Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione
Corso di Laurea Triennale in Ingegneria Informatica

PariDHT: sincronizzazione

Relatore: Ch.mo Prof. Enoch Peserico Stecchini Negri De Salvi
Correlatore: Dott. Paolo Bertasi

Laureando: Nicola Gobbo

Anno accademico 2009/2010

Alla mia famiglia,
che mi ha guidato e sostenuto.

A Paola,
che valorizza ogni mio gesto.

Prefazione

Da alcuni anni le reti *peer-to-peer* (P2P da ora in avanti) sono uscite dalla nicchia universitaria in cui sono state teorizzate, entrando a far parte dell'uso abituale che le persone fanno di internet; il merito per questo trend positivo va dato a programmi quali eMule, Azureus, Tor, Skype, i quali sfruttano le reti P2P per condividere file, effettuare telefonate o tutelare la propria privacy, celando la loro complessità agli utenti attraverso interfacce grafiche curate e intuitive.
PariPari è un'applicazione che vuole inserirsi in questo contesto, mirando a fondere in un'unica piattaforma ciò che ora è disponibile solo tramite software distinti e poco cooperanti tra loro; questa flessibile offerta di servizi, assieme alla facilità d'uso e alla garanzia di anonimato, ne fanno i suoi punti di forza per l'utenza.
Una tale piattaforma, però, ha bisogno di eseguire molte operazioni, alcune dipendenti tra loro, altre indipendenti e contemporanee, ma, questa molteplicità di attori in gioco rende necessaria e inevitabile la presenza di arbitri, i quali garantiscano un corretto e affidabile progredire dell'elaborazione: questi arbitri sono i sistemi di sincronia.

Indice

Introduzione

L'introduzione dei sistemi operativi nel mondo dell'informatica ha rappresentato un momento di svolta, perchè, si è passati dalla concezione di *macchina sequenziale* (batch machine) a quella di *macchina multi-tasking*. Una macchina multi-tasking offre la possibilità di eseguire più flussi di istruzioni "contemporaneamente" sfruttando una stessa, e molto spesso unica, unità di elaborazione. Questa nuova potenzialità ha permesso, fin da subito, di aumentare l'efficienze nell'uso dell'unità elaboraiva e, nel corso degli anni, anche lo sviluppo di applicazioni alla portata di utenti non esperti, in quanto, grazie al multi-tasking, era possibile interagire con l'utente in modo sempre più stretto. La piattaforma *PariPari*, di cui si parlerà nell'elaborato, è inserita in questo contesto, perchè mira a fornire una moltitudine di servizi che l'utente può decidere di usare, nella misura e quantità a lui più consone, sfruttandoli tutti assieme, senza per forza dover attendere la fine di uno, per eseguirne un altro.

Negli applicativi concorrenti[1], le istruzioni non sono più tutte rigidamente collegate da un legame di consequenzialità: per questo motivo, quindi, necessitano di accorgimenti particolari per evitare che l'elaborazione possa prendere strade impreviste. A tal scopo sono stati sviluppati modelli e strumenti che permettono di descrivere i sistemi multi-tasking, facilitandone l'implementazione e la verifica della correttezza. In questo elaborato si procederà con l'analisi del profilo di concorrenza del plug-in PariDHT relativo alla piattaforma *PariPari* e, dopo averne verificato la correttezza, verrà proposta una soluzione per i problemi di sincronia riscontrati. I capitoli si dipanano, quindi, nel seguente modo:

- nel Capitolo 1 vengono presentati la piattaforma *PariPari*, la sua architettura, e il plug-in PariDHT;
- il Capitolo 2 fornisce un'infarinatura sulla teoria della concorrenza;
- nel Capitolo 3 si passa all'analisi del plug-in PariDHT sotto il profilo del multi-tasking, evidenziando da una parte gli errori di concorrenza e dall'altra le inefficenze dovute a una bassa sincronia;
- il Capitolo 4 propone delle soluzioni per risolvere i problemi riscontrati, ponendole in modo da essere fruibili anche al di fuori del plug-in in esame;
- nel Capitolo 5, infine, vengono presentati dei test che mirano a valutare l'efficenza degli strumenti di sincronia adottati.

[1] Dal latino CONCÚRRERE ovvero correre assieme, sono applicativi dove più attori cooperano tra loro per arrivare ad uno scopo prefissato.

Capitolo 1

PariPari

PariPari è un'applicazione che, attraverso una rete distribuita di tipo P2P, offre dei servizi sia agli utenti che ne fanno parte, sia a computer esterni. Questi servizi spaziano da quelli propri delle reti P2P come la condivisione di file, a quelli già disponibili nella "tipica" internet ovvero DNS, web server, posta elettronica e molti altri, ma uno dei punto di forza di questa piattaforma è la loro disponibilità tramite un unico programma, avviabile comodamente da browser. Lo sviluppo di *PariPari*, cominciato 5 anni fa, è stato finora portato avanti da studenti dell'Università di Padova che hanno apportato il loro contributo anche con tesi ed elaborati, sia per la laurea triennale che specialistica; questa base di più di 40 persone è suddivisa in gruppi ognuno con il compito di seguire un "modulo" dell'applicazione. Il linguaggio scelto per il codice è Java™ : la sua indipendenza dal sistema operativo garantisce una agevole distribuzione dell'applicazione per tutte le architetture, inoltre, gli studenti padovani hanno già familiarità con questo linguaggio in quanto viene introdotto nei corsi dei primi anni di studio e questo facilita la formazione dei futuri sviluppatori.

1.1 L'architettura

Per poter organizzare al meglio le forze a disposizione e semplificarne lo sviluppo stesso, *PariPari* è stata progettata fin da subito come un'applicazione modulare: i moduli o *plug-in* sono sezioni di programma a se stanti che offrono un servizio specifico. Tra tutti i plug-in ce n'è uno particolare chiamato *Core* il quale, unico lanciabile stand-alone, si fa carico dell'avvio della piattaforma, di reperire e "installare" gli altri moduli e gestirne le intercomunicazioni e richieste. I restanti plug-in a loro volta sono suddivisi in cerchia interna ed esterna. Nella cerchia interna troviamo tutti i servizi fondamentali per l'utilizzo di *PariPari*:

connectivity si incarica di centralizzare gli aspetti che riguardano la "connessione" (ad internet, ad esempio) della piattaforma, ovvero l'allocazione di socket, vincolati da costrizioni quali banda massima, massimo numero di socket richiedibili, ecc.;

dht è il modulo incaricato di creare la rete distribuita tra le varie istanze di *PariPari*: grazie ai suoi servizi, i plug-in hanno la possibilità di pubblicare o ricercare informazioni, senza la necessità di un server centralizzato;

local storage offre la possibilità di accedere alla memoria di massa, applicando opportuni vincoli sulle quantità e i luoghi in cui è possibile memorizzare dei file;

credits questo modulo rappresenta una vera innovazione in quanto instaura una sorta di mercato tra i plug-in dove quelli più performanti diventano anche più ricchi grazie ai servizi offerti agli altri plug-in e, acquisendo moneta virtuale, possono spenderla attraverso richieste agli altri plug-in.

La cerchia esterna invece raccoglie tutti i servizi "di alto livello" rivolti all'utente. I nomi di questi moduli, attualmente in via di sviluppo sono: torrent, database, distributed storage, distributed backup, DNS, login, mulo, VoIP, IM, IRC, NTP, WEB. Questa lista è destinata a crescere anche, e soprattutto, in seguito alla release ufficiale in quanto ogni programmatore potrà sviluppare plug-in personalizzati che rispondono alle proprie particolari esigenze.

1.2 La rete *PariPari* e il plug-in PariDHT

La rete che *PariPari* sfrutterà per l'offerta dei suoi servizi è di tipo P2P. Queste particolari reti si slegano dal classico modello *client-server* dove un singolo server fisico conosciuto serve le richieste di tutti i client, abbracciando un più flessibile modello basato su *nodi* paritari: un nodo non è altro che un generico partecipante della rete (ad esempio un'istanza di *PariPari*) che conosce un numero limitato di altri partecipanti e in alcune circostanze offre funzionalità di server, mentre in altre di client. Le reti P2P stanno venendo alla ribalta negli ultimi anni, da quando cioè le leggi degli Stati hanno cominciato ad essere più ferree in merito a ciò che può o meno circolare in Internet: ci si è accorti che il modello client-server soffriva del grosso svantaggio che, se per un qualche motivo il server centrale risultava irraggiungibile, non era più possibile creare la "rete"; ci si è mossi, perciò, verso il modello distribuito in cui la rete può formarsi e rimane operativa finchè almeno uno dei nodi partecipanti risulta raggiungibile. In una internet di questo tipo, però, non esistono erogatori di servizi noti a priori ed è quindi necessario un meccanismo per una loro ricerca rapida e dinamica: questo meccanismo è la DHT.

1.2.1 Cos'è una DHT

DHT è un acronimo per *Distributed Hash Table* e identifica un particolare sistema di immagazzinamento e reperimento delle informazioni che sfrutta una rete di calcolatori. Le informazioni che è possibile salvare all'interno di una DHT sono delle coppie <*chiave, valore*> dove la chiave identifica univocamente la risorsa, ovvero il valore stesso; la differenza rispetto a una normale tabella hash sta nel fatto che queste coppie <*chiave, valore*> sono distribuite il più possibile equamente tra tutti i partecipanti alla rete; ciò presenta un problema di riperimento delle informazioni condivise ma, a questo scopo, esistono degli algoritmi che, nel caso medio, raggiungono una complessità computazionale logaritmica nel numero di nodi.[1].

La prima particolarità di questa infrastruttura sta nel fatto che i nodi sono identificati all'interno della rete da un *ID* che condivide lo stesso dominio delle chiavi: questa accortezza permette di individuare i nodi *responsabili* di una determinata risorsa, ovvero quei partecipanti aventi ID "più vicino" alla chiave memorizzata. Parlando di vicinanza resta ancora da definire cosa intendiamo per nodi "vicini" o "lontani" e per fare questo introduciamo il concetto di *metrica*: una metrica è una funzione matematica $d : X \times X \to \Re$ tale che, per ogni $x, y, z \in X$, gode delle seguenti proprietà:

[1]Si parla di caso medio in quanto la complessità della ricerca è legata a fattori aleatori; nel caso peggiore essa può salire fino a toccare la linearità nel numero di nodi stessi.

- $d(x, y) \geq 0$
- $d(x, y) = 0 \iff x = y$
- $d(x, y) = d(y, x)$
- $d(x, z) \leq d(x, y) + d(y, z)$

I nodi responsabili di una determinata *chiave* sono quindi quei nodi tali che $d(ID, chiave)$ è minore possibile.

La definizione di DHT data finora manca ancora di un punto: l'*overlay network*. Per poter rendere la rete operativa è necessario che ogni nodo mantenga dei *contatti*, non casuali, con altri nodi, strutturando così la topologia della rete. Questa topologia deve godere della proprietà che, per ciascuna chiave k, o il nodo possiede k oppure conosce un nodo più vicino ad essa. Detta caratteristica permette, da qualsiasi nodo della rete, di spedire un messaggio al responsabile di una certa risorsa k, secondo un algoritmo che si ripete ad ogni hop della comunicazione; esso prevede che chi riceve il messaggio abbia due possibilità:

1. tutti i contatti a sua disposizione sono più distanti da k rispetto a se stesso, ciò ne fa automaticamente il responsabile della risorsa e può quindi consumare il messaggio;
2. identifica il nodo di cui è a conoscenza avente l'ID più vicino possibile a k, inoltrandogli il messaggio.

Una simile sovrastruttura è fondamentale per permettere il funzionamento della rete stessa, dato che, viene assunta vera da tutti gli algoritmi di salvataggio e reperimento delle informazioni.

1.2.2 PariDHT

Il plug-in di *PariPari* che si incarica di creare e gestire questa rete è *PariDHT*, sviluppato seguendo molto da vicino le orme di Kademlia[5]. Come dominio delle chiavi (e di conseguenza degli ID dei nodi) sono stati scelti i numeri naturali a 256bit: questa scelta è volta a minimizzare il numero di collisioni che possono avvenire nella scelta casuale dell'ID o nella generazione della chiave a partire da una risorsa, attraverso una funzione di hash. Una volta specificato il dominio, la funzione scelta per la metrica è stato l'operatore binario XOR, ovvero $d(x, y) = x \oplus y$ il cui risultato è interpretato come numero intero positivo a 256bit. Per mantenere i contatti con gli altri partecipanti, PariDHT utilizza una struttura dati composta da 256 insiemi di k nodi chiamati k-bucket: l'i-esimo k-bucket conterrà i nodi la cui distanza d dal nodo corrente è $2^i \leq d < 2^{i+1}$. Grazie alla figura 1.1 è più facile capire il perchè sia stata scelta questa regola per il mantenimento dei contatti; posto b il numero di bit:

- il b-esimo k-bucket conterrà k tra i possibili 2^{b-1} nodi situati nell'altra metà della rete
- il $(b-1)$-esimo k-bucket conterrà k tra i possibili 2^{b-2} nodi situati nella stessa metà ma nell'altro quarto della rete.
- il $(b-2)$-esimo k-bucket conterrà k tra i possibili 2^{b-3} nodi situati nello stesso quarto ma nell'altro ottavo della rete.
- ...

- il primo k-bucket potrà contenere un solo nodo ovvero quello avente solo il bit meno significativo diverso dal nodo corrente.

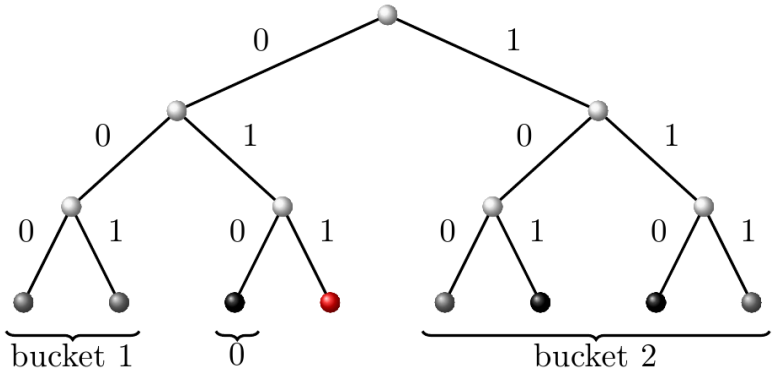


Figura 1.1: Albero della rete *PariPari* con nodi a 3bit situati sulle foglie[3]

La struttura topologica così creata permette di raggiungere una complessità computazionale per le operazioni di inserimento e ricerca, logaritmica nel numero di nodi, in quanto ogni hop della comunicazione tra mittente e destinatario ne dimezza la distanza. Quanto appena detto, però, è valido fintanto che ogni k-bucket contiene almeno un nodo valido, per questo motivo, nonostante $k = 1$ sia un valore sufficiente, viene di solito scelto $5 \leq k \leq 20$ per alleviare i problemi introdotti dal *churn*[2] dei nodi.

[2]Con churn si intende la disconnessione improvvisa di un nodo dalla rete che, agli occhi degli altri partecipanti, risulta nell'impossibilità di comunicare con il nodo disconnesso.

Capitolo 2

Concorrenza

In questo capitolo verranno definiti alcuni concetti chiave della programmazione concorrente per poter meglio comprendere le esigenze che il plug-in PariDHT presenta in questo contesto e capire le soluzioni adottate.

2.1 Processi multi-threaded

Ciascuno di noi, ogni qualvolta accede a un PC, è solito avviare una serie di programmi quali un browser internet, un programma di posta, piuttosto che un foglio di calcolo: questa molteplicità di programmi sembrano lavorare "tutti assieme" sul nostro sistema, condividendone risorse fisiche e virtuali, sotto la guida del sistema operativo, un ambiente ad alta concorrenza. In informatica con il termine *concorrenza* si identifica la caratteristica dei sistemi nei quali più processi stanno progredendo in maniera simultanea, potenzialmente interagendo tra loro[1]. Se guardiamo per un momento all'hardware dei moderni calcolatori però, ci accorgiamo che solo gli ultimi anni hanno portato la diffusione di processori multi-core: in precedenza l'utenza domestica lavorava unicamente su single-core, ma come poteva il sistema operativo gestire più programmi allo stesso tempo? Questa contemporaneità in realtà è solo una parvenza, resa possibile dalla gestione della risorsa processore da parte del sistema operativo. Nella pratica il processore (o i processori nei sistemi multi-core) è assegnato ad ogni *processo* per una certa porzione di tempo, esaurita la quale viene sospeso ed un altro prende il suo posto: questa porzione di tempo o *time-slice* è quantificata in modo da permettere al programma di portare avanti i suoi compiti e all'utilizzatore di non accorgersi di questi continui cambi di "processo corrente".

