

UNIVERSITÀ DI PISA



FACOLTÀ DI SCIENZE MATEMATICHE FISICHE E NATURALI  
CORSO DI LAUREA IN INFORMATICA

Relazione di tirocinio

# Un orchestratore per XPeer

Luca Pardini

Tutore accademico  
Prof. Giorgio Ghelli

Anno Accademico 2005/2006

# Indice

<b>1</b>	<b>Introduzione</b>	<b>1</b>
1.1	Il sistema XPeer . . . . .	2
1.2	L'orchestratore . . . . .	2
1.3	Descrizione del lavoro svolto . . . . .	3
1.4	Organizzazione della relazione . . . . .	3
<b>2</b>	<b>Architettura dell'Orchestratore</b>	<b>4</b>
2.1	Il sistema XPeer . . . . .	5
2.2	Architettura generale . . . . .	6
2.2.1	Protocolli di comunicazione . . . . .	6
2.3	Architettura del Proxy . . . . .	7
2.3.1	Comandi . . . . .	8
2.4	Architettura dell'Orchestratore . . . . .	12
<b>3</b>	<b>XOL: Un linguaggio di orchestrazione</b>	<b>15</b>
3.1	I linguaggi concorrenti . . . . .	15
3.1.1	Il $\pi$ -calcolo . . . . .	16
3.2	La libreria JCSP . . . . .	17
3.3	Il linguaggio XOL . . . . .	17
3.3.1	Sintassi . . . . .	18
3.3.2	Semantica . . . . .	19
3.3.3	Casi d'uso . . . . .	21
3.3.4	Regole di compilazione . . . . .	26

---

<b>4</b>	<b>Aspetti implementativi</b>	<b>35</b>
4.1	La libreria JCSP . . . . .	35
4.1.1	I canali . . . . .	35
4.2	Implementazione del Proxy . . . . .	36
4.2.1	Configurazione del Proxy . . . . .	37
4.3	Implementazione dell'Orchestratore . . . . .	38
4.3.1	L'interfaccia grafica . . . . .	39
4.3.2	Configurazione dell'Orchestratore . . . . .	39
<b>5</b>	<b>Conclusioni</b>	<b>41</b>
	<b>Bibliografia</b>	<b>43</b>

# Elenco delle figure

2.1	Rappresentazione di una ipotetica rete di XPeer . . . . .	5
2.2	Diagramma di deployment del sistema . . . . .	6
2.3	Protocollo di comando dal punto di vista del Proxy . . . . .	9
2.4	Diagramma di attività del Proxy . . . . .	10
2.5	Diagramma di attività del gestore dei Proxy . . . . .	13
3.1	Esecuzione di un programma XOL . . . . .	18

# Elenco delle tabelle

2.1	Comandi inviabili da un Proxy . . . . .	11
4.1	Proprietà modificabili nell'orchestratore . . . . .	40

# Capitolo 1

## Introduzione

La nascita dei sistemi Peer-to-Peer (P2P) ha suscitato molto interesse per i notevoli cambiamenti introdotti nelle architetture dei sistemi distribuiti che si rifacevano prevalentemente al modello client-server.

Un sistema P2P è costituito da un insieme di nodi autonomi (peer) che condividono un insieme di risorse all'interno di una rete di computer.

Una rete P2P estende il concetto di client-server; le comunicazioni non sono solo verso alcuni server prefissati, ma tra nodi qualsiasi: un peer ha sia funzionalità da client, richiedendo risorse agli altri, sia funzionalità da server, offrendo le proprie risorse a coloro che ne fanno richiesta.

Questo tipo di sistemi si è molto diffuso, complici anche le connessioni a banda larga, grazie ai suoi principali vantaggi:

- Replicazione e disponibilità dei dati: permettono facilmente di trovare dati d'interesse e recuperarli in caso di perdita accidentale.
- Capacità di adattamento: la rete gestisce bene i numerosi ingressi e uscite da parte degli utenti, adattandosi di conseguenza.
- Bassi costi di amministrazione: i costi sono distribuiti tra ogni peer e sono nettamente inferiori ai pesanti sforzi richiesti da un server centralizzato.

Un sistema P2P presenta però alcuni svantaggi, come ad esempio il costo potenzialmente elevato delle operazioni di *resource-discovery*.

