiting for take a ticket...
Thread 16 had taked ticket 866
Thread 23 is waiting for take a ticket...
Thread 23 had taked ticket 867
```

Come si può evincere dal log, i Threads ottengono numeri di posizione consecutivi e diversi tra loro. Inoltre si rispetta la priorità in quanto il thread che inoltra la richiesta è anche quello che riceverà il ticket successivo (vedi Thread nř 21,9 e 22).

7.4.1 Esempio di sviluppo con modello Thread based

In questa implementazione il fulcro della coordinazione risiede nel CoordinationManager. Questo oggetto incapsula la business logic basata sull'utilizzo delle primitive di sincronizzazione offerte dai Mutex della libreria System.Threading di .Net. Più esattamente, ogni qual volta si compia un'operazione su una coda regolata da questo componente allora viene richiamato, o se necessario creato, il Mutex dedicato alla coda interessata. Per convenzione si è scelto di identificare i Mutex sfruttando gli identificativi univoci delle code. In questo modo ogni coda possiede il proprio Mutex e le regolazioni di accesso delle diverse code concorrentemente attive non interferiscono le une con le altre. Di seguito si riporta il costruttore del CoordinationManager ed i metodi invocati per acquisire e rilasciare i diritti di precedenza per l'accesso alle aree mutuamente esclusive del codice.

```
public CoordinationManager(int queueId){
  this.CoordinatorName = CreateName(queueId);
  Mutex openMutex = null;
  bool exist = Mutex.TryOpenExisting(this.CoordinatorName, out openMutex);
  if (exist){
    this.Coordinator = openMutex;
```

```
  }else{
    this.Coordinator = new Mutex(false, this.CoordinatorName);
  }
}
```

Costruttore del CoordinationManager

```
public void Wait(){
  this.Coordinator.WaitOne();
  toRelease = true;
}

public void Release(){
  if(toRelease){
    toRelease=false;
    this.Coordinator.ReleaseMutex();
  }
}
```

Metodi per richiedere l'accesso (Wait) e rilasciare il lock (Release)

```
public static void TakeATicket(int queueId, Ticket model, TatContext db){
  var coordinationManager = new CoordinationManager(queueId);
  try{
    coordinationManager.Wait();
    int number = DbUtility.GetCurrentQueueLastTicketNumber(queueId, db);
    CreateTicketAndSave(number, db);
  }catch (Exception ex){
    throw ex;
  }finally{
    coordinationManager.Release();
  }
}
```

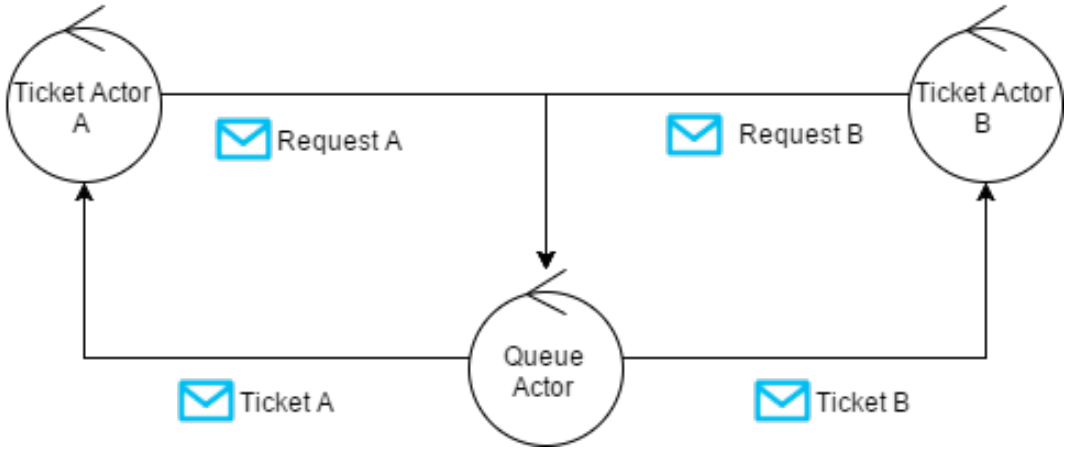
Sezione critica di mutua esclusione in cui si storicizza la creazione di un nuovo ticket

7.5 La soluzione ad Attori

La soluzione che fa uso del modello ad Attori non necessita di un'entità coordinante come nel caso del modello a Thread. Infatti l'idea di esecuzione sequenziale delle operazioni

critiche è insita nel modello. Questo implica una maggiore semplicità di sviluppo e una notevole diminuzione di sovrapposizione delle operazioni.