Abbiamo finora parlato di processi senza fornire una loro definizione: un processo è un'istanza di un programma[2] in esecuzione, è composto, oltre che dalla sequenza di istruzioni da eseguire, da tutte le informazioni necessarie al sistema operativo per tracciarne le attività e poterlo sospendere/riprendere in sicurezza, ovvero senza che il normale corso del processo venga alterato da questi *context-switch*. Fra le informazioni contenute, il sistema operativo tiene traccia anche di tutti i thread che il processo ha generato: un thread è il risultato della suddivisione di un processo in più tronconi esecutivi, i quali possono essere svolti concorrentemente secondo le politiche di scheduling del sistema operativo[3]. Un thread esiste, di solito, solo contestualmente a un processo, con il quale

[1] [2, `http://en.wikipedia.org/wiki/Concurrency_(computer_science)`]

[2] Un programma è inteso come una sequenza di istruzioni che permettono a un computer di eseguire una qualche mansione [2, `http://en.wikipedia.org/wiki/Computer_program`]

[3] [2, `http://en.wikipedia.org/wiki/Thread_(computer_science)`]

condivide la memoria utilizzata, il codice esecutivo, i descrittori di file ed i permessi; questa condivisione permette ai thread di uno stesso processo di:

- dialogare tra loro in maniera diretta sfruttando la memoria condivisa; questo tipo di interazione, tra processi, è strettamente proibito dal sistema operativo;

- effettuare context-switch molto rapidi in quanto gli unici dati da salvare sono quelli relativi allo stato dei registri del processore; un processo invece deve aggiornare tutte le informazioni viste in precedenza: per questo motivo i thread sono anche detti *lightweight-process*.

Questa breve digressione tra thread e processi, chiamati in genere *task*, era volta a comprendere i vantaggi e le possibilità del modello di programmazione multithreading, cardine della programmazione concorrente in Java™ che ora andremo a delineare.

2.2 Programmazione concorrente

In questo elaborato, si parlerà principalmente di programmazione concorrente[4] ovvero di tutto l'insieme di concetti e algoritmi che permettono la relalizzazione di un sistema concorrente. La programmazione concorrente si fonda sui principi di *correttezza*, *performance* e *robustezza*:

correttezza indica il rispetto di tutti i vincoli e le specifiche imposte, in qualsiasi momento dell'elaborazione;

performance si intende il tentativo di rendere meno dispendioso possibile, in termini di tempo di elaborazione, l'apparato necessario a coordinare tutti gli attori in gioco;

robustezza indica la capacità del codice di gestire gli errori o le situazioni critiche che si possono presentare durante l'esecuzione.

La programmazione concorrente è di grande aiuto nello sviluppo di un'applicazione in quanto permette di parallelizzare il lavoro o di demandare dei compiti specifici a dei thread paralleli che li possano svolgere concorrentemente al thread principale. Un esempio di suo utilizzo potrebbe essere il correttore ortografico di un qualsiasi edito di testi: mentre un thread acquisisce i caratteri digitati dall'utente, un altro si incarica di individuare eventuali errori ortografici ed un terzo aggiorna lo schermo in modo da mostrare i cambiamenti avvenuti.

Quello che finora potrebbe non esser chiaro è il perchè, quando abbiamo a che fare con più thread, sia necessario dotarsi di precauzioni. Il motivo può essere espresso attraverso un esempio che coinvolge la classe `Contatore` scritta in Java™ che, ad ogni invocazione del metodo `count()`, restituisce un numero intero incrementale. Cerchiamo ora di descrivere cosa potrebbe accadere se due thread che condividono il medesimo oggetto `Contatore` invocano nello stesso istante il metodo `count()`:

[4] Da non confondere con la *programmazione parallela* che mira a suddividere un problema in sottoproblemi ognuno dei quali può venir risolto (o semplificato) indipendentemente dagli altri [2, `http://en.wikipedia.org/wiki/Parallel_computing`]

Listing 2.1: Contatore non thread-safe

```
public class Counter {
  private long counter = 0;

  public int count() {
    return counter++;
  }
}
```

Thread 1	**Thread 2**
• Lettura del valore della variabile: `counter` vale `0`	
• Incremento del valore: `0 + 1 = 1`	
	• Lettura del valore della variabile: `counter` vale `0`
	• Incremento del valore: `0 + 1 = 1`
	• Scrittura della variabile: `counter` ← `1`
• Scrittura della variabile: `counter` ← `1`	
• Ritorno del valore incrementato: ritorno del valore `1`	
	• Ritorno del valore incrementato: ritorno del valore `1`

Come si può vedere da questo esempio, se durante l'esecuzione capitasse questo malaugurato ordine nelle operazioni, il contatore non si comporterebbe più *correttamente* in quanto non rispetterebbe le specifiche precedentemente descritte. Grazie a quanto appena detto, possiamo ora definire più agevolmente il concetto di *thread-safety*: una classe si dice *thread-safe* se si comporta correttamente quando più thread vi accedono, senza preoccuparsi degli interventi dello scheduler o dei context-switch prodotti dall'ambiente di esecuzione e senza altra sincronizzazione necessaria da parte del codice chiamante[4]. La domanda che potrebbe sorgere ora è: quando più thread accedono ad uno stesso oggetto, quest'ultimo deve essere sempre dotato di un arbitro che ne regoli gli accessi? La risposta è no: perchè possano sorgere problemi di concorrenza, deve esitere almeno uno stato condiviso tra più task, e, almeno uno di questi, deve accedere in scrittura allo stato (stato mutabile) senza una sincronizzazione adeguata. Queste due condizioni permettono di catalogare automaticamente come thread-safe le classi immutabili (i cui stati interni sono invarianti) o le classi prive di stato (ad esempio una classe contenente solo metodi che agiscono esclusivamente sui parametri in ingresso).

Se una classe non è thread-safe, i problemi in cui può incorrere sono raggruppati in due grandi insiemi: *race conditions* e *data races*. Una race condition si verifica quando la correttezza dell'elaborazione, dipende dalle tempistiche relative o dall'alternarsi di più thread, durante l'esecuzione[4]: in altre parole, ottenere il risultato corretto è una questione di fortuna, imputabile al non verificarsi di alcuna situazione critica, nelle tempistiche di esecuzione. Esempi di codice non thread-safe che afferiscono a questo insieme sono il *check-then-act* in cui l'eseguire una determinata operazione è conseguenza di un test su uno stato mutabile o il *read-modify-write* in cui una variabile viene letta, modificata e poi scritta, spiegati con due esempi in tabella 2.1. Una data race, invece, si verifica quando l'infrastruttura di gestione della concorrenza non è usata per coordinare tutti gli accessi a un dato stato condiviso: si rischia una data race se un thread scrive una variabile che deve

Tabella 2.1: Esempi di race condition

```
private instance = null

public Object getInstance() {
  if (instance == null) {
    instance = new Object();
  }

  return instance;
}
```

Questo esempio mostra il caso di un check-then-act. I vincoli di questo tipo di codice dicono che l'oggetto `instance` deve venir inizializzato una volta sola e deve essere lo stesso per chiunque invochi il metodo `getInstance()`: la non-correttezza di questo codice può essere dimostrata immaginando che due thread eseguano il controllo `instance == null` nello stesso momeno avviandosi, quindi, ad inizializzare due oggetti distinti che saranno poi ritornati.

```
private long counter = 0;

public int count() {
  return counter++;
}
```

Questo esempio mostra il caso di un read-modify-write. I vincoli e le problematiche di questo codice sono già state delineate nell'esempio del contatore e consistono nel fatto che `counter++` viene di fatto suddiviso in tre operazioni distinte: una lettura, una modifica e una scrittura.

essere successivamente letta da un altro thread o se un thread legge una variabile che potrebbe essere stata scritta da un altro thread ed entrambi non sono stati sincronizzati[4]. Un problema che può originare una data race è il cosiddetto *out-of-thin-air safety* ovvero la mancanza di visibilità, da parte di un thread, di una variabile modificata da un altro thread: un problema di questo tipo dipende dalle politiche di caching dell'ambiente di esecuzione dove, per velocizzare l'esecuzione dei thread, possono venir fatte delle copie locali di alcune variabili, le cui modifiche vengono rispecchiate nella variabile originaria solo quando l'ambiente stesso lo reputa opportuno.

2.3 Pattern di base per la programmazione concorrente

Si è discusso finora dei problemi che un programmatore non attento può incontrare scrivendo del codice multi-threaded; in questa sezione verranno invece descritti gli strumenti a disposizione per gestire i thread e scrivere dei programmi concorrenti corretti.

La programmazione concorrente, forse sorprendentemente, si avvale unicamente di due costrutti fondamentali: i thread, di cui si è già discusso, e i *locks*. Un lock è un costrutto offerto dai linguaggi che permettono una programmazione concorrente che, attraverso due direttive `lock()` e `unlock()`, permette un *accesso esclusivo* alle direttive contenute al loro interno. L'accesso esclusivo (o in mutua esclusione) garantisce ad un thread A che si trovi ad eseguire una di queste direttive, che nessun altro attore concorrente B potrà eseguirne alcuna finchè A non uscira dal blocco esclusivo, eseguendo una `unlock()`. Quando un thread T_1 incontra una direttiva `lock()` si dice che *tenta di acquisire* il lock L; quello che fa è un tentativo, in quanto L potrebbe già esser occupato da un altro thread T_2: in questo caso l'unica opzione a sua disposizione è aspettare che T_2 liberi il lock attraverso la direttiva `unlock()`. Una volta acquisito L, T_1 ha la garanzia che nessun'altro thread

possa eseguire alcuna istruzione protetta dallo stesso lock, fintanto che esso non invoca la direttiva `unlock()`.

Tabella 2.2: Possibile implementazione delle direttive `lock()` e `unlock()`

```
function lock()
  // aspetto che il lock si liberi
  while(lock_flag)

  // il lock ora e' libero

  // occupo il lock
  lock_flag = true
```

```
function unlock()
  // rilascio il lock
  lock_flag = false
```

Per capire più a fondo la struttura di un lock, in tabella 2.2 è presentata una possibile implementazione delle sue direttive: da questi due pseudo-codici salta subito all'occhio che il loro succo sta semplicemente nel controllare o impostare un flag. Nulla vieta, quindi, di usare flag diversi per creare lock indipendenti tra loro: le direttive possono venir ridefinite in `lock(f)` e `unlock(f)` dove `f` è il flag da acquisire. Osservando il codice della `lock()`, però, ci si accorge che esso è una forma camuffata di check-then-act: le istruzioni di controllo del flag e sua impostazione non sono atomiche. Questo porta a pensare che all'interno della definizione di un lock di alto livello (linguaggio di programmazione), serva nuovamente[5] un lock, magari di livello inferiore (sistema operativo): agendo in questo modo, però, è necessario che il "livello più basso" offra autonomamente una direttiva `lock()` *atomica* da poter sfruttare ai livelli superiori. Il concetto di atomicità di una o più istruzioni può essere intuito pensando a queste operazioni come indivisibili nel senso di tutte-o-nessuna; una definizione più formale invece spiega: due operazioni A e B si dicono atomiche l'una rispetto all'altra se, rispetto a un thread che sta eseguendo A, quando un altro thread esegue B allora o tutta B è stata eseguita, oppure non ne è stata eseguita alcuna parte; un'operazione si dice atomica se è un'operazione atomica rispetto a tutte le altre, inclusa se stessa[4]. Nella realtà, quindi, è necessario che la `lock()` avvenga atomicamente rispetto a qualsiasi altra operazione dell'intero sistema e questa garanzia può essere messa a disposizione solo dall'hardware o più precisamente dalla CPU. Nei sistemi single-core l'unità centrale di elaborazione fornisce un lock esplicito attraverso la disabilitazione delle interruzioni[6]: in questo modo è impedito qualsiasi context-switch e il thread corrente sarà l'unico a poter operare fino a quando non riabiliterà nuovamente le interruzioni. Nei sistemi multi-core, invece, la disabilitazione delle interruzioni riguarderebbe un singolo core e niente vieterebbe agli altri di caricare thread interferenti, per questo motivo questi sistemi offrono delle istruzioni atomiche[7] *test-and-set* (TAS) o *compare-and-swap* (CAS) che scrivono nella locazione di memoria fornita un dato valore solo se questa contiene lo stesso valore. La differenza in questi due tipi di istruzioni sta nel fatto che una TAS solitamente opera su un solo bit e ha come unico parametro in ingresso la locazione di memoria, la quale verrà impostata al valore *true*, ritornando il contenuto precedente; una CAS invece opera solitamente su 32 bit, ha come parametri di

[5] Altrimenti, così com'è stata presentata, l'implementazione sarebbe sbagliata dato che il controllo e l'impostazione del flag non avvengono in mutua esclusione.

[6] Le interruzioni (o interrupt, in inglese) sono un sistema di segnalazione asincrono che permette alla CPU di gestire eventuali richieste dell'hardware interrompendo temporaneamente l'esecuzione del programma in corso e passando ad un altro che possa occuparsi della periferica stessa.

[7] L'atomicità è garantita anche rispetto agli altri core

ingresso la locazione di memoria, il valore di confronto e il nuovo valore ed è per questo più versatile della test-and-set riuscendo a risolvere il problema del consenso[8] per un numero arbitrario di processori, invece di solamente 2[1]. Nonostante queste operazioni non siano un lock esplicito, permettono la loro costruzione a livelli superiori, come si può vedere nel codice 2.2

Listing 2.2: Lock costruito attraverso un'istruzione CAS

```
/* si supponga di avere un'istruzione CAS definita come:
*   compare_and_swap(location, compare_value, new_value)
*/

function lock(location)
  //attendo fino ad ottenere il lock
  while(compare_and_swap(location, 0, 1) == 1);

  // ho acquisito il lock
```

Un ultimo aspetto dei lock sul quale è bene far luce è la *rientranza* ovvero: cosa fare se un task già in possesso del lock f tentasse nuovamente una `lock(f)`? Se il lock non fosse rientrante, il caso descritto costringerebbe il task a rilasciare il lock per poi doverlo riacquisire un attimo dopo ma, questa seppur piccola finestra temporale, permetterebbe ad un altro thread di acquisire f: ciò comporta una nuova eventualità di cui tener conto, nello sviluppo dell'applicazione. Un lock rientrante, d'altra parte, è in possesso anche di un contatore incrementato dopo ogni `lock()` e decrementato prima di ogni `unlock()`: un thread perde il possesso del lock solo se l'`unlock()` che si appresta ad eseguire porterebbe il contatore a zero. Attraverso un lock rientrante, quindi, il task dell'esempio potrebbe tranquillamente affrontare l'entrata nel nuovo blocco di mutua esclusione senza mai perdere il possesso del lock stesso, il quale verrà rilasciato solo dopo tante `unlock()` quante sono state le `lock()` attraversate.

2.4 Semafori, monitor e problemi di accesso a risorse multiple

A partire dal concetto di lock, sono stati sviluppati altri costrutti di sincronizzazione più versatili e potenti; in questa sezione verranno analizzati i due più importanti storicamente: *semafori* e *monitor*.

I semafori, teorizzati dall'informatico olandese Edsger W. Dijkstra, sono degli oggetti dotati di una variabile intera interna, di due metodi `P` e `V` e di un insieme di thread in attesa. Un semaforo, così come un lock, serve a regolare l'accesso a una porzione di codice o, più tipicamente, a una risorsa, da parte di più thread ma, stavolta, non necessariamente in mutua esclusione. Quando un task esegue una `P` sul semaforo, ovvero cerca di ottenere l'accesso alla risorsa, specifica il numero di "permessi" che vuole acquisire; il semaforo garantirà l'accesso alla risorsa solamente se il valore della sua variabile interna è maggiore o uguale ai permessi richiesti, decrementando suddetta variabile di conseguenza; in caso contrario, ovvero se il semaforo non contiene un numero sufficiente di permessi per il thread richiedente, questo verrà inserito nell'insieme di attesa. Una chiamata di

[8] Il problema del consenso permette di capire se, avendo due costrutti di sincronizzazione X e Y, sia possibile una simulazione degli oggetti Y, senza attese arbitrarie (wait-free), utilizzando solamente costrutti X e registri di lettura/scrittura atomici.

tipo `V`, d'altra parte, serve a rilasciare il numero di permessi passati come parametro[9], incrementando di conseguenza la variabile interna del semaforo; se questo incremento soddisfa le richieste di permessi di alcuni task nell'insieme di attesa, essi verranno rimossi dall'insieme e l'accesso alla risorsa sarà loro garantito, previa diminuzione della variabile interna.

I monitor, inventati da C. A. R. Hoare e Per Brinch Hansen, sono oggetti in cui ogni metodo viene eseguito in mutua esclusione rispetto agli altri: un dato task ha la possibilità di eseguire un metodo solo se nessun'altro thread sta eseguendo un metodo dello stesso oggetto, in caso contrario viene inserito in una coda di attesa relativa all'oggetto stesso. Questa caratteristica permette di semplificare notevolmente lo sforzo di implementare il monitor in sè, anche se deve essere preceduta da un'accorta definizione degli oggetti che fungeranno da monitor per non far decadere le prestazioni dell'intero sistema.