Un problema fondamentale che si presenta durante la costruzione di un nuovo sistema P2P, è la difficoltà nell'eseguire test significativi per verificare sia la robustezza che la correttezza degli algoritmi usati.

## 1.1 Il sistema XPeer

XPeer è un database distribuito sviluppato dal gruppo di ricerca di Basi di Dati del Dipartimento di Informatica dell'Università di Pisa.

Esso può essere usato per raccogliere qualunque tipo di informazioni purché codificate con il formato XML. Per le interrogazioni su questi dati viene utilizzato il sottoinsieme FLWR di XQuery, linguaggio d'interrogazione standard per i dati XML sviluppato da W3C.

La rete di XPeer utilizza una architettura P2P ibrida nella quale sono presenti sia nodi dedicati alla gestione dei dati che nodi che si occupano della gestione della rete. La struttura di questa rete è organizzata ad albero, dove il livello delle foglie è costituito da tutti i peer connessi alla rete, mentre gli altri nodi dell'albero si occupano di gestire l'evoluzione della rete e di instradare le interrogazioni.

Questo approccio ci permette di evitare l'inondamento di richieste dai nodi foglia, tipico dei sistemi P2P puri, ovvero di quei sistemi con una rete non strutturata.

## 1.2 L'orchestratore

La creazione del sistema XPeer ha portato alla necessità di testare gli algoritmi progettati e implementati in una possibile e reale applicazione del sistema.

Per fare ciò non è pensabile doversi occupare, ogni volta, di creare una rete reale di nodi, lanciando su ogni macchina "a mano" ciascun nodo, ma si è reso necessario un sistema centralizzato, detto **orchestratore**, che permette di ripetere gli esperimenti in modo automatico.

Tale sistema deve quindi poter leggere un file speciale, detto *spartito*, che raccolga il comportamento che devono avere i vari nodi, ed eseguirlo. Inoltre il sistema deve raccogliere le informazioni utili per valutare il risultato dell'esper-

imento per dare la possibilità a chi lo utilizza di capire se il sistema ha reagito correttamente o in modo anomalo.

### 1.3 Descrizione del lavoro svolto

Durante il tirocinio mi sono occupato della progettazione, realizzazione e testing dell'orchestratore, ho anche ideato un linguaggio di specifica, che facilita la scrittura degli spartiti da utilizzare, prendendo spunto da un'algebra di processi.

Infine, mi sono occupato dei primi e rudimentali test del sistema XPeer, in modo da verificare l'effettiva utilità del sistema da me progettato e realizzato.

### 1.4 Organizzazione della relazione

La relazione è strutturata nel modo seguente:

- nel Capitolo 2 parleremo dell'architettura del sistema di orchestrazione;
- nel Capitolo 3 introdurremo le algebre di processi e parleremo del linguaggio da me ideato per descrivere il comportamento del sistema XPeer durante un test;
- nel Capitolo 4 descriveremo gli aspetti implementativi di questo sistema.

# Capitolo 2

## Architettura dell'Orchestratore

L'Orchestratore è un sistema in grado di eseguire dei test di funzionamento sul sistema XPeer [SMGC04]. Per farlo seguirà le direttive contenute in un file, che chiameremo *spartito*, contenente una serie di istruzioni da eseguire, con alcune dipendenze temporali per creare configurazioni, anche molto complicate, di eventi per il sistema XPeer, quali attivazione di nuovi nodi, connessioni tra i nodi, richieste di compilazione ecc.

L'intero sistema di orchestrazione è così suddiviso:

- un gestore per ogni nodo della rete, che chiameremo *Proxy*, con il compito di ricevere ed eseguire istruzioni inviate dall'Orchestratore;
- un'entità, che chiameremo *Orchestratore*, con il compito di caricare ed eseguire gli spartiti, e di tener traccia di tutti i Proxy che sono stati avviati;
- un insieme di protocolli di comunicazione tra l'Orchestratore e i vari Proxy;
- un linguaggio con il quale poter scrivere gli spartiti da far eseguire all'Orchestratore.

In questo capitolo introdurremo brevemente alcune principali caratteristiche del sistema XPeer e parleremo approfonditamente dei primi tre punti che corrispondono all'architettura del sistema; dedicheremo successivamente un capitolo per il linguaggio utilizzato negli spartiti.