In questo modello gli attori A e B inviano un messaggio di richiesta, ad esempio per ottenere un nuovo ticket, all'attore che gestisce la coda. Le richieste giungono nella mail box del gestore della coda e questi le eseguirà una alla volta. In questo modo i due attori richiedenti avranno ciascuno come risposta il ticket richiesto senza alcun pericolo di ricevere ticket con lo stesso numero progressivo.




```
Debug Trace: 09.39.07.552661 -  Test is started!!!
09.39.13.321841 - ACTOR_TICKET_0: asking for take a ticket.
09.39.13.321841 - ACTOR_TICKET_2: asking for take a ticket.
09.39.13.331851 - ACTOR_TICKET_0: asked ticket.
09.39.13.331851 - ACTOR_TICKET_2: asked ticket.
09.39.13.332853 - ACTOR_TICKET_1: asking for take a ticket.
09.39.13.332853 - ACTOR_TICKET_1: asked ticket.
09.39.13.336856 - ACTOR_TICKET_4: asking for take a ticket.
09.39.13.336856 - ACTOR_TICKET_3: asking for take a ticket.
09.39.13.339857 - Queue actor: saving a new ticket requested from ACTOR_TICKET_0
09.39.13.720225 - Queue actor: saved ticket 77 requested from ACTOR_TICKET_0
09.39.13.720225 - Queue actor: saving a new ticket requested from ACTOR_TICKET_2
09.39.13.720225 - ACTOR_TICKET_0: my ticket id is 77 and my position into the queue is 64
09.39.13.792264 - Queue actor: saved ticket 78 requested from ACTOR_TICKET_2
09.39.13.792264 - Queue actor: saving a new ticket requested from ACTOR_TICKET_1
09.39.13.792264 - ACTOR_TICKET_2: my ticket id is 78 and my position into the queue is 65
09.39.13.853319 - Queue actor: saved ticket 79 requested from ACTOR_TICKET_1
09.39.13.854321 - ACTOR_TICKET_1: my ticket id is 79 and my position into the queue is 66
09.39.18.338368 - ACTOR_TICKET_3: asked ticket.
09.39.18.338368 - Queue actor: saving a new ticket requested from ACTOR_TICKET_3
09.39.18.338368 - ACTOR_TICKET_4: asked ticket.
09.39.18.392394 - Queue actor: saved ticket 80 requested from ACTOR_TICKET_3
09.39.18.392394 - Queue actor: saving a new ticket requested from ACTOR_TICKET_4
09.39.18.392394 - ACTOR_TICKET_3: my ticket id is 80 and my position into the queue is 67
09.39.18.452449 - Queue actor: saved ticket 81 requested from ACTOR_TICKET_4
```

```
09.39.18.452449 – ACTOR_TICKET_4: my ticket id is 81 and my position into the queue is 68
09.39.18.511501 – Test is finished!!!
```

7.5.1 Esempio di sviluppo con modello Actor based

Come precedentemente anticipato si è utilizzata la libreria Akka nella sua implementazione per il framework .Net. In questa implementazione un attore definisce i propri comportamenti alla sua creazione, all'interno del costruttore, invocando il metodo Receive ed indicando il tipo di dato in input ed il metodo da invocare per gestire l'evento. Ad esempio il QueueActor definisce l'azione TakeATicket con la quale gli altri attori possono registrarsi alla coda ed ottenere un nuovo biglietto.