Questi due costrutti, più il lock visto precedentemente, non devono esser considerati una panacea per tutti i problemi di concorrenza, soprattutto quando si parla di risorse multiple: se il programmatore di sistemi concorrenti non presta particolare attenzione all'uso che fa di questi strumenti potrebbe trovarsi ad affrontare complicazioni quali *deadlocks*, *livelocks*, *starvations* difficili da diagnosticare e riprodurre.
Un deadlock, o stallo, è una condizione in cui due o più task possono venire a trovarsi se sussistono tutte e 4 le seguenti condizioni:

1. mutua esclusione: almeno una delle risorse del sistema non è condivisibile;
2. allocazione parziale: i processi che già detengono delle risorse, possono richiederne delle altre in un secondo momento;
3. non sottraibilità: solo il processo che detiene una risorsa è in grado di rilasciarla;
4. attesa circolare: due o più processi formano una catena circolare in cui ogni processo è in attesa di una risorsa che il task successivo detiene.

In pratica un insieme di processi si dice in stallo se ogni processo dell'insieme aspetta un evento che solo un altro processo dell'insieme può provocare. Questa condizione di stasi può essere prevenuta inficiando almeno una delle precedenti condizioni, ma, un suo verificarsi, comporta necessariamente la distruzione di tutti i task in gioco e di quelli che da essi dipendono.
Parallelamente al deadlock e come caso particolare di una starvation esiste il livelock in cui gli stati dei processi coinvolti cambiano continuamente ma in maniera tale che nessuno di essi progredisca nell'elaborazione.
Il breve elenco di problemi di concorrenza si conclude con la resource starvation: un singolo processo può trovarsi nella condizione di attendere il rilascio di una o più risorse ma, pur potendosi verificare la disponibilità delle stesse, questo viene sopravanzato da altri task che ne richiedono meno, lasciandolo in questo stato di blocco per un periodo *indefinito*. Un errore progettuale di questo tipo si può verificare quando, in un semaforo, esistono pochi task lenti che richiedono decine di permessi e molti task rapidi che invece ne richiedono solo uno: se non esiste un'adeguata gestione dell'insieme dei processi in attesa sul semaforo (ad esempio attraverso una coda), è *molto probabile* che i pochi processi lenti e ingordi subiscano starvation da parte dei molti processi più economici.

[9] Il modello del semaforo non vincola in alcun modo i thread a rilasciare tanti permessi quanti ne hanno acquisiti, anzi, non obbliga nemmeno ad eseguire le operazioni `P` e `V` accoppiate.

Capitolo 3

Multi-threading in PariDHT

In un'applicazione multi-threaded gestire la concorrenza significa all'incirca raggiungere due obiettivi: da una parte tutte le classi sottoposte a concorrenza devono garantire la thread-safety, dall'altra i thread che interagiscono tra loro devono raggiungere il più elevato grado di sincronia possibile. Il primo obiettivo deve essere considerato prioritario, in quanto mancarlo, ovvero scrivere codice *thread-unsafe*, solitamente porta ad errori in esecuzione difficili da diagnosticare e da ricondurre ai giusti responsabili; inoltre, una bassa sincronizzazione tra i vari task non potrà mai portare ad errori, tuttavia le performance dell'intero sistema risulteranno tanto più penalizzate quanto più i task a non essere sincronizzati sono usati durante la normale vita dell'applicazione.
Verranno ora analizzati gli aspetti relativi a questo contesto, in riferimento all'attuale implementazione del plug-in PariDHT.

3.1 PariDHT, un plugin di PariPari

Per capire fino in fondo lo scenario che descriveremo e le eventuali porzioni di codice di spiegazione, è utile ricordare che l'ambiente in cui si lavorerà è un plug-in della piattaforma *PariPari*. Questa piattaforma, come già detto, è scritta in Java™ e per cercare di proteggersi da eventuali plug-in malevoli è dotata di un sistema di controlli che permettono o impediscono ai plug-in determinate azioni. Una delle operazioni rigorosamente proibite consiste nella creazione arbitraria di oggetti `Thread`. Per evitare che plug-in troppo esigenti riescano a bloccare l'intero sistema con thread inutili, ognuno di essi è dotato di un thread-pool gestito direttamente da Core: quando un plugin ha bisogno di eseguire un determinato blocco di istruzioni, in parallelo rispetto al thread principale, deve fare una "prenotazione di esecuzione"; Core è incaricato di raccogliere tutte queste prenotazioni, ed eseguire il codice relativo, su un thread libero del pool assegnato al plugin, attendendo, nel caso fossero tutti occupati. Andrebbero quindi lette sempre in quest'ottica, le porzioni di codice simili a quella presentata nel listato 3.1, ma, grazie alle classi `PariPariRunnable` e `PariPariThread`, Core stesso esegue un'ottima astrazione dal sistema di prenotazioni, permettendo di fatto di continuare a parlare di `Runnable` e `Thread`. Nel linguaggio Java™ la classe `Thread` rappresenta lo stesso concetto di thread già descritto. L'interfaccia `Runnable`, invece, definisce il codice che dovrà essere eseguito da un thread. La relazione di cardinalità tra questi due attori è di tipo N:M in quanto un `Thread` può eseguire più `Runnable`[1] ma pure un dato `Runnable` può essere eseguito da più `Thread`.

[1] Si pensi ad una possibile implementazione di un timer in cui a un thread vengono assegnati molteplici task da eseguire.

Listing 3.1: Definizione di un PariPariRunnable e avvio del PariPariThread che lo esegue

```
class MyRunnable extends PariPariRunnable {
  @Override
  public void go() {
    // codice eseguibile di MyRunnable
  }
}

public void startNewMyRunnable() {
  PariPariRunnable r = new MyRunnable();
  // creo il thread che eseguira' MyRunnable
  PariPariThread t = new PariPariThread(r);
  // avvio il thread creato
  t.start();
}
```

La scelta di Java™ come linguaggio di programmazione apre la strada anche all'utilizzo della sua libreria standard messa a disposizione degli sviluppatori. Essa contiene classi e interfacce che permettono di non dover ogni volta reinventare la ruota: ADT, gestione astratta di stream dati, creazione di interfacce utente e tantissime altre, sono le possibilità che mette a disposizione. Si vorrebbe metter in luce, però, il package `java.util.concurrent` introdotto dalla versione 1.5 della libreria[2]: ivi sono contenuti una serie di strumenti di aiuto per i programmatori di sistemi concorrenti come ad esempio ADT concorrenti, code bloccanti, semafori, lock, e tipi di dato atomici.

3.2 Profilo di concorrenza di PariDHT

PariDHT è un plug-in che deve gestire una doppia molteplicità di richieste: da una parte, essendo nodo di una rete P2P, deve mantenere i legami con questa rete rispondendo in modo appropriato a tutti i pacchetti che da essa provengono, dall'altra, essendo plug-in di una piattaforma, dovrà essere in grado di svolgere tutte le mansioni che gli verranno commissionate dalla piattaforma stessa. Un primo profilo della concorrenza in PariDHT è descritto in figura 3.1 e parla di task di *servizio*, task *lavoratori* e classi *condivise*: tutti questi task dialogano tra loro e scambiano informazioni con classi condivise da tutto il plug-in, creando un sistema la cui attuale implementazione verrà ora analizzata, lasciando invece al capitolo successivo la spiegazione delle modifiche apportate.

3.2.1 Classi condivise

Nella descrizione iniziale di una DHT generica, si è detto che ogni nodo deve mantenere una serie di informazioni che permettono la costruzione della rete stessa: per questa ragione, poichè all'interno del plug-in queste informazioni vengono aggiornate e gestite, in maniera concorrente, da vari attori, si è stabilito di dar loro un'ubicazione nota, precisamente, all'interno delle classi qui di seguito descritte.

`ConfigFile` Classe contenente, non tanto informazioni strettamente necessarie al funzionamento della DHT, quanto, piuttosto tutti i parametri di configurazione necessari al plug-in. É sottoposta a concorrenza in quanto tutti i task vi fanno riferimento per ottenere i valori tipici del loro funzionamento, ma non sono presenti misure

[2]Questa versione è quella ufficiale di sviluppo

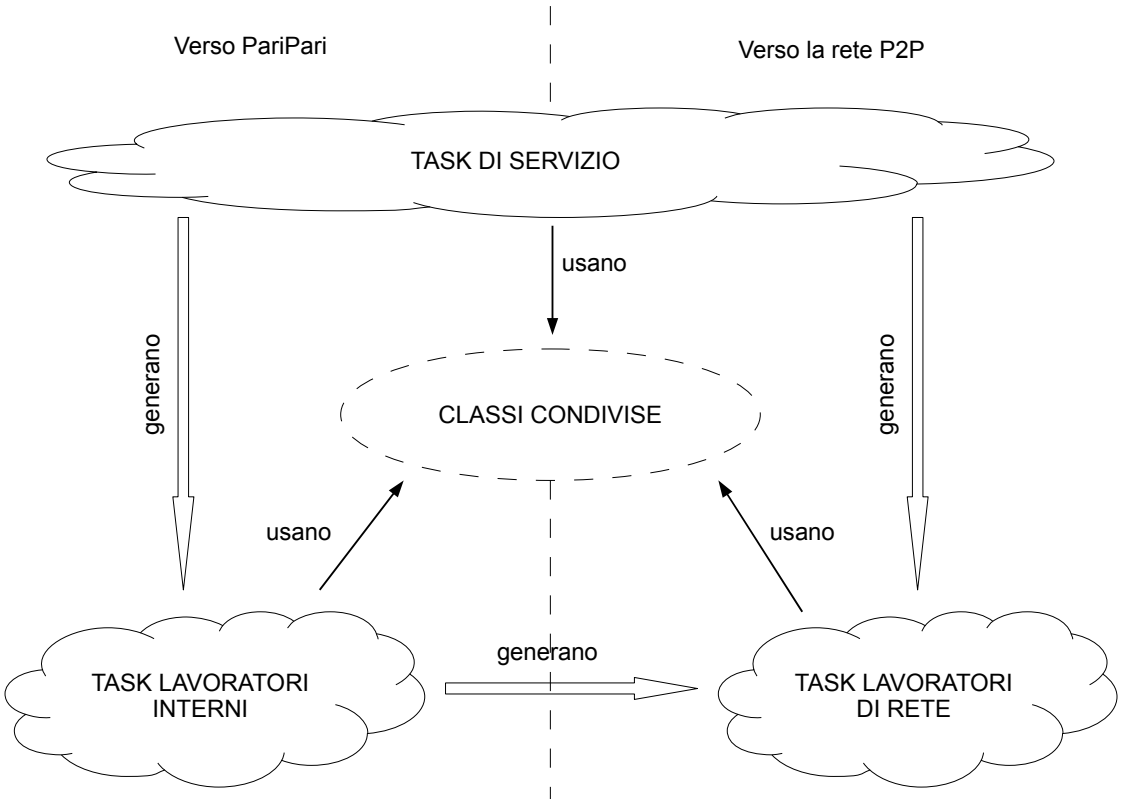


Figura 3.1: Profilo di concorrenza grezzo per PariDHT

per gestire questi accessi contemporanei. A motivo del fatto che non sia mai stato necessario prendere precauzioni in merito a questi aspetti, si può ipotizzare che, tra l'inserimento di un dato di configurazione e la sua lettura, di solito passa un tempo sufficientemente lungo tale per cui la cache di memoria della JVM viene aggiornata per tutti i thread. Inoltre, generalmente, una volta inserito un parametro di configurazione, esso viene quasi esclusivamente letto e, per di più, non vi sono effetti "visibili" dati da eventuali cambiamenti nei valori ritornati. Queste rimangono comunque congetture in quanto, "se più thread accedono alla stessa variabile di stato mutabile senza un'appropriata sincronizzazione, il programma è semplicemente rotto"[4].

`KeyStorer` Questa classe mantiene le coppie <*chiave, valore*> di cui il nodo corrente è responsabile. Il thread che inserisce una di queste coppie, tuttavia, non è mai lo stesso che le richiede nuovamente, perciò è necessario gestire la concorrenza di questa classe. Il profilo della concorrenza, però, risulta negativo anche in questo caso in quanto la classe non è dotata di nessuna accortezza.

`Node` La classe `Node` rappresenta il concetto di nodo di una rete P2P. É stata implementata come classe immutabile, ovvero contiene tutte le informazioni atte a identificare il nodo nella rete, ma ognuna di esse può venir solamente letta. Questa peculiarità la rende automaticamente thread-safe.

`NodeStorer` Il `NodeStorer` è la struttura che mantiene i k-bucket che contengono i contatti agli altri nodi della rete. É per definizione sottoposto a concorrenza, in quanto, il thread che vi inserisce o aggiorna i nodi, difficilmente sarà l'unico ad usarli; perciò,

per regolare gli accessi agli stati modificabili, è stato scelto un approccio basato su monitor rendendo mutualmente esclusiva l'esecuzione dei metodi di questa classe. Questa scelta, unitamente al fatto che la classe scambia solo oggetti immutabili e che gli stati interni non sono esposti, garantisce la thread-safety.

`PendingPacket` Questa classe descrive una richiesta inviata sulla rete e che non ha ancora ricevuto risposta. Permette uno scambio asincrono di informazioni tra chi ha effettuato la richiesta e chi riceve la risposta, ovvero due task distinti; tuttavia, gli stati condivisi non sono adeguatamente protetti e questo può originare problemi di data race.

3.2.2 Task di servizio

Con "task di servizio" si intende quella categoria di thread necessari al funzionamento stesso del plug-in. In questo momento vi sono 4 classi che appartengono a questa categoria: `Dispatcher`, `NetSender`, `NetReceiver` e `PeriodicThread`, ognuno dei quali è istanziato una sola volta ed usato da tutto il plug-in.

`Dispatcher` Questo thread è sempre in ascolto delle richieste che la piattaforma *PariPari* fa al plug-in PariDHT. Queste richieste sono quelle proprie di una DHT e quindi sono ricerche di chiavi o inserimenti di coppie <*chiave, valore*>. Da un punto di vista della concorrenza, non sono presenti problemi in quanto l'unico stato posseduto è una lista bloccante in cui vengono inserite le richieste dell'intera piattaforma, che `Dispatcher` provvederà a togliere ed espletare.

`NetSender` Come si può intuire dal nome, questo task è incaricato di inviare sulla rete fisica, tramite un socket, tutti i pacchetti necessari; anche qui l'analisi della concorrenza si concentra sui suoi due stati condivisi: una tabella hash concorrente contenente le richieste effettuate dal plug-in di cui non si è ancora ricevuta una risposta ed una coda bloccante[3] di pacchetti da inviare, che viene usata da tutte le classi del plug-in che hanno bisogno di inviare dei pacchetti sulla rete. Quando un pacchetto viene inserito in questa lista, `NetSender` lo preleva e, se esso è una richiesta, appena prima di inviarlo, lo aggiunge alla tabella delle richieste pendenti, altrimenti lo immette direttamente nella rete. Questo tipo di comportamento simula il modello produttore-consumatore, nel caso in cui sono presenti più produttori: quindi, affinchè non vi siano problemi di concorrenza è necessario che l'oggetto, condiviso tra produttori e consumatore, utilizzato per scambiare le informazioni, sia thread-safe, e ciò è garantito dal fatto che viene usata una `LinkedBlockingQueue` creata per questo scopo[4].

`NetReceiver` In ascolto su un socket, il suo compito è quello di ricevere i pacchetti provenienti dalla rete, farne un veloce controllo di validità e creare una `InRequest` per gestirli opportunamente.

`PeriodicThread` All'interno di PariDHT vi sono delle operazioni da ripetere nel tempo ad intervalli regolari. Per svolgerle, è stata preposta una classe specifica, il cui funzionamento è simile a quello di un timer: vi è un thread che controlla ad intervalli regolari una lista di operazioni da eseguire, e quando il tempo di attesa di uno di

[3] Una coda bloccante è una struttura dati in cui gli utilizzatori possono venire temporaneamente sospesi se si verifica una delle seguenti due condizioni: viene tentato un inserimento in una coda piena o una rimozione da una coda vuota.

[4] [6, `LinkedBlockingQueue` API]

questi task si esaurisce, quest'ultimo viene eseguito dallo stesso thread controllore. Questo oggetto è sottoposto a concorrenza in quanto i thread che inseriscono i task sono molteplici e solitamente diversi dall'unico che li esegue; non è però presente alcuna gestione degli accessi alla lista condivisa, rendendo l'implementazione, pur funzionante, sbagliata dal punto di vista della concorrenza.

3.2.3 Task lavoratori

La categoria dei task lavoratori comprende tutte quelle classi create per portare a termine un compito specifico, in maniera parallela rispetto ai task di servizio, ovvero, tramite thread separati. Questi sono gli attori principali sui quali si è costruito il sistema di sincronizzazione, il quale permette una loro semplice coordinazione.