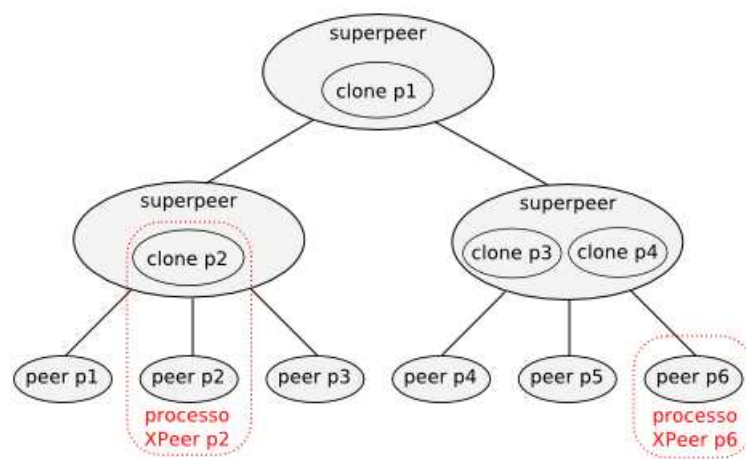
## 2.1 Il sistema XPeer

XPeer è un sistema P2P basato su una architettura ibrida strutturata ad albero dove le foglie sono costituite da nodi della rete (peer) e i nodi intermedi dell'albero sono costituiti da insiemi di nodi che si occupano della gestione (superpeer).

I peer sono i nodi della rete che esportano dati XML e che eseguono le interrogazioni. Per far parte della rete un peer deve essere connesso a un superpeer.

I superpeer sono entità virtuali composte da uno o più cloni. Queste entità si occupano di gestire l'evoluzione della rete e di instradare le interrogazioni.

Ogni peer della rete può diventare clone di un superpeer (inteso come insieme di cloni) mantenendo la sua identità di peer. Ciò significa che ogni istanza del sistema XPeer (da qui in avanti verrà chiamata processo XPeer) è divisa in due entità: peer e superpeer (inteso come clone). L'entità peer è sempre attiva e rappresenta un nodo foglia della rete. L'entità superpeer inizialmente non è attiva, ma può venir attivata su richiesta di un superpeer già presente nella rete entrando a far parte dell'insieme dei cloni del superpeer che l'ha attivata. In questo secondo caso un processo XPeer corrisponderà a due diversi nodi della stessa rete, uno come foglia (entità peer) e uno come clone di un insieme di nodi gestori (entità superpeer).



**Figura 2.1:** *Rappresentazione di una ipotetica rete di XPeer*

## 2.2 Architettura generale

Il sistema di orchestrazione è un sistema distribuito su un insieme di macchine in grado di comunicare utilizzando il protocollo tcp/ip. Una macchina, che chiameremo PC Orchestratore, si dedica all'esecuzione di un'istanza (artifact) della componente Orchestratore mentre le restanti macchine, che chiameremo PC Proxy, si occupano di eseguire un'istanza della componente Proxy.

La componente Proxy contiene un processo XPeer, cioè un peer più l'eventuale attivazione del superpeer associato, e comunicano con l'Orchestratore attraverso dei canali distinti realizzati sopra il protocollo tcp/ip, come vediamo in figura 2.2.

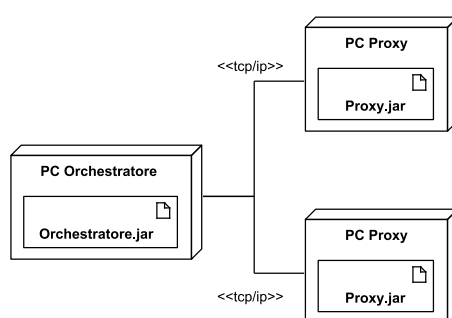


Figura 2.2: Diagramma di deployment del sistema

### 2.2.1 Protocolli di comunicazione

Le comunicazioni tra Proxy e Orchestratore avvengono attraverso tre protocolli: il protocollo di connessione, il protocollo di comando e il protocollo di abbonamento.

#### Protocollo di connessione

Il protocollo di connessione si rende necessario perché, inizialmente, l'Orchestratore non ha idea di quali Proxy siano stati attivati, quindi, ha bisogno di un meccanismo per mettersi in contatto con ogni singolo Proxy.