```
public QueueActor(){
  Receive<ActorInfo>(info => TakeATicket(info));
}
```

I TicketActor invece definiscono l'azione TicketSaved la quale viene invocata dal QueueActor per notificare il nuovo biglietto.

```
public TicketActor(){
        Receive<Ticket>(ts => TicketSaved(ts));
}
```

Come si può immaginare il flusso di esecuzione diventa più semplice rispetto la variante a Thread: il TicketActor chiede un nuovo ticket inviando le informazioni necessarie(es.: email) al QueueActor il quale esamina la richiesta, crea un nuovo biglietto e notifica l'attore richiedente con il biglietto appena creato. L'invio di un messaggio ad un attore si effettua invocando il metodo Tell su quell'attore. Ad esempio un attore che desidera richiedere un nuovo ticket eseguirà il seguente codice:

```
QueueActor.Tell(myInfo);
```

L'invocazione del metodo Tell non ritorna alcun valore. Questo comportamento è legato al design del modello adottato: l'invio di un messaggio non produce alcun risultato immediato e non è in alcun modo bloccante.

Quando l'attore gestore della coda prenderà in carico la richiesta allora invierà un messaggio all'attore richiedente informandolo dell'esito.

7.6 Test di performance e valutazioni

In ambito aziendale la bontà di un modello è valutata anche in base alle performance offerte: per questo motivo sono stati sviluppati due casi di test, uno per il modello a Threads e l'altro per il modello ad Attori, e sono stati messi a confronto.

7.6.1 Struttura dei test

I due test sono stati costruiti in modo da essere confrontati nonostante i diversi approcci implementativi richiesti dai due modelli di programmazione così diversi tra loro. In forma generale i test seguono la seguente forma:

1. Pulizia iniziale dell'ambiente di esecuzione: in questa fase vengono liberate tutte le risorse necessarie per la corretta esecuzione del test in modo che ogni iterazione venga eseguita sempre nel medesimo contesto di esecuzione.

2. Creazione delle infrastrutture necessarie per l'esecuzione: ogni modello necessita di infrastrutture dedicate per eseguire il proprio compito. Nel caso del modello a Thread viene istanziato il componente dedicato alla sincronizzazione dei Threads; nel caso del modello ad Attori viene creato il contesto di esecuzione degli Attori e l'attore coordinante per l'esecuzione del test. Per entrambi i test viene anche preparata la connessione alla base di dati con la quale gli oggetti interagiranno nonché caricata la coda di riferimento per la quale verranno creati i ticket di test.

3. Inizio del test: vengono creati n Threads/Attori ognuno dei quali concorrerà alla creazione del proprio ticket. Ogni entità si occupa della misurazione del tempo richiesto per completare la richiesta di un nuovo ticket.

4. Fine test: viene preso il tempo totale di esecuzione dell'intero test e, basandosi sui singoli tempi di ogni Thread/Attore, viene calcolato il tempo medio richiesto per l'assegnazione di un singolo ticket.

5. Scrittura delle informazioni risultanti dal test: i tempi misurati vengono salvati su file in formato CSV per poi essere elaborati.