`InRequest` Questo task gestisce i pacchetti che la rete P2P invia al nodo: soddisfa le richieste rispondendo con le informazioni di cui il nodo stesso è in possesso oppure smista le risposte ricevute alla richiesta pendente corretta. Quest'ultima operazione viene eseguita sfruttando la tabella delle richieste in sospeso, condivisa con `NetSender`, ovvero, `InRequest` controllerà che essa contenga effettivamente una richiesta relativa alla risposta ricevuta, inoltrando i risultati all'attendente. Si noti quindi come si configura nuovamente il modello produttore-consumatore tra `NetSender` e `InRequest`, il quale richiede che l'oggetto condiviso tra produttori e consumatori sia thread-safe, affinchè non ci siano problemi di concorrenza: tale condivisione è pertanto valida nella corrente implementazione in quanto in quanto la tabella hash condivisa è una `ConcurrentHashMap`, prelevata dal package `java.util.concurrent`, la quale garantisce la thread-safety[5].

`OutRequest` Effettua tutte le operazioni necessarie ad inviare un pacchetto sulla rete ed attenderne la risposta, gestendo ed effettuando gli eventuali re-invii. Questo task si trova a metà strada tra i veri utilizzatori della rete e `NetSender`, offrendo l'astrazione necessaria per permettere sia l'invio di un pacchetto "logico", sia la fruizione dei risultati: per questo motivo, tutti i campi necessari al "trasferimento" dei risultati devono essere protetti contro eventuali data-races, e ciò viene nuovamente fatto attraverso il costrutto del monitor, proteggendo i metodi di impostazione (*setter*) e lettura (*getter*) di queste variabili, ovvero tutti i loro possibili accessi.

`LookUp` Questa classe implementa l'algoritmo di ricerca del nodo più vicino ad un dato ID, all'interno della DHT. Possedendo dei valori di ritorno che possono essere fruiti da altri thread, quindi, anche questa classe deve gestire gli accessi ai suoi stati condivisi: nell'attuale implementazione ciò viene fatto in lettura, attraverso il costrutto monitor, ma non in scrittura, generando un errore.

`NSInsertionThread` L'inserimento di un nuovo nodo nel `NodeStorer` è un'operazione non banale in quanto richiede di controllare che il nodo da inserire sia attivo e, nel caso il k-bucket risulti pieno, ripetere ricorsivamente questa azione con il nodo ivi presente da più tempo. Questo algoritmo richiede del tempo ed è stato per questo implementato tramite un thread che possa venir eseguito parallelamente a quelli principali. Una particolarità di questo task è che non possiede stati condivisi, e per questo è automaticamente thread-safe. Viene citato in questa lista per completezza e perchè, facendo uso di altri thread, si proporrà un sistema per gestirli efficacemente.

[5] [6, `ConcurrentHashMap` API]

`NSFillThread`, `BootstrapThread` Questi due thread hanno la peculiarità di venir eseguiti una volta sola e più precisamente durante il caricamento del plug-in. Il primo cerca di riempire il `NodeStorer` con dei nodi predefiniti o provenienti dalla sessione precedente, il secondo invece si incarica di registrare tutti i task periodici su `PeriodicThread`.

3.2.4 Task che utilizzano altri task

Tutti i task qui descritti non sono indipendenti gli uni dagli altri in quanto molto spesso un thread ne genera altri: si pensi ad esempio a `LookUp` il quale, dovendo comunicare in rete, utilizzerà una serie di `Outrequest` per questo scopo. Nell'attuale implementazione la sincronia tra questi thread si avvale del semplice costrutto *busy-waiting* descritto nel codice sottostante:

Listing 3.2: Attesa *busy-waiting* per la fine di un task

```
// attendo che someTask finisca...
while(!someTask.ended()) {
  // attendo 1000ms
  wait(1000);
}

// ora someTask ha finito...
```

Questa forma di sincronia andrebbe sempre evitata in quanto è molto dispendiosa in termini di risorse di calcolo e tempo di elaborazione:

- se il task figlio è lungo, il processo padre sarà costretto a risvegliarsi più volte eseguendo molteplici controlli sullo stato del figlio: ogni risveglio, come è stato descritto nel capitolo sulla concorrenza, provoca un context-switch che priva della risorsa processore altri thread che la potrebbero usare in modo più profiquo;
- se il task figlio è breve, invece, il padre attenderà per un tempo maggiore del necessario dilatando il periodo totale di elaborazione.

Per mitigare parzialmente questi effetti, l'implementazione adotta un tempo di attesa che segue la formula del backoff esponenziale, fino ad un valore massimo: questo permette di non dilatare troppo l'attesa in presenza di task rapidi e risvegliare il processo padre un numero minore di volte qualora il figlio risultasse lento.

Listing 3.3: Attesa *busy-waiting* con backoff esponenziale

```
// imposto l'attesa iniziale
int time = START_TIME;

// attendo che someTask finisca...
while(!someTask.ended()) {
  wait(time);

  // raddoppio il tempo di attesa finche'
  // esso e' minore di MAX_TIME
  if (time < MAX_TIME) {
    time = time * 2;
  }
}

// ora someTask ha finito...
```

Questo modus operandi però non risolve adeguatamente il problema: il sistema di sincronia va modificato alla radice e il prossimo capitolo presenta un possibile approccio.

Capitolo 4

PariDHT: sincronizzazione

Il capitolo precedente ha descritto le problematiche di concorrenza presenti nel plug-in PariDHT, in questo vedremo delle metodologie per risolvere questi problemi ed altri simili. Per una più facile fruizione dei contenuti essi verranno suddivisi per tipologie di problema, partendo da quelli di concorrenza ed a seguire poi con quelli di sincronizzazione.

4.1 Concorrenza

Come descritto nel capitolo precedente, se tutte le classi di un programma multi-threaded non sono thread-safe, l'intero programma è sbagliato in quanto può originare errori imprevedibili. Si comincerà allora descrivendo alcuni pattern comuni di programmazione concorrente che sono tornati utili durante la correzione dell'implementazione di PariDHT.

4.1.1 Trasferimento asincrono di oggetti

All'interno di PariDHT vi è spesso un trasferimento di risorse tra due thread in modo asincrono, ovvero senza che entrambi i thread siano presenti al momento del passaggio; se questo concetto viene incarnato da una classe usata come tramite per il passaggio, ecco che ritroviamo `PendingPacket`, altrimenti è possibile pensare alla lettura dei risultati tra thread chiamanti e lavoratori durante le normali operazioni di PariDHT. Va specificato fin da subito che questo modo di operare deve essere attentamente valutato, altrimenti rischia di diventare un anti-pattern, in quanto riduce la sincronia tra i thread. Viene comunque di seguito presentato per la sua semplicità e perchè permette di introdurre la keyword `volatile` di Java™ .
Un trasferimento di oggetti asincrono è solitamente caratterizzato da quattro componenti:

- un metodo che permette di impostare gli oggetti da trasferire: `setData(Object data)`;
- un metodo che permette di leggere gli oggetti trasferiti: `Object getData()`;
- un metodo per sapere se il trasferimento è avvenuto e i dati sono disponibili: `boolean hasData()`;
- una variabile interna mutabile per mantenere l'oggetto da trasferire.

I metodi *get* e *set* non sono stati citati solo per buona norma di programmazione ad oggetti; l'incapsulamento infatti, insieme a immutabilità e specifiche precise sugli stati varianti, sono buone norme per la programmazione concorrente, dal momento che, una

variabile di stato mutabile e accessibile al di fuori di una classe, renderebbe quest'ultima thread-unsafe, in quanto potrebbe venir usata in modi inaspettati introducendo errori in esecuzione.

Fatta questa panoramica sull'oggetto, com'è possibile evitare possibili data race nell'accesso alla variabile interna? Una prima soluzione consiste nell'utilizzo del costrutto monitor: questo approccio, pur essendo corretto, in un caso così semplice può essere ottimizzato, evitando che, mentre un thread controlla se sono stati inseriti dati attraverso `hasData()`, un altro thread, che magari vorrebbe inserirli, sia temporaneamente bloccato. Per fare questo si può semplicemente marcare `volatile` la variabile condivisa. La parola chiave `volatile` istruisce il compilatore e l'ambiente di esecuzione che la variabile seguente è condivisa e quindi le operazioni fatte su di essa non devono subire riordinamenti. Le variabili volatili, inoltre, non sono mantenute in nessuna cache quindi una loro lettura restituirà sempre il valore più aggiornato. Queste garanzie permettono di affermare che, in un'implementazione come quella presentata nel listato 4.1, l'operazione presente nel metodo `setData()` *happens-before* l'operazione presente nel metodo `getData()`. La relazione happens-before, descritta nella Java™ Language Specification, accompagna tutto il package `java.util.concurrent` e sta a significare che se due azioni possono venir ordinate secondo questa logica, allora l'esito della prima azione è istantaneamente visibile alla seconda. Ciò implica che, se una scrittura viene eseguita temporalmente prima di una lettura, allora la lettura ritornerà sempre i dati appena impostati.

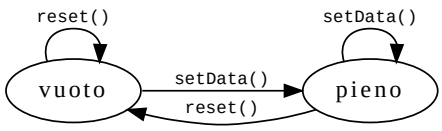
Listing 4.1: Classe che permette un trasferimento asincrono di oggetti

```
public class AsyncTransfer {
  private volatile Object holder;

  public setData(Object data) { holder = data; }
  public Object getData() { return holder; }
  public boolean hasData() { return holder == null; }
}
```

Un oggetto così strutturato garantisce la thread-safety per un numero qualsiasi di thread scriventi e lettori, nel senso che ogni thread lettore fruirà dei dati scritti dall'ultimo thread che ha eseguito `holder` ← `data`.

Problemi di concorrenza: check-then-act camuffati L'oggetto descritto finora presenta i due stati *vuoto* e *pieno* e un'unica transizione tra di essi, attivata dal metodo `setData()`. Si ipotizzi ora di aggiungere ai precedenti un ulteriore metodo `reset()` ,il quale non fa altro che aggiungere una nuova transizione dallo stato *pieno* allo stato *vuoto*.



Questo metodo, implementato attraverso un semplice `holder` ← `null`, non altera la thread-safety della classe, ma vediamo ora cosa potrebbe accadere se venisse utilizzata attraverso il codice 4.2.

Listing 4.2: Utilizzo improprio di una classe thread-safe

```
/* Simulazione di un thread lettore con a
 * disposzione un oggetto di tipo AsyncTransfer. */

// ...
```

```
if (asyncTransfer.hasData()) {
  useData(asyncTransfer.getData());
}

// ...
```

Un uso di questo tipo era corretto senza il metodo `reset()` ma ora potrebbe generare degli errori a runtime, ad esempio attraverso la serie di operazioni presentate nella tabella sottostante.

Thread 1 (scrittore)	**Thread 2 (lettore)**
• imposta i dati attraverso `setData()`	
	• `hasData()` ritorna `true` quindi il thread si accinge ad usare questi dati
• effettua un reset dei dati	
	• `useData()` utilizzerà dati non validi

Cosa non ha funzionato?
L'errore in questo caso sta nell'uso sbagliato che si fa dell'oggetto, in quanto si configura una race condition di tipo check-then-act; perchè un codice simile funzioni, il controllo sulla presenza dei dati e la loro restituzione dovrebbero essere un'operazione atomica, ma questo non è permesso dall'oggetto stesso. Un programmatore di sistemi concorrenti, nella progettazione di classi thread-safe, deve quindi tener conto anche di chi utilizzerà gli oggetti da lui sviluppati, introducendo metodi adeguati e documentando i possibili usi ed eventuali errori.

Problemi di concorrenza: read-modify-write di variabili volatile Le variabili di tipo `volatile` devono essere usate con cautela in quanto solamente le letture/scritture vengono fatte atomicamente e con garanzie di visibilità. Un'istruzione del tipo `volatileVariable++` è, come già visto, composta da tre operazioni, ovvero una lettura, una modifica e una scrittura: l'attributo `volatile` garantisce solo la visibilità ma non l'atomicità, quindi l'incremento non risulterebbe thread-safe. Per questo motivo un'attributo di tipo `volatile` è da preferire ad un'esplicita sincronizzazione solo se si verificano tutte e tre le seguenti condizioni[4]:

- le modifiche alla variabile non dipendono dal valore corrente, oppure si riesce ad assicurare che solo un singolo thread per volta modificherà tale valore;
- la variabile non partecipa nella definizione degli invarianti[1] con altre variabili d'istanza;
- non è necessario alcun lock quando un thread sta accedendo alla variabile.

4.1.2 Classi dotate di un singolo stato condiviso: l'esempio di ConfigFile

Nell'esempio precedente si è già parlato di una classe dotata di un unico stato condiviso, ma questo stato era una variabile che veniva semplicemente letta e scritta. Cosa fare se invece questo stato è un oggetto di cui vengono utilizzati i metodi? La risposta in questo caso è semplice: se lo stato condiviso è thread-safe e l'uso che ne viene fatto è legale, allora anche la classe utilizzatrice risulta thread-safe; se lo stato condiviso non garantisce

[1] Con *invarianti* si intendono i vincoli sullo stato di un oggetto

la correttezza in presenza di concorrenza, allora tale correttezza deve essere garantita dall'utilizzatore sincronizzando tutti i possibili accessi all'oggetto.

Tale thread safety all'interno di `ConfigFile` è stata ottenuta attraverso il primo metodo, ovvero adottando un'implementazione concorrente per l'hashtable contenente i parametri di configurazione. Le modifiche però non si sono limitate ad una semplice sostituzione in quanto tutto il codice è stato rivisto alla ricerca degli usi erronei della classe: questi erano principalmente check-then-act, simili a quelli visti nella sezione precedente, e che sono stati sostituiti con singoli metodi atomici forniti dall'oggetto concorrente usato.

A titolo di completezza, viene presentato anche il secondo metodo che implica la sincronizzazione di tutti gli accessi allo stato condiviso: agire in questo modo risulta comodo qualora non esistesse un'implementazione thread-safe dell'oggetto interno e il tempo necessario a scriverne una non garantirebbe guadagni sufficienti in termini di prestazioni. Il linguaggio Java™ , per questo scopo, mette a disposizione la parola chiave `synchronized` che permette di acquisire il *lock intrinseco*[2] di un oggetto definendo un blocco di codice, la cui entrata corrisponde ad effettuare una `lock()` e l'uscita una `unlock()`; ciò garantisce che l'esecuzione di uno di questi blocchi avvenga in mutua esclusione rispetto agli altri, fintanto che essi sono "protetti" dallo stesso oggetto[3] (e di conseguenza lo stesso lock intrinseco). Esistono tre modi diversi di usare la parola `synchronized` riassunti in tabella 4.1. Con questo strumento a disposizione, garantire la mutua esclusione sull'uso di un stato interno diventa un'operazione meccanica: *ogni* azione su questo oggetto deve avvenire all'interno di un blocco sincronizzato *protetto dallo stesso lock intrinseco*.

4.1.3 Classi con più stati condivisi

I problemi di concorrenza presenti in PariDHT sono tutti riassumibili nelle due sezioni precedentemente descritte. Per completezza però, viene anche trattato il caso in cui, all'interno di una classe, siano presenti più variabili di stato mutabili e condivise. In questo caso, per garantire che la classe risulti thread-safe, non è sufficiente che tutti gli stati siano thread-safe in quanto vi possono essere delle transizioni atomiche da garantire come tali, ma riguardanti più di una variabile. Il procedimento di risoluzione della concorrenza in questi casi segue due possibili strade: utilizzare il concetto del *monitor* oppure una *gestione analitica*.

Il monitor ha dalla sua parte la facilità implementativa, garantita dalla keyword `synchronized` descritta precedentemente. Il programmatore deve prestare attenzione solo a che tutti gli accessi agli stati condivisi siano protetti dal medesimo oggetto, perchè, così facendo, si ottiene una mutua esclusione totale che permette di scrivere codice senza tener conto delle problematiche dovute alla concorrenza. Questo approccio però ha un grande svantaggio, ovvero non permette di raggiungere performance elevate. Si pensi ad esempio ad una tabella hash in cui vi sono pochi thread scrittori e moltissimi thread lettori: attraverso il monitor i thread lettori sarebbero costretti ad operare uno alla volta, nonostante non facciano modifiche sui dati e potrebbero quindi agire in parallelo. Attraverso questo sistema è stata implementata la classe `NodeStorer`: essa deve coordinare principalmente l'accesso a tutti i k-buckets, e, siccome il numero di operazioni in lettura è confrontabile con quelle in scrittura, renderebbe costosa un'analisi della concorrenza più accurata.

[2]Ogni oggetto caricato dalla JVM possiede un proprio lock rientrante a cui è associata una coda di thread in attesa

[3]É utile ricordare che un oggetto è l'istanza di una classe in memoria: una classe può essere istanziata più volte dando luogo ad oggetti diversi e indipendenti.