L'Orchestratore crea un canale di input *molti a uno* al fine di ricevere dai Proxy notifiche di attivazione. Dall'altra parte ogni singolo Proxy, che assumiamo conosca il canale di input dell'Orchestratore, crea un proprio canale di input *uno a*

*uno* e lo invia all'Orchestratore inserendolo nel messaggio di notifica di attivazione. In questo modo, l'Orchestratore può estrarre il canale del Proxy dal suo messaggio di attivazione e utilizzarlo subito per rispondere con un messaggio di conferma.

### **Protocollo di comando**

Il protocollo di comando si rende necessario quando l'Orchestratore vuol far eseguire una certa operazione a un Proxy che conosce, ovvero che ha precedentemente eseguito un protocollo di connessione.

Per comunicare con il Proxy, l'Orchestratore utilizza il canale ricevuto durante il protocollo di connessione e realizza un protocollo a domanda/risposta, dove la domanda consiste nel comando da svolgere e la risposta consiste nel risultato di tale operazione o, laddove il risultato non sia previsto, un ack di conferma. Inoltre, dato che l'Orchestratore è multithread, è possibile porre molte domande in parallelo.

### **Protocollo di abbonamento**

Il protocollo di abbonamento si rende necessario quando l'Orchestratore vuole ricevere continuamente informazioni da un Proxy che conosce.

Per innescare questo protocollo è necessario che l'Orchestratore invii al Proxy un comando (tramite il protocollo di comando) contenente quali sono le informazioni interessanti e su quale canale devono essere inviate. Una volta innescato, il protocollo consiste semplicemente nell'invio da parte del Proxy di tutte le informazioni richieste sul canale indicato. Il Proxy invia queste informazioni appena vengono generate.

Per disinnescare questo protocollo è sufficiente inviare un secondo comando, diverso dal precedente, per informare il Proxy che le informazioni che stava inviando non sono più necessarie.

## **2.3 Architettura del Proxy**

Il Proxy rappresenta il componente più semplice del sistema di orchestrazione, essendo poco più di un interprete di comandi che invoca procedure del sistema

XPeer. Essendo un interprete, il Proxy è un'entità che non termina mai: una volta avviato, dopo ogni comando ricevuto ne aspetta sempre uno nuovo.

Il Proxy è composto da:

- un oggetto, denominato *connector*, che si occupa di gestire la connessione e disconnessione tra Proxy e Orchestratore;
- l'interprete vero e proprio, denominato *proxymonitor*, che si occupa di ricevere i messaggi sul canale di input, di decodificarli e, se sono corretti, di inviarli a un esecutore;
- un insieme di esecutori, denominati *executorthread*, che si occupano di effettuare un comando invocando le procedure di XPeer opportune.

Un Proxy, appena viene avviato, attua il protocollo di connessione tramite il connector e contemporaneamente si mette in attesa di un comando tramite il proxymonitor. Ricevuto un comando, il proxymonitor lo affida a un executorthread e si rimette subito in attesa di un nuovo comando. L'executorthread controlla la correttezza del comando e, se corretto, lo esegue, occupandosi di completare il protocollo di comando spedendo all'Orchestratore il risultato dell'operazione, che potrebbe anche essere un messaggio di errore (figura 2.3).

Il parallelismo introdotto tra la ricezione di un comando e l'esecuzione dello stesso è dovuto alla possibilità di eseguire concorrentemente più di un comando, per esempio due interrogazioni. Il comportamento di un Proxy è sintetizzato in figura 2.4.

### 2.3.1 Comandi

Per comando si indica la coppia composta da un'istruzione e dal canale dove viene inviato il risultato dell'istruzione.

Ogni Proxy può eseguire esclusivamente un determinato sottoinsieme delle istruzioni elencate nella tabella 2.1, in base ai prerequisiti richiesti.

Ogni istruzione è composta da:

- il nome dell'istruzione;