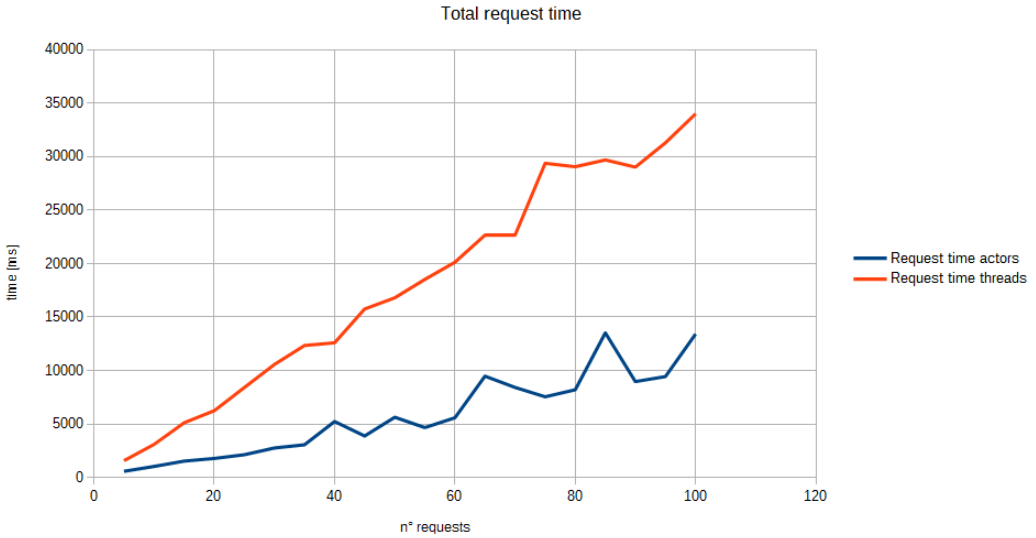
6. Pulizia finale dell'ambiente di esecuzione: tutte le risorse utilizzate durate il test vengono liberate.

Per analizzare il comportamento del sistema al crescere del carico di lavoro ciascun test è stato eseguito per 50 volte e per ogni nuova esecuzione del test sono state aggiunte due richieste concorrenti rispetto il test precedente. Ciò significa che la prima iterazione del test ha comportato 2 richieste concorrenti, la seconda 4 richieste concorrenti, la terza 6 richieste e così via, fino ad arrivare all'ultima iterazione in cui sono state effettuate 100 richieste simultanee.

I test sono stati eseguiti su una macchina con CPU a 64 bit equipaggiata con 2 core fisici alla massima velocità di 2 Ghz,sistema operativo Microsoft Windows 10 64 bit, 8 GB di ram.

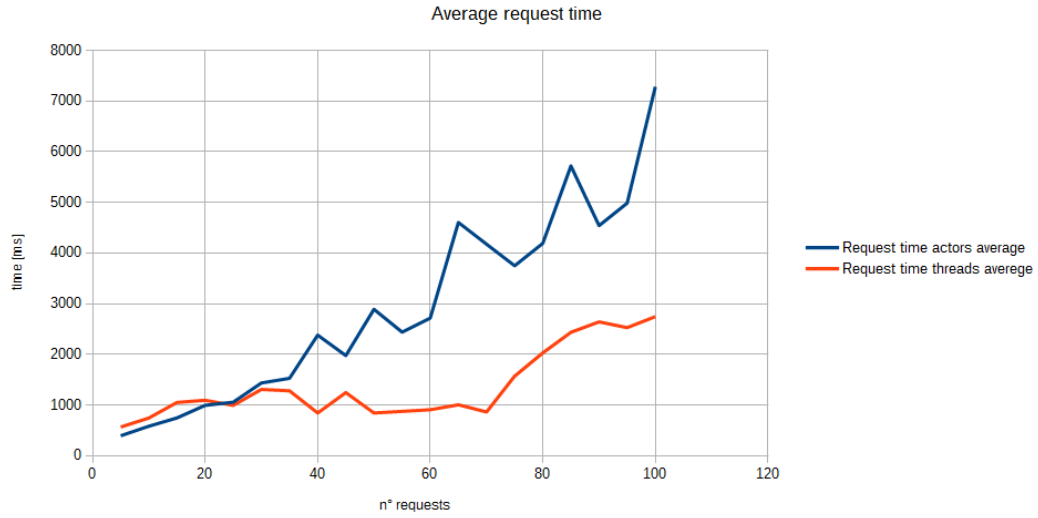
7.6.2 Valutazioni qualitative

I risultati ottenuti dai test non devono essere interpretati in termini di performance assolute ma comparati tra loro in modo qualitativo per poter analizzare le performance dei due modelli applicati al caso di studio in oggetto, traendone informazioni utili al loro studio e valutazione.



In questo grafico è possibile osservare l'andamento del tempo richiesto per eseguire tutte le richieste concorrenti. La prima cosa da notare è che per entrambi i modelli si è ottenuta una crescita approssimata di tipo lineare all'aumentare del carico di lavoro. Oltre a ciò è evidente come l'approccio ad Attori sia quello vincente in termini di performan-

ce assolute in quanto nettamente più veloce tanto più sono le richieste simultaneamente eseguite dal server.