Tabella 4.1: Possibili usi della keyword *synchronized*

nella firma di un metodo: in questo modo viene acquisito il lock dell'oggetto `this` (una parola chiave che ritorna l'oggetto corrente); il codice protetto risulta essere l'intero metodo. Questo uso è sconsigliato in quanto qualunque thread abbia un riferimento all'oggetto corrente ha la possibilità di acquisirne il lock intrinseco per un tempo indefinito, impedendo che altri thread possano adoperarne i metodi sincronizzati ivi contenuti.

```
synchronized void foo() {
  // ...
}
```

nella firma di un metodo statico: in questo caso, non essendo disponibile la keyword `this`, viene acquisito il lock dell'oggetto *class*; il codice protetto risulta essere nuovamente l'intero metodo. Questo uso è sconsigliato alla stregua del precedente in quanto qualunque thread può ricavare l'oggetto *class* di una qualsiasi classe, acquisirne il lock intrinseco, ed impedire ad altri thread di adoperare i metodi sincronizzati protetti da questo oggetto.

```
static synchronized void foo() {
  // ...
}
```

definendo un blocco sincronizzato: questa modalità permette di esplicitare l'oggetto sul quale effettuare il lock, ed il codice protetto è quello interno al blocco. Questo è l'uso che solitamente si fa della parola chiave `synchronized` in quanto permette la sincronizzazione su un oggetto privato con cui il codice esterno alla classe non può interferire e offre la possibilità di scegliere con maggior granularità le porzioni di codice da eseguire in mutua esclusione.

```
private Object mutex = new Object();

void foo() {
  // ...

  synchronized(mutex) {
    /* codice protetto
     * dall'oggetto mutex */
  }

  // ...
}
```

D'altra parte, con la gestione analitica della concorrenza si studia il tipo di carico a cui è sottoposta una classe al fine di ridurre allo stretto indispensabile, o eliminare totalmente, i blocchi di codice da eseguire in mutua esclusione. Ciò garantisce un incremento delle performance, in quanto, minore è il tempo in cui un thread detiene un lock, maggiore è la probabilità che altri task non debbano arrestarsi in attesa che quel lock sia libero (diminuendo i context-switch) e, inoltre, anche se il lock risultasse occupato, il tempo di attesa necessario per la sua acquisizione diminuirebbe; non sempre, però, lo sforzo necessario per l'analisi e l'implementazione giustifica i benefici che si possono ottenere e va quindi preventivamente valutato. Un esempio di questo tipo di implementazione viene fornito dalle classi `WorkerManager` e `SynchronizableManager` trattate nella sezione successiva.

4.2 Sincronizzazione

L'altro punto critico che deve essere gestito all'interno di un sistema concorrente in cui più task devono comunicare tra loro è la sincronizzazione di questi stessi task, in modo da massimizzare le performance dell'applicazione. In questa sezione verrà descritta l'infrastruttura atta a questo scopo e che è stata implementata nel plug-in PariDHT.

4.2.1 Definizione degli obiettivi

Nel capitolo precedente si è spiegato come l'unico costrutto di sincronia presente all'interno di PariDHT sia il busy waiting; da una parte ciò permette completa libertà nella definizione della nuova struttura da adottare, dall'altra si tratta di apportare modifiche all'intero plug-in e questo può essere fatto solo a patto che non venga snaturato o stravolto tutto ciò che già è stato scritto. Gli obiettivi tenuti a mente sono quindi stati:

mantenimento del codice corrente per quanto possibile si è cercato di non alterare le classi già scritte, ciò significa sia non alterare la struttura del codice già scritto, sia non doverne aggiungere molto per sfruttare il meccanismo di sincronizzazione.

riusabilità del codice la riusabilità è uno dei punti di forza dei linguaggio di programmazione ad oggetti, tuttavia, quando si fanno delle aggiunte in corso d'opera, non sempre si riesce a "piegare" il codice in modo da poterlo usare tale e quale, senza ripetizioni.

facilità d'uso questo è stato un fattore cruciale nella progettazione. Quando si parla di concorrenza, gli algoritmi di risoluzione, anche di sistemi banali, diventano più complessi e in un progetto quale *PariPari*, in cui vi è un alto ricambio di sviluppatori, diventa difficile spiegare come sono stati scritti e come si utilizzano gli strumenti messi a disposizione. Per questo motivo ogni classe deve essere adeguatamente commentata in ogni suo punto e deve fornire, attraverso i suoi metodi, un'efficace astrazione, tale da permetterne un utilizzo immediato e una facile e sicura estensione.

correttezza, robustezza e performance questi tre obiettivi sono propri della programmazione concorrente e sono stati già descritti; una menzione particolare, però, va data alla robustezza in quanto, per il veloce ricambio generazionale proprio di un progetto come *PariPari*, è necessario che il codice scritto sia il più possibile "stupid proof", ovvero, qualora l'uso degli strumenti messi a disposizione non risultasse corretto, esso deve venir segnalato tramite eccezioni opportune e descrittive.

4.2.2 Attesa di un singolo task

Il primo passo fatto nella progettazione è stato in realtà già descritto nei capitoli precedenti; si è trattato di individuare un denominatore comune ai vari task di PariDHT e questo ha portato all'individuazione delle due categorie di thread: *di servizio* e *lavoratori* (o worker).
I thread di servizio non avevano particolari caratteristiche se non quella di avviarsi al caricamento del plug-in e terminare alla sua uscita; i thread lavoratori, invece, possedevano tutti una variabile che descriveva il loro *stato corrente*. Questa variabile, aggiornata dal task stesso durante la sua esecuzione, era ciò che i thread chiamanti usavano per capire se il worker avviato aveva terminato i suoi compiti ed era quindi possibile reperire i risultati, oppure, in caso contrario, era necessario attendere ulteriormente, attraverso il busy

waiting. Il comportamento appena descritto si prestava in maniera efficace a definire una condizione di attesa e una di risveglio per i chiamanti:

attesa Un task chiamante dovrà attendere la fine di un worker *finchè* quest'ultimo non raggiunge uno stato terminale;

risveglio Un task chiamante, in attesa della fine di un worker, dovrà essere risvegliato *quando* questo worker raggiunge uno stato terminale.

Nonostante sembrino delle tautologie, queste due condizioni sono la base del sistema di sincronizzazione in quanto, se una delle due mancasse, l'unico modo per attendere la fine di un task resterebbe davvero il busy waiting. Nella pratica esse implicano la necessità di indagare una variabile di stato interna per entrare in condizione di attesa, e "capire" quando essa viene impostata a un particolare valore al fine di riprendere l'esecuzione. Un simile comportamento può non essere intuitivo in quanto resta da definire come un task in attesa, ovvero privo della risorsa processore, possa capire quando si verifica la condizione di risveglio: il trucco sta nel fare in modo che sia un altro attore, magari proprio chi effettua il cambiamento di stato, ad informarlo di ciò, passivamente, permettendogli di proseguire nuovamente con i suoi compiti. É chiaro che un programmatore non può gestire autonomamente tutta la serie di operazioni che stanno dietro alla sospensione e riattivazione di un thread, in quanto esse dovrebbero passare attraverso il sistema operativo e le sue chiamate; questo comunque non presenta un limite in quanto tutti i linguaggi di programmazione attuali offrono queste capacità. Java™ ad esempio fornisce, per ogni oggetto, una terna di metodi che sono `wait()`, `notify()` e `notifyAll()`, atti a questo scopo: un thread che esegue una `wait()` su un oggetto verrà messo in uno stato di attesa finchè un'altro thread non esegue una `notify()` o `notifyAll()` sullo stesso oggetto. Questi metodi necessitano che il thread invocante detenga il lock intrinseco dell'oggetto sul quale vengono invocati, perciò devono essere sempre posizionati all'interno di un blocco `synchronized`. La differenza tra `notify()` e `notifyAll()` è visibile quando più threads sono in attesa sullo stesso oggetto: come è possibile dedurre dal nome, con la prima chiamata solo uno di questi thread viene risvegliato, mentre con la seconda tutti riprendono la loro esecuzione; l'utilizzo di questi due metodi non è intercambiabile in quanto in alcuni casi svegliare un singolo thread potrebbe condurre ad un deadlock perchè, ad esempio, solo uno tra tutti i task in attesa ha i requisiti necessari per proseguire, e non è detto che `notify()` svegli proprio quello.

In tabella 4.2 è presentata una possibile implementazione della logica appena descritta che permette di specificare alcune cose. In primo luogo l'oggetto `condition`, del quale vengono chiamati i metodi `wait()` e `notifyAll()`, deve essere lo stesso identico oggetto in memoria, altrimenti la notifica di risveglio non va a buon fine. In seconda battuta, si può notare che il codice parla di condizione di attesa, ma non è esplicitata la condizione di risveglio in quanto è subordinata al non verificarsi della condizione di attesa stessa: se risultasse necessario permettere il risveglio quando solo parte di una condizione di attesa non è più valida, potrebbe essere utile sfruttare la classe `Lock` e il suo metodo `newCondition()` del package `java.util.concurrent.locks`, che permette di definire più code di attesa, ognuna con una propria politica indipendente. In terzo luogo, il blocco `synchronized` comprende sia il test delle condizioni di attesa che la loro modifica, dal momento che è necessario garantire la visibilità di queste variazioni a tutti i thread: resta comunque valido che, se la visibilità è già garantita in altro modo, le uniche istruzioni che devono venire obbligatoriamente inserite all'interno del blocco di mutua esclusione sono `wait()` e `notifyAll()`.

Tabella 4.2: Realizzazione di attesa e risveglio con i metodi `wait()` e `notifyAll()`

```
void await() {
  synchronized(condition) {
    while(<condizioni_di_attesa>) {
      condition.wait();
    }
  }
}
```

```
void foo() {

  synchronized(condition) {
    /* .....
    * codice che modifica
    * i valori delle variabili di
    * condizione */

    condition.notifyAll();
  }
}
```

Attraverso questo sistema è dunque possibile un'attesa senza sprechi di risorse; tuttavia, la sua implementazione avrebbe comportato la scrittura di codice simile in molte classi distinte. Per evitare queste replicazioni di codice si è deciso di standardizzare gli stati che un worker poteva attraversare durante lo svolgimento dei suoi compiti. Così facendo, si è potuta introdurre una sola classe astratta, con la definizione dei metodi di attesa e risveglio, che tutti i worker avrebbero dovuto semplicemente estendere. In merito a questo approccio va fatta una precisazione in quanto, in Java™ , obbligare una classe ad estenderne un'altra è un'operazione delicata perchè, non essendoci ereditarietà multipla, impedisce alla classe di poterne estendere altre, creando una rigida struttura verticale. La scelta verso l'ereditarietà va quindi motivata e le ragioni, in questo caso, stanno nel fatto che, essendo PariDHT un plug-in di *PariPari*, ogni suo task è già obbligato ad estendere la classe `PariPariRunnable`, perciò è bastato introdurre una nuova classe astratta `AbstractWorker`, che estenda `PariPariRunnable` ed implementi il meccanismo di sincronia, per aver ottenuto lo scopo senza limitare lo sviluppo del codice più di quanto non fosse già stato fatto.

Gli stati definiti sono riportati in figura 4.1 assieme alle loro transizioni: queste ultime sono semplicemente ipotizzate ma niente vieta a un task di aggiungere le proprie o modificare quelle esistenti.

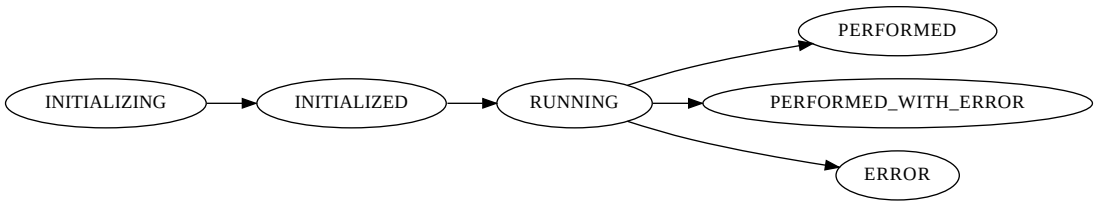


Figura 4.1: Possibili stati di un worker

Sebbene i nomi siano esplicativi, è opportuno fare un breve riassunto del significato di ognuno di questi stati:

initializing Un worker si trova in questo stato appena la sua classe viene istanziata, ovvero mentre sta eseguendo il costruttore;

initialized Questo stato indica che il task ha terminato la sua inizializzazione e sta attendendo di poter essere eseguito;

running Il lavoratore è stato avviato e sta svolgendo i compiti assegnati;

performed Rappresenta un corretto svolgimento dei compiti assegnati;

error Indica una terminazione anormale del thread lavoratore dovuta ad eccezioni impossibili da gestire;

performed with error Indica una terminazione prematura dei compiti del worker, in seguito a condizioni note ma che non permettono un ulteriore avanzamento.

La classe `AbstractWorker` definisce gli ultimi tre stati come stati terminali (in cui risvegliare eventuali attendenti) e forza il passaggio attraverso INITIALIZING, RUNNNING e PERFORMED, anche se quest'ultimo solo nel caso il task si trovi ancora in uno stadio intermedio quando l'elaborazione termina. Le altre transizioni sono lasciate alla libertà del programmatore.

4.2.3 Attesa di almeno un task

Un'analisi ulteriore del comportamento di PariDHT ha evidenziato in molti punti del plug-in la necessità di avviare una serie di task lavoratori, attendere che almeno uno di essi finisca, sfruttare i risultati del worker terminato per poi rimettersi in attesa dei restanti, continuando questa serie di operazioni finchè tutti i lavoratori avviati non sono terminati. Nell'implementazione in esame, l'attesa consisteva ancora una volta in un busy waiting, seguito da un test di tutti i worker avviati, per capire se almeno uno di essi avesse finito. Questo modo di operare è dispendioso per tutti i motivi già visti per il busy waiting ed in più costringe il thread chiamante ad un lavoro ulteriore per capire quale sia il worker terminato, anche qualora non ce ne fosse nessuno. Adottando uno stratagemma simile a quello visto nella sezione precedente, però, si può ipotizzare di mettere in attesa il task chiamante e fare in modo che siano i lavoratori, quando terminano, a segnalare l'evento risvegliando il chiamante e, possibilmente, alzando una bandierina che ne permetta un riconoscimento immediato: questo è l'obiettivo da cui è partita la definizione della classe `WorkerManager`.

Avendo basato la progettazione della classe sulle esigenze del codice, il primo modello prodotto è stato di tipo black-box: il manager avrebbe dovuto accettare l'inserimento di una serie di worker e, finchè tutti non fossero stati ritornati, permettere l'attesa per la fine di uno di essi; inoltre, avrebbe dovuto mettere a disposizione un metodo per dare la possibilità ai lavoratori terminati di avvertirlo della loro fine. Quest'ultima è una condizione chiave, che permette al manager sia di evitare il busy waiting, sia, soprattutto, di essere una classe passiva, ovvero senza la necessità di un thread interno. In figura 4.2 è esplicitato il modello appena abbozzato con i domini di concorrenza a cui il manager sarà sottoposto: agli ingressi troviamo i *produttori*, ovvero tutti quei thread che inseriranno dei workers nel manager, ed i *workers* stessi, che segnaleranno la loro fine; all'uscita invece vi sono i task *consumatori*, che preleveranno i lavoratori terminati o, se non ce ne sono, attenderanno la loro fine.

Un ulteriore requisito imposto al manager è l'*asincronia*. La fine di un worker, come pure un consumatore che si mette in attesa, sono eventi asincroni, che accadono cioè in momenti del tempo arbitrari: per funzionare correttamente, il manager deve *ricordarsi* quali sono i worker terminati in modo che, all'atto della segnalazione, non sia necessaria la presenza contemporanea di un consumatore; perciò, quando uno di questi consumatori

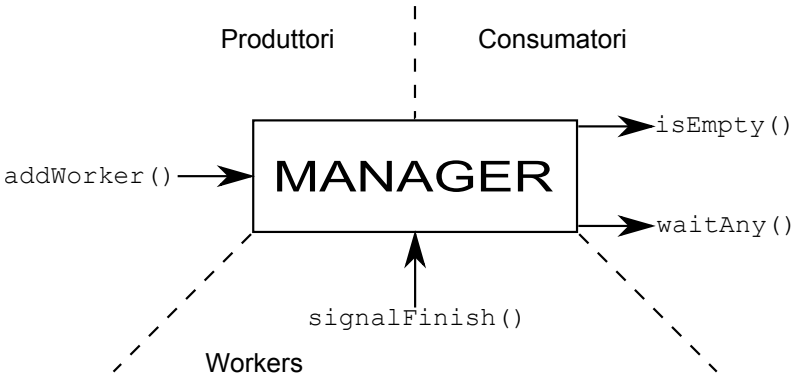


Figura 4.2: Prima definizione black-box del manager

richiederà un worker, se ve ne sono già di terminati, verranno semplicemente restituiti senza attesa.