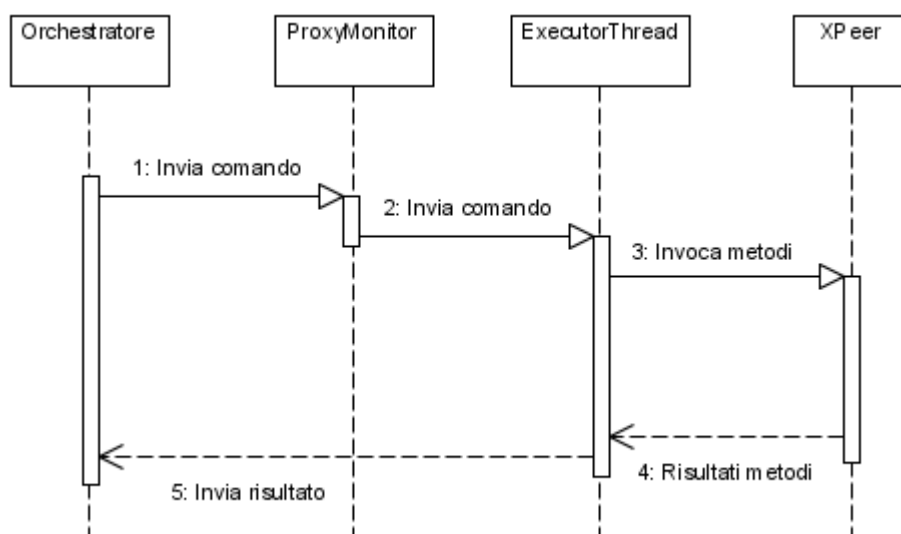


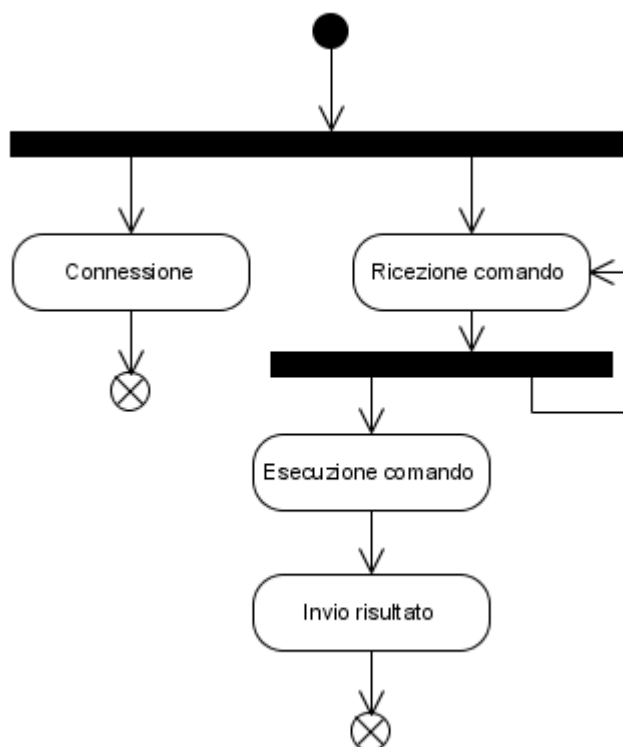
Figura 2.3: Protocollo di comando dal punto di vista del Proxy

- una lista di parametri;
- alcuni prerequisiti necessari per l'esecuzione, detti *precondizioni*.

Le precondizioni si rendono necessarie per come è strutturato il sistema XPeer. Infatti, alcune istruzioni del processo XPeer possono essere eseguite correttamente soltanto se tale processo si trova in un determinato stato. Per fare un esempio, non è possibile far compilare una interrogazione a un peer se prima non è stato avviato.

Le istruzioni hanno la seguente semantica informale:

- l'istruzione *start* serve per lanciare il processo XPeer contenente soltanto l'entità peer;
- l'istruzione *setDataSource* serve per impostare il datasource come documento xml esportato dal peer verso la rete;
- l'istruzione *close* serve per chiudere il processo XPeer;
- l'istruzione *connect* serve per connettere l'entità peer di un processo XPeer a un'entità superpeer;



**Figura 2.4:** Diagramma di attività del Proxy

- l'istruzione *superConnect* serve per connettere l'entità superpeer di un processo XPeer a un'altra entità superpeer;
- l'istruzione *becomeSP* serve per attivare l'entità superpeer in un processo XPeer;
- l'istruzione *compileQuery* serve per far compilare una interrogazione a un peer di un processo XPeer, il risultato sarà l'interrogazione compilata;
- l'istruzione *executeQuery* serve per avviare una compilazione ed esecuzione di una interrogazione su un peer. I risultati dell'esecuzione verranno inviati appena vengono generati o alla fine dell'esecuzione (a seconda del valore del parametro di caching se il caching è impostato a vero) sul canale per i risultati passato come parametro;
- l'istruzione *subscribe* serve informare il Proxy che deve inviare tutti i record

di log del livello passato come parametro, sul canale passato anch'esso come parametro;