In questo grafico è possibile osservare l'andamento del tempo medio richiesto per eseguire una singola richiesta. Per il modello a Thread si ha un andamento costante fino alla soglia delle 70 richieste concorrenti. Oltre tale soglia le performance peggiorano velocemente procedendo con aumento approssimativamente lineare. Per quanto riguarda l'approccio ad Attori si può osservare fin dall'inizio del test una crescita pressoché lineare del tempo medio richiesto per eseguire una singola richiesta.

Questi risultati si spiegano analizzando la configurazione della libreria Akka.Net. Di default il Dispatcher di Akka, che si occupa di creare e gestire il ciclo di vita degli Attori, utilizza un pool di Threads prestabilito a 3 unità. Questo spiega perché il tempo medio per singola richiesta risulta essere cosi alto.

Alla luce di questa considerazione è possibile immaginare anche il motivi di un divario prestazionale cosi ampio nel grafico precedente: il continuo cambio di contesto a cui è sottoposto lo scheduler del sistema operativo, nel caso del modello a Threads, introduce un overhead non trascurabile che va a ripercuotersi sulle performance globali. Al contrario il modello ad Attori, che opera in un pool di thread molto più ridotto, riduce l'overhead dello scheduler.

Conclusioni

La concorrenza e la scalabilità sono due grandezze legate a doppio nodo tra loro, intrinsecamente presenti nei moderni sistemi distribuiti. L'esponenziale aumento dei dispositivi connessi alla Rete e degli utilizzatori del Web hanno portato questi due fattori a diventare sempre più preponderanti nella scelta dell'architettura da adottare e del paradigma di programmazione da usare.

Il proliferare di applicazioni Web aumenta il peso di questi due fattori nei moderni sistemi e introduce, soprattutto per le Real Time Web Applications, esigenti requisiti di latenza e recovery. Il Web 3.0 è caratterizzato da un panorama totalmente diverso rispetto al passato: un gran numero di connessioni spesso in idle, scambi di piccole quantità di dati ma in modo molto veloce, inversione di notifica nella comunicazione client-server (server-side message pushing).

Questa tesi si è focalizzata sull'introdurre ed analizzare alcuni aspetti delle moderne metodologie di programmazione Web, tenendo conto della storia dell'evoluzione del Web nel corso del tempo e in prospettiva dei bisogni attuali.

In particolar modo ci si è voluto soffermare sul modello ad Attori esponendo le sue specifiche, i suoi possibili utilizzi e confrontandolo con altri modelli come ad esempio il modello a Threads.

In fine, si è analizzato un reale caso di studio in cui è stato applicato il modello ad Attori e ne sono stati mostrati vantaggi e limiti, corredando lo studio con test di tipo qualitativi per verificare i vantaggi rispetto l'approccio a Threads.

Il modello a Thread è stato per molto tempo l'approccio standard a qualsiasi problema informatico. La sua efficacia viene a mancare quando si ha a che fare sistemi di grandi dimensioni e altamente concorrenti. Il modello ad Attori risulta essere un valido sostituto, soprattutto se consideriamo il valore aggiunto che porta con se rispetto il modello a Threads come la scalabilità, la natura distribuita e i meccanismi di recovery nativi.

Un'architettura differente è quella event-driven che ha attualmente il suo massimo esponente in ambito Web in Node.js. L'architettura adottata da Node.js è incentrata sul concetto che ogni azione deve essere asincrona poiché tutto si basa sull'Event Loop che è single Thread. Da questo punto di vista la vicinanza con il modello ad Attori è molto evidente in quanto ogni singolo Attore può essere interpretato come un'Event Loop che compie azioni asincrone quali l'invio di messaggi. Sarebbe molto interessante poter estendere Node.js in questo senso in modo da utilizzare un approccio più vicino a quello ad Attori. Questa estensione amplierebbe le potenzialità dell'architettura e porterebbe il grande beneficio di avere più Event Loops che comunicano tra loro utilizzando il modello di comunicazione a messaggi.