L'analisi fatta finora risulta già più elaborata di quelle che erano le necessità del plug-in: finora infatti si è parlato di più produttori e consumatori, mentre nel codice essi si riducono ad un unico thread che a volte funge da produttore e a volte da consumatore. Per sviluppare uno strumento più versatile, tuttavia, si è deciso di non adottare questa ipotesi semplificativa, avendo però un occhio di riguardo a non renderne complicato l'utilizzo nel caso in cui ci si trovasse effettivamente in questa condizione. La scelta dei termini usati, comunque, non è casuale, poichè è possibile vedere il manager come il contenitore delle risorse nel modello concorrente produttore-consumatore, nel quale, come si è detto, vi siano sia più produttori che più consumatori. Questo modello è sufficientemente descrittivo e pone subito un problema: qualora i thread che inseriscono i worker terminassero questo loro lavoro, cosa succederebbe agli eventuali consumatori in attesa? Senza alcun tipo di segnalazione, essi continuerebbero ad attendere pazientemente la fine di un task lavoratore inesistente, rimanendo bloccati fino a quando o l'applicazione viene chiusa oppure qualche altro thread li interrompe. Per evitare questo comportamento i thread produttori, una volta ultimato il loro compito, dovrebbero segnalare al manager l'evento, in modo che esso possa notificare opportunamente gli eventuali consumatori in attesa, permettendo una loro corretta terminazione. Si è deciso di incaricare il manager di questo compito in quanto ciò permette di rendere indipendenti produttori e consumatori, ovvero non è necessario introdurre alcuna forma di sincronia esterna per evitare che si verifichi lo stallo precedentemente descritto, il che facilita sempre di gran lunga lo sviluppo. La rete di Petri[4], in figura 4.3, descrive come sia possibile ottenere questa funzionalità: qualora i produttori abbiano raggiunto il loro stadio terminale, ciò viene segnalato marcando il posto FINE. Così facendo, qualora non ci fossero più marche nel posto DEPOSITO che rappresenta i worker inseriti, viene abilitata la transizione evidenziata, che permette anche al consumatore di terminare correttamente in C_END.

4.2.4 Definizione di `WorkerManager`

Nella definizione dei metodi del `WorkerManager`, fruibili dai task consumatori, sono stati adottati due prefissi distinti a seconda dei metodi da usare: quando solo un thread

[4]Le reti di Petri sono un modello matematico per la descrizione di sistemi distribuiti o concorrenti: si basano su un grafo bipartito composto da *posti* (i cerchi, significanti delle condizioni), da *transizioni* (le barrette, ovvero degli eventi atomici che possono accadere) e da *archi* che legano posti e transizioni definendo pre e post-condizioni.

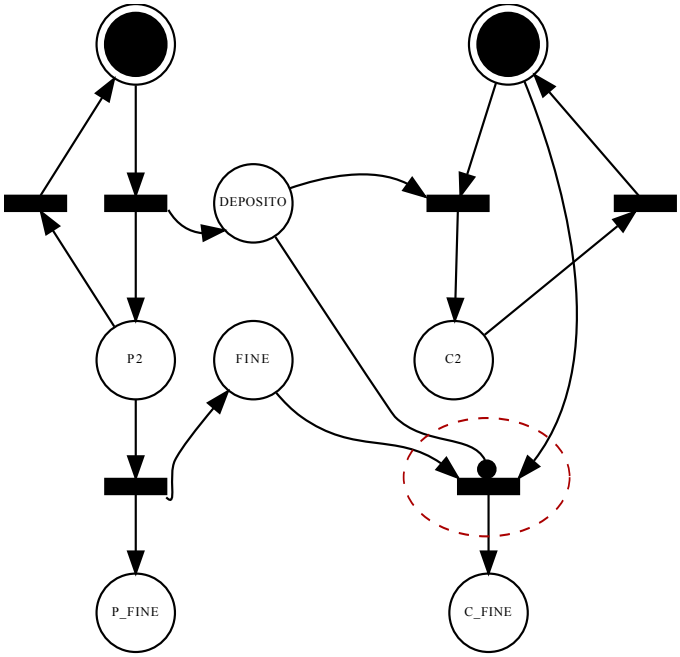


Figura 4.3: Rete di petri che simula il modello produttore/consumatore dotato di stadio finale

produttore/consumatore accede al manager, si è nel caso d'uso dei metodi `wait`, mentre quelli progettati per più consumatori, iniziano con `book`. La differenza tra i due è subordinata al concetto di *safe-booking*. Un *safe-book* (prenotazione sicura) viene fatto da un thread quando, nell'istante in cui si pone in attesa della terminazione di un worker, determina che sicuramente ne otterrà uno. Questa condizione tuttavia non è sempre vera e, nel caso non lo fosse, il safe-book segnala immediatamente l'errore, senza alcuna attesa. Sfruttando questa definizione, possiamo dire che i metodi `wait*` sono basati direttamente su safe-booking e lanciano un'eccezione ogni qualvolta questa operazione non va a buon fine; i metodi `book*`, invece, sono a conoscenza dello stato di terminazione dei thread produttori quindi aspettano l'opportunità di un safe-book fino a quando il manager non raggiunge il suo stadio terminale, rilasciando in questo caso eventuali thread in attesa. Utilizzando un metodo `book*`, quando un singolo thread fa da produttore e consumatore, si può incorrere in un'attesa infinita se questo thread non ha preventivamente inviato il segnale di terminazione al manager; usare i metodi `wait*` quando si ha a che fare con più consumatori, invece, non porta a grossi rischi, salvo essere consapevoli del fatto che essi tentano di effettuare un semplice safe-book e quindi, nel caso non ci fossero worker disponibili, ritornano immediatamente. Il manager in ogni caso garantisce, anche in presenza di "errori" di utilizzo, che i task ritornati attraverso i suoi metodi di uscita siano, soli e tutti, quelli registrati[5], ed ognuno una volta sola. Un'ultima precisazione va fatta sull'equità (fairness): il manager è *non equo*, ciò significa che non è garantito l'ordine con il quale saranno soddisfatti i thread in attesa della terminazione di un worker. Se due task A e B sono in attesa sullo stesso metodo (ad esempio `waitAny()`) ma A lo è da più tempo, quando un worker termina, non ci sono garanzie che esso venga assegnato ad A piuttosto che a B. Ciò che viene garantito, invece, è l'ordine di restituzione dei task lavoratori terminati, il quale segue una politica FIFO (Firs-In First-Out), basata sugli

[5]Questo a patto che tutti i task lavoratori siano terminati e abbiano correttamente segnalato questo fatto al manager.

arrivi delle loro segnalazioni di terminazione.

Per facilitare lo sviluppo e la comprensione di `WorkerManager` è indispensabile definire gli stati in cui esso può venirsi a trovare e le possibili transizioni tra essi: la figura 4.4 è un loro utile riassunto a cui segue un'analisi più approfondita di ogni stato. Quando il

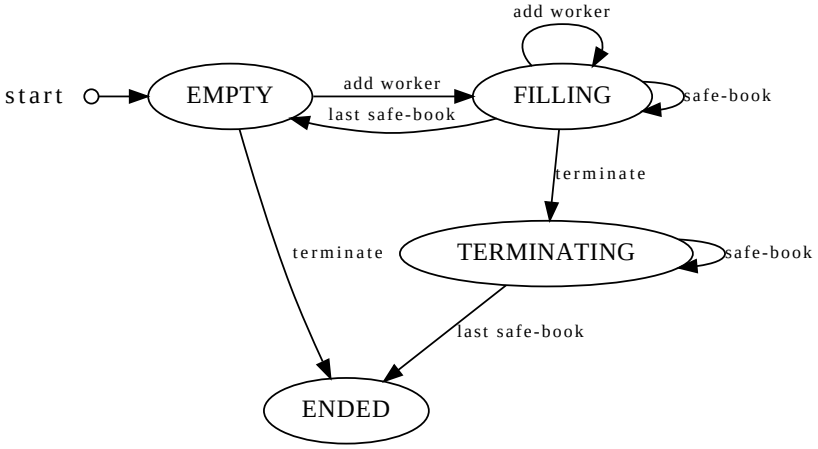


Figura 4.4: Definizione degli stati del manager

manager viene istanziato si troverà inizialmente nello stato EMPTY, in quanto esso significa *possibilità di inserire nuovi worker ma impossibilità di eseguire un safe-book*. É utile notare che questa definizione permette la presenza di worker all'interno del manager ma, se sono stati tutti prenotati in modo sicuro, esso è comunque considerato vuoto. Durante l'inizializzazione, in ogni caso, non è ancora stato aggiunto alcun task lavoratore, quindi il manager è vuoto a tutti gli effetti; tuttavia, appena un produttore invoca il metodo di inserimento, la transizione è verso lo stato FILLING. Quando il manager si trova nello stato FILLING significa che è *disponibile ad accettare nuovi inserimenti e safe-books*: esso, assieme ad EMPTY, sono i normali stati operativi, discriminati essenzialmente dalla disponibilità o meno di un safe-book. Nel momento in cui i produttori informano il manager che non vi sarà più alcun inserimento di task lavoratori si entra nella fase terminale del manager, identificata dallo stato TERMINATING. Un manager che si trova in questo stato *impedisce qualsiasi aggiunta di nuovi worker ma accetta safe-books* finchè ne ha la disponibilità: è una condizione, questa, di svuotamento, in cui si attende semplicemente la terminazione di tutti i task lavoratori ancora pendenti. Quando anche nello stato TERMINATING non vi è più la possibilità di un safe-book, allora il manager raggiunge la sua conclusione con lo stato ENDED nel quale *non sono permessi nè inserimenti nè safe-books*: ancora una volta si vuole sottolineare che, raggiunto questo stato, il manager può ancora contenere alcuni worker pendenti, ma questi sono già stati tutti prenotati in modo sicuro.

Questi sono gli stati in cui può venirsi a trovare il manager ma, nel caso in cui un singolo thread svolga la funzione sia di produttore che consumatore, non è necessario che si passi attraverso gli stadi terminali. Sotto questa ipotesi semplificativa, l'utilizzo del manager è regolato dalla logica stessa del thread produttore/consumatore, quindi il raggiungimento dello stato EMPTY può stare a significare "tutti i worker inseriti sono stati ritornati", terminando senza problemi in modo prematuro il ciclo di vita del manager stesso.

In seguito alla definizione di questi stati sono stati introdotti altri due metodi per il manager: `waitUntilEmpty()`, che permette all'unico thread produttore/consumato-

re di attendere la terminazione di tutti i worker inseriti o, più precisamente, continua ad eseguire dei safe-books finchè almeno uno di essi non va a buon fine; il metodo `bookUntilEnded()`, invece, che continua ad eseguire questi safe-books fino a quando il manager non raggiunge il suo stato ENDED. Quest'ultimo metodo calza bene nel caso in cui vi siano più produttori ed un unico consumatore che non debba eseguire una computazione eccessiva con ogni worker terminato; nel caso più thread vengano posti in attesa su questo metodo, però, si deve prestare attenzione al fatto che, essendo il manager non equo, non vi sono garanzie sull'eguale ripartizione dei task lavoratori terminati tra i vari attendenti.

In merito alla robustezza del codice, il manager prende due precauzioni che possono aiutare ad individuare eventuali errori di utilizzo: da una parte, viene impedito l'inserimento di uno stesso worker più di una volta, dall'altra, è permessa la segnalazione della loro terminazione solo da parte dei worker già registrati ed in ogni caso ciò non può avvenire più di una volta. La prima accortezza permette di individuare eventuali race condition sull'uso del manager, dovute ad una scarsa sincronizzazione tra i thread produttori; la seconda, invece, impedisce di snaturare il concetto di thread lavoratore ideato come un oggetto dotato di una sua evoluzione e terminazione irreversibili.

4.2.5 Astrazione dai thread

L'ultimo passaggio nell'introduzione dell'infrastruttura di sincronia ha cercato un'astrazione dal concetto di thread. Questa necessità è sorta perchè in DHT esistono delle classi, quali `PendingPacket` e `DHT`, che non sono dei task, ma sulle quali altri thread stanno in attesa di un evento, in modo simile a quanto accade nell'attesa tra due thread. In futuro, inoltre, si potrebbe voler attendere che almeno uno, di un insieme di questi oggetti, venga segnalato, riproponendo le stesse problematiche risolte con l'introduzione di `WorkerManager`. Per questi motivi il concetto di worker utilizzato finora è stato generalizzato in quello di *oggetto sincronizzabile* (interfaccia `ISynchronizable`), che permette ad altri thread di attendere il raggiungimento di un suo qualche stadio conclusivo e, anche in questo caso, irreversibile.

Per cercare di rendere standard questa infrastruttura e soprattutto il sistema di segnalazione al manager della "terminazione" degli oggetti sincronizzabili, quest'ultimi sono stati estesi attraverso gli oggetti sincronizzabili *attivi* (interfaccia `IActiveSynchronizable`): un oggetto di questo tipo si incarica di segnalare (da qui l'aggettivo "attivo") il raggiungimento del suo stadio terminale presso dei *semafori per synchronizable* (interfaccia `ISynchronizableSemaphore`), opportunamente registrati presso l'oggetto stesso.

Grazie a questi accorgimenti, è stato possibile definire due manager: uno per semplici oggetti sincronizzabili, incarnato dalla classe `SynchronizableManager`, e l'altro per task worker, che ha continuato ad esser chiamato `WorkerManager`. `SynchronizableManager` ha mantenuto tutti i metodi e le funzionalità viste durante l'analisi del manager; `WorkerManager`, invece, utilizza un `SynchronizableManager`, ma è dotato di una coda di task lavoratori non ancora avviati: il manager è incaricato di lanciarli un po' alla volta facendo in modo che, in ogni istante, non ne risultino pendenti più di un certo numero, chiamato *concorrenza massima*. Quest'ultima affermazione non deve far pensare che il manager sia diventato un'entità attiva, poichè esso è rimasto privo di thread interni, ma, per poter lanciare i task in attesa, è costretto a "rubare" cicli macchina, a volte, ai thread produttori che inseriscono nuovi worker, altre, ai worker stessi, quando segnalano la loro terminazione. In seguito ad esigenze del plug-in, inoltre, è stato necessario introdurre la possibilità di eliminare tutti gli eventuali task in attesa attraverso il metodo `dropWaiting()`. Questa nuova dinamica costringe a rivedere profondamente il grafo degli stati del manager

precedentemente descritto, dal momento che non risulta più vera l'affermazione che "un worker inserito nel manager verrà sicuramente restituito attraverso i metodi di uscita"; in questo particolare manager, quindi, un safe-book può essere effettuato solo nei confronti dei task avviati, ma non di quelli ancora in attesa. Gli stati di `WorkerManager` hanno così acquisito la forma presentata in figura 4.5.

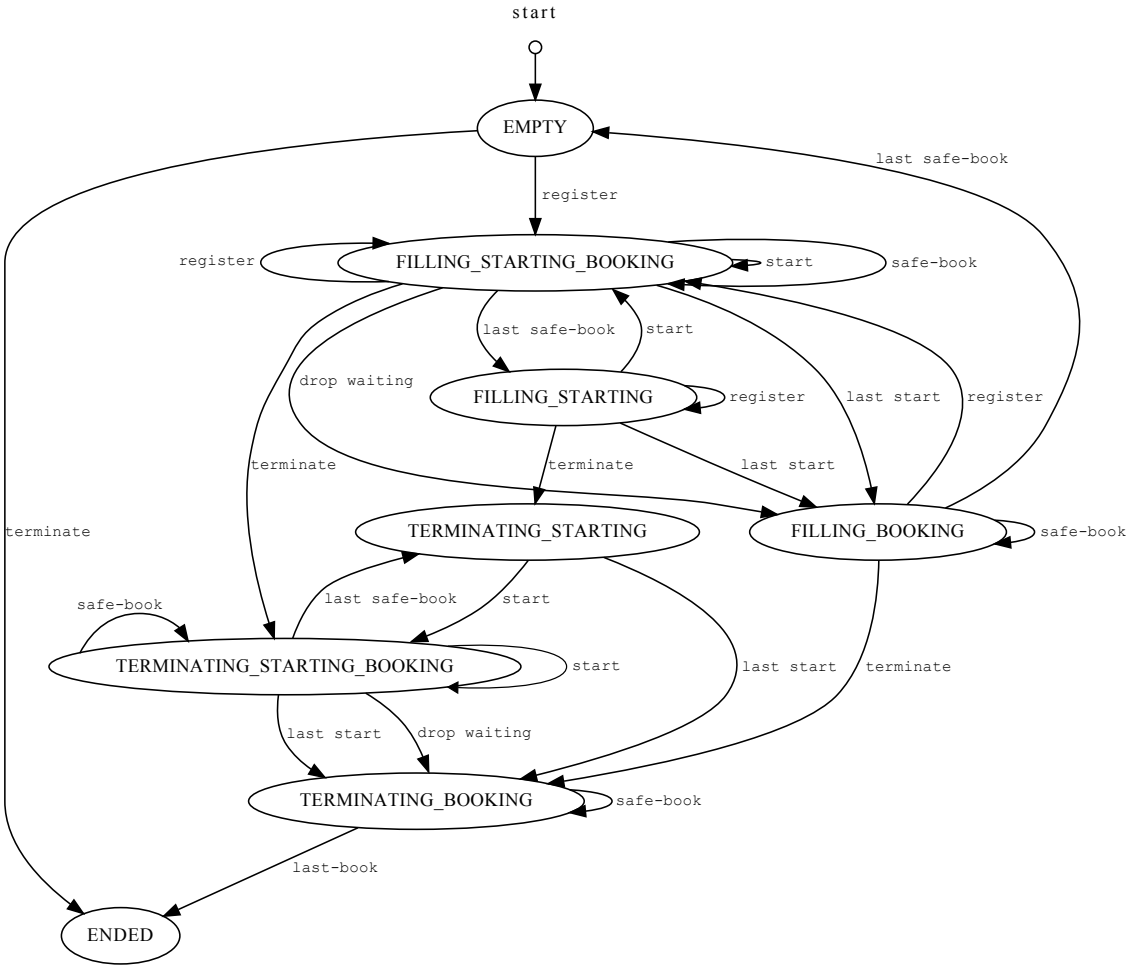


Figura 4.5: Definizione degli stati di `WorkerManager`

I nomi degli stati rappresentano le operazioni ammesse mentre sugli archi sono presenti gli eventi che permettono le transizioni. FILLING_STARTING_BOOKING, ad esempio, significa che in quello stato è possibile registrare nuovi task lavoratori, avviare quelli in attesa ed effettuare dei safe-book, mentre TERMINATING_BOOKING significa che le nuove registrazioni sono interdette come pure non vi sono worker in coda, ma rimane la possibilità di effettuare almeno un safe-book.

In figura 4.6, infine, è presentato un grafico riassuntivo di quanto è stato introdotto: rispetto a quanto detto finora, l'unica novità riguarda l'introduzione dell'interfaccia `ISynchronizableManager` in cui sono definiti i metodi base che un manager deve garantire; si vuole notare che questa interfaccia volutamente non estende `ISynchronizableSemaphore` in modo da mantenere separati i concetti di "semaforo" e manager.

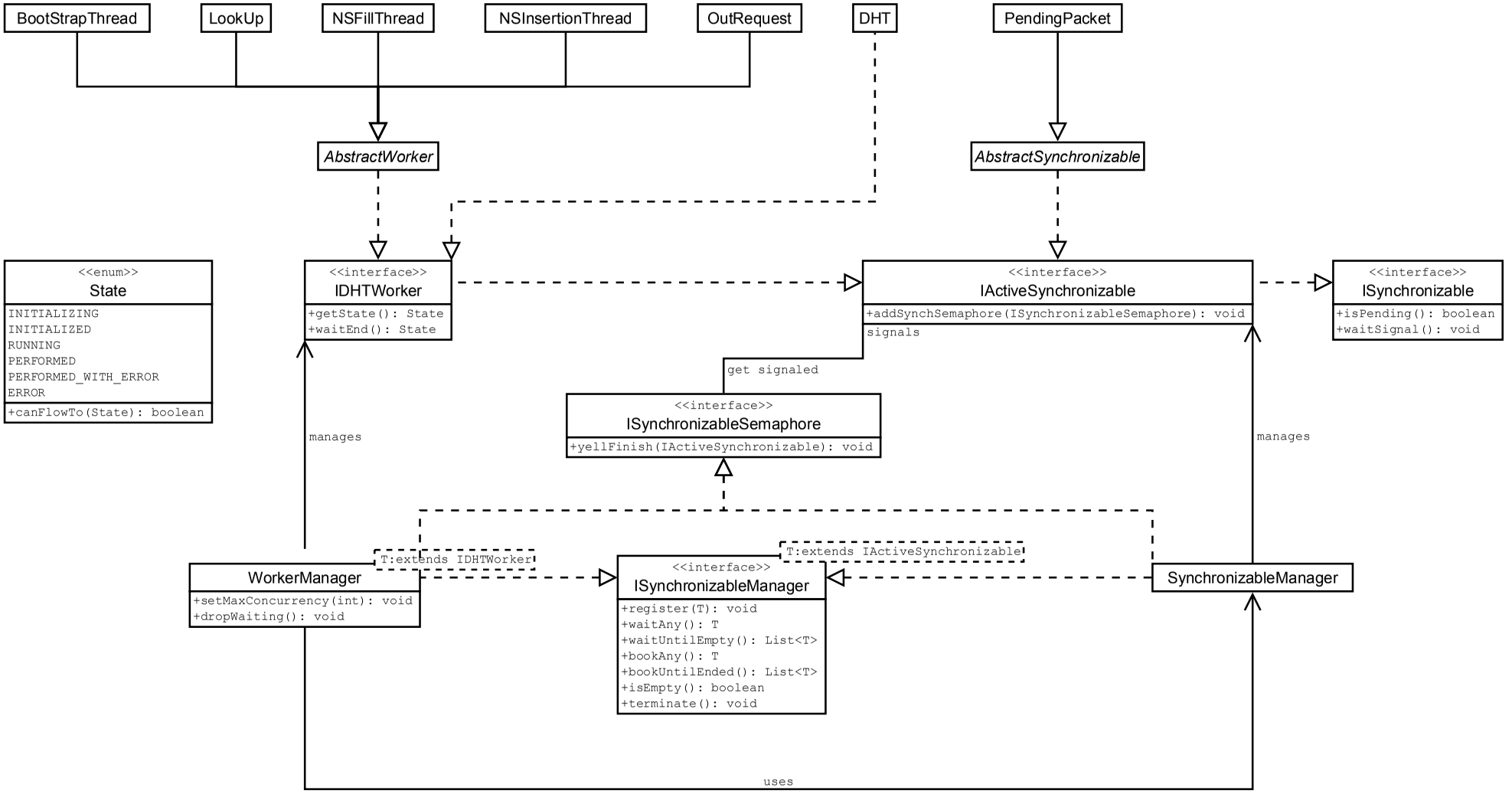


Figura 4.6: Grafico riassuntivo dell'infrastruttura di sincronizzazione

Capitolo 5

Analisi delle prestazioni

Quando si introducono nuovi componeni in un sistema è necessario valutare, sia prima che dopo, che questi oggetti offrano davvero le caratteristiche per i quali si è deciso di adottarli. L'analisi preventiva è già stata fatta nei capitoli precedenti; in questo verranno presentati alcuni test che mirano a verificare se gli obiettivi proposti sono stati raggiunti, e con quali margini di guadagno.

5.1 Ambiente di test

Parlare di test senza un'accurata descrizione dell'ambiente in cui esso è stato svolto è inutile perchè, nel mondo dell'informatica, le prestazioni degli elaboratori e le tecnologie adottate variano così velocemente che i risultati di un test condotto oggi potrebbero venir drasticamente ridimensionati o addirittura sovvertiti dall'hardware di domani[1].

Da un punto di vista hardware, tutti i test sono stati svolti sullo stesso elaboratore dotato di processore Intel® Core™ 2 Duo T7250 a 2.00 GHz con 2048 kB di memoria cache e una quantità totale di memoria RAM pari a 2 GB. I dati sulla memoria, dal punto di vista dei test fatti, sono utili solo relativamente, in quanto non verranno usate grosse moli di dati, quanto piuttosto un numero elevato di thread da dover gestire. Sopra lo strato fisico si trova un sistema Linux a 32bit basato sulla distribuzione Slackware 12.2 ma montante il kernel 2.6.34.1 compilato, nella sua versione *vanilla*[2], per l'architettura corrente. Essendo il codice scritto interamente in Java™ , è bene specificare anche quale sia la versione della Java Virtual Machine che fa da interprete, e, in questo, caso si parla di Java™ SE Runtime Environment (build 1.6.0_16-b01) in esecuzione su Java HotSpot™ Server VM (build 14.2-b01, mixed mode). La scelta di questa versione dell'interprete, rispetto a quella usata per lo sviluppo del codice - più vecchia - è dovuta alla correzione dell'algoritmo di gestione del lock intrinseco [3] che avrebbe potuto falsare i risultati. Per restare più fedeli possibile a quella che è la realtà di utilizzo, inoltre, tutti i test sono stati eseguiti all'interno della piattaforma *PariPari* (versione del Core 1.11.201006032255 t.a.l.p.a) utilizzando un plug-in appositamente creato per lo scopo.

La maggior parte dei test sono basati sull'uso della classe `WorkerManager` in quanto, in primo luogo, l'attuale implementazione sfrutta al suo interno un `SynchronizableManager`, permettendo un test combinato di entrambi questi strumenti; in secondo luogo, `WorkerManager`

[1] Si pensi alla continua espansione della cache interna dei microprocessori che sta favorendo sempre più algoritmi ricorsivi a discapito di quelli iterativi, da sempre considerati più veloci.

[2] Per kernel vanilla si intende il codice sorgente rilasciato dalla Linux Kernel Organization, Inc. attraverso sito `www.kernel.org` senza patch o modifiche di terze parti.

[3] [4, cfr. 13.2 Performance Consideration]

è il manager effettivamente impiegato all'interno del codice per tutti i meccanismi di sincronia, e, quindi, è quello che deve garantire le maggiori prestazioni.

5.2 Attesa di un singolo task

Il primo test che è stato sviluppato mira a valutare l'efficacia della struttura di sincronia nei confronti dell'attesa di un singolo thread. Per permettere un facile confronto tra i vari strumenti testati, l'efficacia è stata misurata in termini di *tempo di risposta* (response time), ovvero il periodo di tempo che intercorre tra la fine del task lavoratore e l'inizio dello sfruttamento dei suoi risultati; ne consegue che più un sistema è sincronizzato, minore è il tempo di risposta fra i suoi componenti.

Descrizione del test Questo primo test si svolge coinvolgendo sempre due thread: un produttore/consumatore e un lavoratore. Il produttore lancia il worker e si mette subito in attesa del suo completamento, mentre il thread worker, una volta avviato, simula lo svolgimento di un'elaborazione, terminando dopo un'attesa prefissata. L'obiettivo è osservare come varia il tempo di risposta all'aumentare del periodo di occupazione del thread worker, utilizzando i tre metodi visti per questo scopo, ovvero busy waiting con backoff esponenziale, wait & signal, e manager, dove gli ultimi due sono offerti nativamente dall'infrastruttura di sincronia. Per quanto riguarda il busy waiting esso usa 2^i come fattore per calcolare il tempo di attesa all'i-esima iterazione, a partire da un tempo base di $500ms$ e fino ad arrivare ad un'attesa massima di $8000ms$; i $500ms$ di tempo base sono stati scelti in quanto adottati tipicamente nell'implementazione corrente del plug-in.

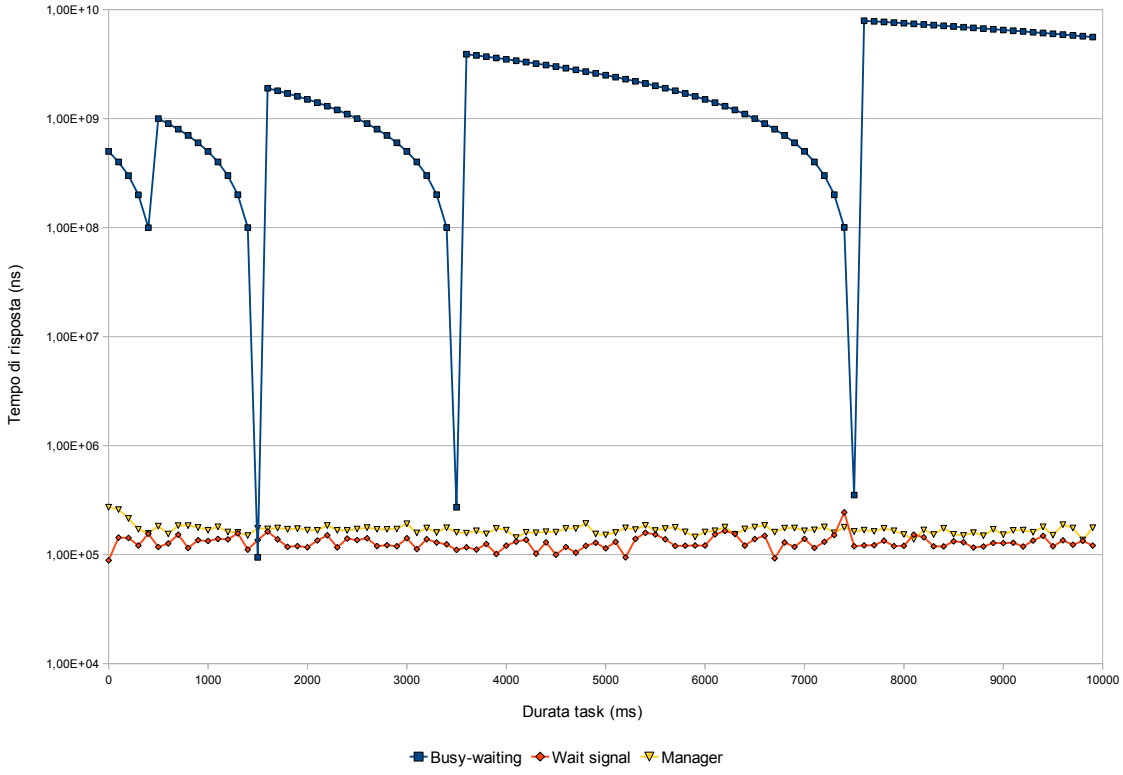


Figura 5.1: Tempo di risposta alla terminazione di un singolo thread lavoratore.

Commento sui risultati La prima cosa che si nota osservando il grafico in figura 5.1 è l'andamento del tempo di risposta del busy waiting: esso presenta una forma simile a lobi in quanto, man mano che l'attimo di terminazione del thread lavoratore si avvicina ad uno dei risvegli "programmati" del busy waiting, il tempo di risposta ovviamente diminuisce, ma, appena l'attimo di terminazione supera il risveglio programmato, per poter sfruttare i risultati si deve attendere il risveglio successivo. Inoltre, la larghezza di ogni lobo è sempre doppia, poichè il busy waiting usa 2^i come fattore di scala per calcolare il prossimo tempo di attesa; a riprova di quanto affermato nella descrizione del test, inoltre, è possibile notare che i minimi di questo grafico si trovano attorno ai valori $500ms$, $1500ms = 500 + 500 * 2$, $3500ms = 1500 + 500 * 2^2$, $7500ms = 3500 + 500 * 2^3$.

Il grafico permette inoltre di osservare che gli strumenti offerti dall'infrastruttura di sincronia sono indipendenti dalla durata del worker e, mediamente, di tre ordini di grandezza più responsivi rispetto al busy waiting. É corretto parlare di media in quanto, se i workers terminassero tutti un'istante prima del risveglio del busy waiting, il suo grafico potrebbe persino stare sempre al di sotto degli altri due; questa, però, è un'eventualità che difficilmente si può verificare, in un ambiente fortemente dinamico e imprevedibile come le comunicazioni di rete. Ciò nonostante, è possibile ridurre questo divario modificando i parametri di attesa del busy-waiting, usando, ad esempio, un tempo base leggermente superiore alla media delle tempistiche ipotizzate per i workers e un fattore di backoff casuale nell'intervallo $[1; 2^i]$.

Si può notare, infine, che l'utilizzo del manager comporta una responsività leggermente minore rispetto al semplice wait & signal: questo comportamento è da imputare al fatto che il manager, avendo una struttura più articolata, necessita di un numero maggiore di operazioni, prima di permettere al thread consumatore di prelevare il worker terminato.

5.3 Attesa di almeno un task

Questo secondo test è stato sviluppato per valutare l'efficacia del manager e del busy waiting nel caso si abbia un insieme di thread lavoratori, si debba attendere la terminazione di almeno un worker di questo insieme, per poi ripetere questa attesa finchè l'insieme non risulti composto solamente da workers terminati. L'efficacia è qui valutata in termini di tempo di risposta medio dell'intero insieme dei thread lavoratori.

Descrizione del test Con questo test si vuole rispecchiare il più possibile la realtà di utilizzo: viene quindi adottato un unico thread produttore/consumatore ed un numero via via maggiore di task lavoratori che, a differenza di prima, simulano il loro lavoro con un'attesa casuale uniformemente distribuita tra $0ms$ e $7600ms$. La granularità del millisecondo per i tempi di attesa è stata scelta perchè PariDHT, lavorando a stretto contatto con la rete, ne condivide le tempistiche, che si attestano sull'ordine delle decine o centinaia di millisecondi; il valore massimo invece è stato posto appena oltre il 4°picco di minimo del grafico precedente e, nonostante sia di un ordine di grandezza superiore alla latenza media della rete, esso rientra nei tempi di attesa tipici del plug-in, che si verificano, ad esempio, nell'attesa di una serie di operazioni di ricerca. Il busy waiting mantiene i parametri precedenti, quindi, un fattore di backoff costante 2^i, un tempo base di $500ms$ e un'attesa massima di $8000ms$; per quanto riguarda il manager, invece, verranno testati i metodi `waitAny()` e `bookAny()`, avendo cura, in quest'ultimo caso, ad eseguire la terminazione del manager una volta inseriti tutti i thread lavoratori.