Ringraziamenti

Ringrazio il Relatore e Professore Alessandro Ricci per l'indispensabile supporto offertomi per la stesura di questa Tesi.

Ringrazio tutti i docenti che ho incontrato durante la mia carriera accademica e che hanno accresciuto in me l'interesse per la disciplina informatica.

Un caloroso ringraziamento ai miei ex colleghi universitari Francesco, Domenico e Raffaele per il supporto durante la mia permanenza in facoltà.

In fine ringrazio i titolari dell'azienda Bsd Software di Cesena per l'accoglienza ricevuta durante il mio tirocinio, i miei ex colleghi Leonardo e Simone per le preziose nozioni insegnatemi e in generale tutti i miei colleghi, passati e presenti.

Bibliografia

1. BERNERS-LEE, Sir Timothy (John). Who's Who 2015 (online Oxford University Press ed.). A & C Black, an imprint of Bloomsbury Publishing plc.

2. FAQ—World Wide Web Foundation

3. "World Wide Web Consortium (W3C) About the Consortium", The World Wide Web Consortium (W3C)

4. O'REILLY, TIM, What is Web 2.0: Design Patterns and Business Models for the Next Generation of Software. Communications & Strategies, No. 1, p. 17, First Quarter 2007.

5. SHACHOR, Gal MILSTEIN, Dan: The Apache Tomcat Connector - AJP Protocol Reference, Apache Software Foundation (2000)

6. O. LASSILA; J. HENDLER, Embracing "Web 3.0", J. IEEE JOURNALS & MAGAZINES

7. J. HENDLER, Web 3.0 Emerging, J. IEEE JOURNALS & MAGAZINES

8. BERNERS-LEE, Tim; HENDLER, James LASSILA, Ora: The Semantic Web. Scientific American (2001), 284(5): 34-43

9. D. EVANS, "The Internet of Things: How the Next Evolution of the Internet Is Changing Everything", Cisco (April 2011)

10. ABBOTT, Martin L. FISHER, Michael T.: Scalability Rules: 50 Principles for Scaling Web Sites, Addison-Wesley Professional (2011)

11. AGHA, Gul: Concurrent object-oriented programming. Commun. ACM (1990), 33: 125-141

12. FOWLER, Martin: Patterns of Enterprise Application Architecture, Addison-Wesley Professional (2002)

13. FOWLER, Martin: Event Sourcing, ThoughtWorks (2005)

14. GUSTAFSSON, Andreas: Threads without the Pain. Queue (2005), 3: 34-41

15. G. HOHPE, Programming without a call stack-event-driven architectures, carfield.com.hk (2006)

16. LEE, Edward A.: The Problem with Threads. Computer (2006), 39: 33-42

17. C. SCHMIDT, Reactor - An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events Douglas, Department of Computer Science Washington University, St. Louis, MO

18. PYARALI, Irfan; HARRISON, Tim; SCHMIDT, Douglas C. JORDAN, Thomas D.: Proactor - An Object Behavioral Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events, Washington University (1997)

19. ROBINSON, D. COAR, K.: The Common Gateway Interface (CGI) Version 1.1, RFC 3875 (Informational) (2004)

20. SCHLOSSNAGLE, Theo: Scalable Internet Architectures, Sams (2006)

21. SUTTER, Herb LARUS, James: Software and the Concurrency Revolution. Queue (2005), 3: 54-62

22. CARL HEWITT; Peter Bishop; Richard Steiger (1973). "A Universal Modular Actor Formalism for Artificial Intelligence". IJCAI.

23. Akka.Net Documentation, http://getakka.net/docs/

24. S. FERG, Event-Driven Programming: Introduction, Tutorial, History, http://event-drivenpgm.sourceforge.net/, Jan. 2006.

25. A. MARDAN, Practical Node.js: Building Real-World Scalable Web Apps, Apress