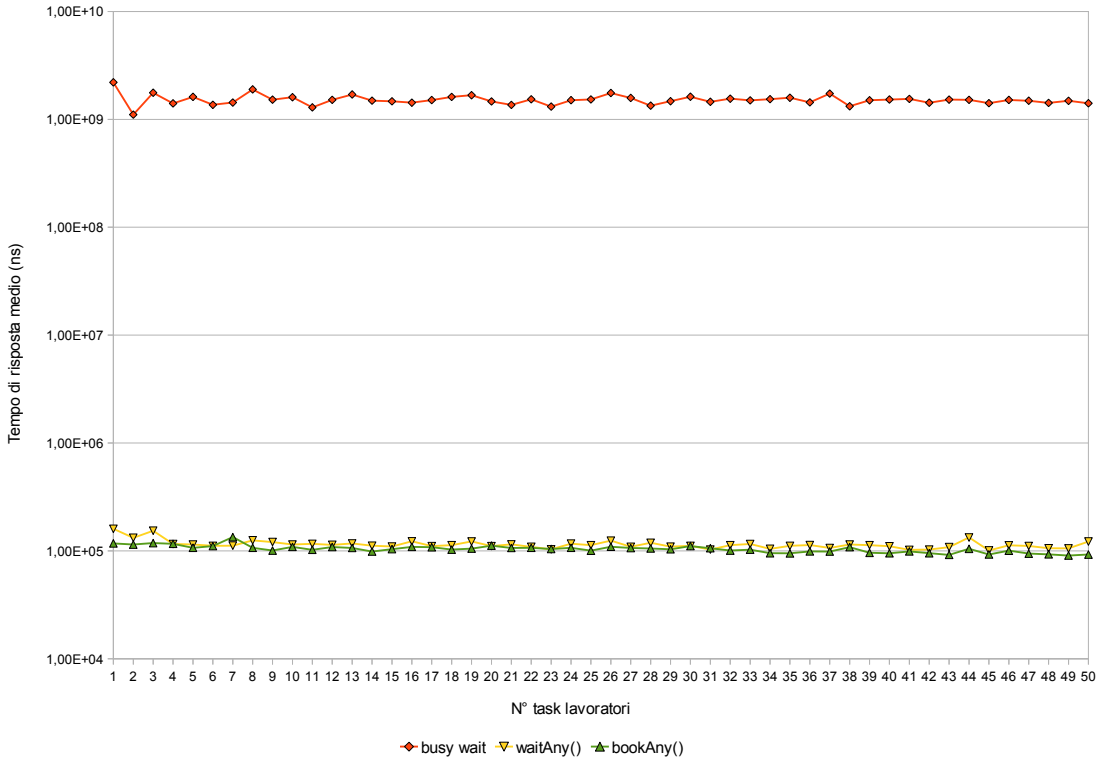


Figura 5.2: Tempo di risposta medio alla terminazione di un insieme di thread lavoratori.

Commento sui risultati In figura 5.2 è presentato l'andamento del tempo di risposta medio. Tutti e tre i grafici tendono alla linearità all'aumentare del numero di task lavoratori introdotti, seppure il busy waiting sia di ben 4 ordini di grandezza meno efficiente dei metodi offerti dal manager e soffra di maggiori variazioni (è da tener presente che la scala dei tempi di risposta è logaritmica). Per quanto riguarda il manager questo comportamento è prevedibile e in linea con i risultati ottenuti nel test precedente in quanto il tempo di risposta è indipendente dalla durata del worker e l'implementazione garantisce overhead quasi nullo in presenza di più lavoratori; il busy waiting, invece, vede limati i suoi minimi, ottenendo però una responsività più lineare: anche in questo caso, comunque, per colmare il divario tra manager e busy waiting è possibile sfruttare gli accorgimenti già esposti nel commento al test precedente.

5.4 Stress test del manager

La serie di test condotti finora su `WorkerManager`, però, non ha ancora permesso di verificare la nuova dinamica offerta da questa classe, ovvero, la possibilità di sfruttare più task consumatori, per estrarre i workers terminati dal manager. Un riscontro in tal senso è fondamentale per poter affermare che, sfruttando più consumatori, si ottengono dei vantaggi apprezzabili; detto questo, però, è necessario anche capire quali siano i limiti di un simile approccio. L'unità di misura della "bontà" dell'esecuzione è, ancora una volta, il tempo di risposta medio, di cui ci si aspetta una diminuzione, all'aumentare del numero di task consumatori.

Descrizione del test Per valutare efficacemente il lavoro del manager, si è cercato di porlo nella condizione di massima concorrenza: 2 thread produttori sono incaricati di registrare presso il manager un totale di 96 task worker, a cui, un numero variabile di consumatori, sono incaricati di far fronte. Si è scelto 96 come tetto per il numero di workers dato che, l'attuale versione di Core mette a disposizione, nel thread-pool assegnato ad ogni plug-in, un massimo di 100 thread: così limitati, i workers non potranno mai impedire ad almeno un thread consumatore di avviarsi e, quindi, al test di completarsi correttamente[4].

Una valutazione corretta delle performance, però, richiede, oltre al numero di consumatori, qualche elemento variabile in più. Per cercare di capire fino a che punto l'aumento dei consumatori porta dei vantaggi nell'esecuzione, questo test è stato eseguito con diverse casistiche di utilizzo, basate sulla variazione dei tempi di elaborazione di workers e consumatori stessi. I casi modellati sono stati di 3 tipi:

- lavoro dei workers di un ordine di grandezza maggiore, in termini di tempo, rispetto a quello dei consumatori;
- lavoro dei consumatori di un ordine di grandezza maggiore rispetto a quello dei workers;
- lavoro di consumatori e workers sullo stesso ordine di grandezza.

Per quanto riguarda i consumatori, l'attesa che modella il lavoro svolto è costante, mentre per i workers è intesa come valore massimo, in quanto si è deciso di mantenere casuali gli istanti della loro terminazione.

Commento sui risultati In figura 5.3 sono riportati i grafici che rappresentano gli andamenti dei tempi di risposta medi al variare del numero di task booker. Tralasciando un momento il secondo grafico, appiattito sull'asse delle ascisse, notiamo che tutti gli altri seguono l'andamento di un ramo di iperbole, tipico delle funzioni inversamente proporzionali, e, in questo caso, di quelle di primo grado: $y = k/x$. Ciò significa che, nel tratto di grafico presentato, raddoppiando il numero di task consumatori, il tempo medio impiegato per servire un worker terminato, si dimezza.

Osservando nuovamente il grafico, è possibile dedurre anche le variazioni della costante di proporzionalità k: *aumenta* all'aumentare del tempo necessario a un consumatore, per servire un worker, oppure, a parità di tempo di servizio, riducendo il divario tra tempo di servizio stesso e durata dell'elaborazione dei workers; *diminuisce* in caso contrario. Queste informazioni permettono ad uno sviluppatore di mantenere costante l'efficienza del suo codice in seguito ad eventuali riscritture dei thread worker o booker: se le riscritture comportano un aumento di k allora, per non diminuire l'efficienza, è necessario aumentare il numero di task consumatori; d'altra parte, se il refactoring porta con se una diminuzione di k, è possibile ipotizzare una diminuzione di questi task.

In ultima analisi, viene preso in esame il secondo grafico, il quale presenta un andamento pressochè lineare, al variare del numero di task consumatori. Questa osservazione, unita alle informazioni sulle tempistiche della legenda e a quanto appena detto su k, permette di affermare che: non si ha alcun giovamento nell'aumentare il numero di thread consumatori, qualora il tempo di servizio risultasse *trascurabile* rispetto a quello impiegato dai worker. La trascurabilità è un concetto strettamente legato all'hardware, ma,

[4]Un valore maggiore di 96, tenendo conto dei 2 thread produttori e del thread di controllo, avrebbe potuto impedire ai consumatori di avviarsi, bloccando il test.

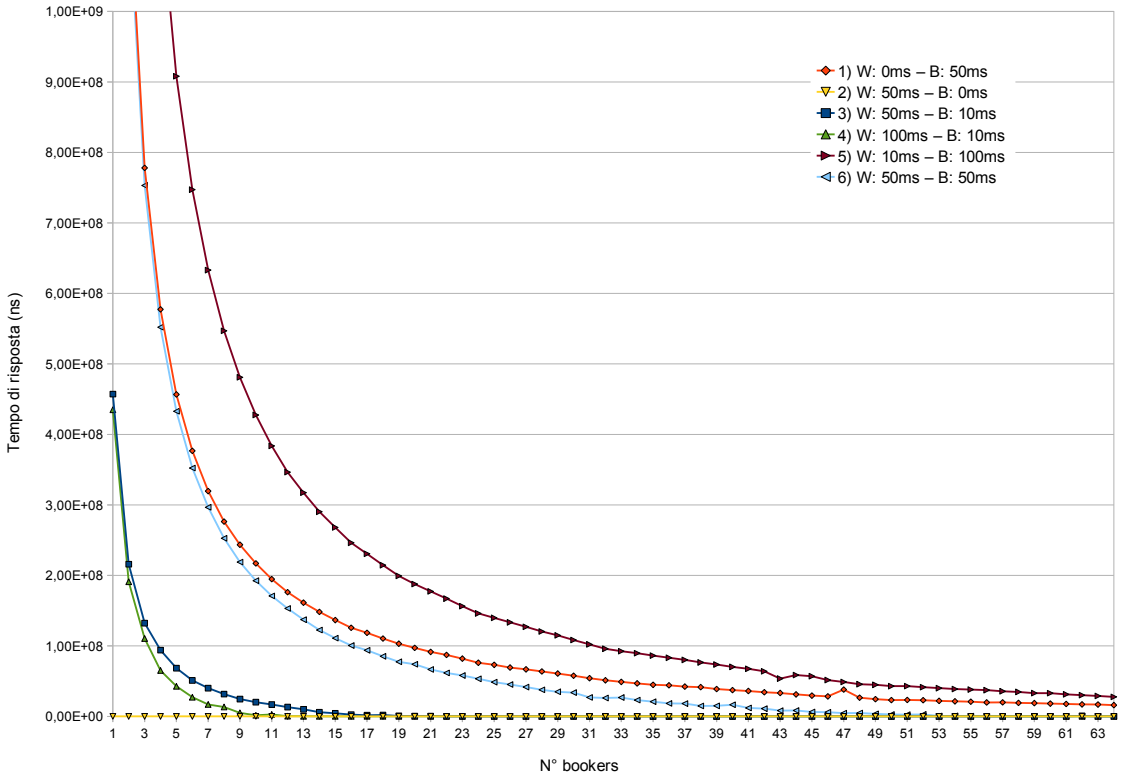


Figura 5.3: Tempo di risposta medio alla terminazione di un insieme di workers, variando il numero di thread booker.

grazie ad ulteriori test non riportati nel grafico 5.3, per non appesantirlo, è stato possibile quantificarla, per l'ambiente di esecuzione di riferimento, in circa tre ordini di grandezza.

Conclusioni e sviluppi futuri

In questo elaborato si è descritto un possibile sistema di sincronizzazione per il plug-in PariDHT, che, con piccoli accorgimenti, è possibile inserire all'interno di qualsiasi plug-in della piattaforma *PariPari*. Questa sua adattabilità è stata uno dei cardini dello sviluppo, e lo stesso dicasi delle performance, della robustezza e della facilità d'uso, dal momento che l'intera infrastruttura doveva essere adottata da un plug-in la cui stesura era già in corso. Un simile approccio, comunque, andrebbe evitato, in quanto, se si punta allo sviluppo di un'applicazione multi-threaded, la definizione degli attori, del ruolo che giocheranno, e delle loro interdipendenze, deve essere fatta in modo chiaro già in fase di progettazione. Così facendo, è possibile individuare eventuali task superflui, ovvero che non servono ad aumentare la parallelizzazione del lavoro, e si possono stabilire quali operazioni devono essere sincrone o asincrone, definendo così, in modo uniforme, le modalità di gestione di queste attività da parte dell'intero codice. Nonostante questa aggiunta postuma, comunque, le performance hanno raggiunto gli obiettivi prefissati.

Un altro cardine dello sviluppo è stato, come già detto, la facilità d'uso: semplicità e concorrenza sono spesso un binomio in constrasto ma, vista la modalità di sviluppo, ad elevata rotazione di partecipanti, si è reso necessario fornire una buona astrazione rispetto all'implementazione sottostante, per rendere intuitivo, da scrivere e quindi poi anche da leggere, il codice che utilizzerà questi strumenti.

Nel caso però, in cui l'intuitività non fosse evidente come sperato, è comunque possibile rifarsi ad un'estesa documentazione di tutte le classi e i metodi introdotti, riportata direttamente all'interno del codice tramite lo standard Javadoc. Se anche la documentazione risultasse ambigua, l'ultima spiaggia con cui un futuro programmatore può confrontarsi è la robustezza dell'implementazione, la quale dovrebbe riuscire ad impedire eventuali usi impropri degli strumenti a disposizione e suggerirne le soluzioni.

Per quanto riguarda invece i possibili sviluppi futuri, alla luce di quanto si è fatto finora, la definizione dei task in gioco in PariDHT potrebbe essere rivista nelle modalità che verranno descritte qui di seguito:

- *le richieste provenienti da Core sono trattate in modo seriale*: questo comportamento potrebbe venir parallelizzato fissando un tetto massimo di elaborazioni contemporanee, permettendo così di diminuire i tempi di attesa da parte degli altri plug-in, rendendo inoltre superflua la classe-thread `Dispatcher`, e, consentendo in tal modo di risparmiare un task di servizio. Per raggiungere questo obiettivo, viste le limitazioni imposte da Core, non è possibile sfruttare un thread pool, sicchè la gestione dei task "serventi" deve essere fatta manualmente; si propone pertanto l'utilizzo di un semaforo numerico.

- `OutRequest` *è un task dormiente per la maggior parte*[5] *del suo ciclo vitale*: ciò non

[5]Test svolti con l'ausilio dell'Eclipse Test and Performance Tools Platform (TPTP), mostrano che l'attesa si attesta su di una percentuale, rispetto al tempo totale di utilizzo della classe, prossima al 100%

è necessariamente un problema - si pensi ai task di servizio che sono nella medesima condizione; ciò non toglie, però, che si possano ipotizzare dei miglioramenti in termini di riduzione della `OutRequest` a classe passiva, aumentando leggermente il carico di lavoro di `NetSender`, `NetReceiver`, e di tutte quelle classi che ne sfruttano internamente i servizi. Ciò permetterebbe una consistente diminuzione del numero di thread attivati durante il normale svolgimento delle funzioni di PariDHT, rendendo il limite di thread imposto da Core un problema trascurabile, soprattutto nel caso di un numero elevato di richieste da parte degli altri plug-in.

- *le richieste da parte degli altri plug-in dovrebbero venir tutte svolte in modo asincrono*: questo permetterebbe loro di commissionarne un numero maggiore e di continuare a svolgere i loro compiti, finchè i risultati che PariDHT deve fornire impediscono ulteriori sviluppi. Contestualmente ad un miglioramento di questo tipo, però, è necessario offrire anche un sistema che permetta agli utilizzatori di "sincronizzarsi" (ovvero attendere fino al loro completamento) con queste operazioni asincrone, possibilmente tramite gli stessi strumenti già delineati in questo testo.

Infine, un'ultima proposta, che però si svincola dalla realtà di PariDHT, potrebbe essere l'*adozione, in tutta PariPari, della struttura di sincronia proposta.* Al momento, *PariPari* non possiede un sistema omogeneo per la gestione della concorrenza; tuttavia, diversi plug-in potrebbero aver adottato sistemi simili per risolvere i loro problemi di sincronizzazione. La soluzione quindi potrebbe essere, mediante le librerie offerte da Core, la fornitura di un set standard di questi strumenti, eventualmente integrati per rispondere alle necessità dei vari plug-in, permettendo in tal modo una maggiore efficienza e correttezza del codice di tutta la piattaforma.

Bibliografia

[1] Soma Chaudhuri. Advanced Topics in Algorithms: Algorithms for Multiprocessor Synchronization. `http://www.cs.iastate.edu/~chaudhur/cs611/`.

[2] Wikimedia Foundation. WIKIPEDIA: The Free Encyclopedia. `http://www.wikipedia.org/`.

[3] Simone Giacon. *PariPari: DHT 2008*. PhD thesis, Università degli Studi di Padova, dipartimento di Ingegneria dell'Informazione, A.A. 2008/2009.

[4] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Addison Wesley Professional, May 2006.

[5] Petar Maymounkov and David Mazières. Kademlia: A Peer-to-peer Information System Based on the XOR Metric. `http://kademlia.scs.cs.nyu.edu`.

[6] Oracle. *JDK™ 5.0 Documentation*. `http://download.oracle.com/javase/1.5.0/docs/`.