- l'istruzione *unsubscribe* serve per non ricevere più i record di log richiesti

Istruzione	Parametri	Precondizioni
start	nome, porta, working dir	peer non attivato
setDataSource	nomeDataBase, nomeDataSource	peer attivato
close	nessuno	peer attivato
connect	porta, host dove connettersi	peer attivato
superConnect	porta, host dove connettersi	superpeer attivato
becomeSP	livello	peer attivato
compileQuery	query	peer attivato
executeQuery	query, canale per i risultati, flag per il caching	peer attivato
subscribe	canale dove inviare, livello da inviare	nessuno
unsubscribe	nessuno	nessuno
split	nessuno	superpeer attivato
clone	nessuno	superpeer attivato
createDataBase	nomeDataBase	peer attivato
createDataSource	nomeDataBase, nomeDataSource, URI	peer attivato
removeDataBase	nomeDataBase	peer attivato
removeDataSource	nomeDataBase, nomeDataSource	peer attivato
getPeersState	nessuno	nessuno
shutDown	nessuno	nessuno
freezePeer	nessuno	peer attivato
defrostPeer	nessuno	peer attivato

**Tabella 2.1:** Comandi inviabili da un Proxy

con l'istruzione *subscribe*;

- l'istruzione *split* serve per avviare il protocollo di splitting di un superpeer;
- l'istruzione *clone* serve per avviare il protocollo di cloning di un superpeer;
- l'istruzione *createDataBase* serve per creare un nuovo database;
- l'istruzione *createDataSource* serve per creare un nuovo datasource;
- l'istruzione *removeDataBase* serve per rimuovere un database precedentemente creato;
- l'istruzione *removeDataSource* serve per rimuovere un datasource precedentemente creato;
- l'istruzione *getPeersState* serve per ricevere come risultato lo stato di un processo XPeer, ovvero dell'entità peer e dell'entità superpeer eventualmente presente;
- l'istruzione *shutDown* serve per chiudere un Proxy;
- l'istruzione *freezePeer* serve per simulare, nell'intero processo XPeer, un blocco del sistema. Dopo questo comando peer e superpeer non risponderanno più agli altri, simulando una situazione di sovraccarico;
- l'istruzione *defrostPeer* serve per far tornare alla normalità il processo XPeer che ha precedentemente ricevuto un'istruzione *freezePeer*.

## 2.4 Architettura dell'Orchestratore

L'Orchestratore è la parte del sistema di orchestrazione che si occupa di amministrare i Proxy che sono stati lanciati sulle altre macchine remote. L'Orchestratore è anche l'interfaccia vera e propria del sistema di orchestrazione che permette agli utenti di creare, modificare e eseguire gli spartiti di test.

L'Orchestratore è composto da:

- una interfaccia grafica;

- un gestore dei Proxy attivati;
- un compilatore di spartiti;
- un esecutore di spartiti.

L'interfaccia grafica è a sua volta composta da:

- un editor di testo per leggere e modificare il codice di uno spartito;
- una console per visualizzare l'output di esecuzione di uno spartito;
- una finestra che visualizza la tabella di tutti i Proxy che hanno eseguito un protocollo di connessione.

Il gestore dei Proxy attivati si occupa del protocollo di connessione dal punto di vista dell'Orchestratore. Per ogni notifica di attivazione che riceve, estrae il canale contenuto dentro il messaggio stesso, lo memorizza in una tabella e, successivamente, completa il protocollo di connessione (figura 2.5).



**Figura 2.5:** *Diagramma di attività del gestore dei Proxy*

Il compilatore, che ai fini di questo tirocinio non è stato realizzato, si occupa di trasformare il codice di uno spartito in codice eseguibile. Nel capitolo seguente mostreremo come fare questa operazione manualmente ma in modo sistematico.

L'esecutore di spartiti esegue il codice dello spartito creando prima l'ambiente necessario alla sua esecuzione, in particolare istanziando i meccanismi di comunicazione e rendendo visibile la tabella dei Proxy che può essere usata durante la scrittura dello spartito stesso. L'esecutore assume che lo spartito sia scritto rispettando il protocollo di comando, ovvero che per ogni comando inviato a un Proxy lo spartito si aspetti una risposta, e viene invocato dall'interfaccia grafica nel momento in cui l'utente decide di eseguire uno spartito.

L'utente che scrive uno spartito, al posto di utilizzare l'indirizzo dei vari Proxy per inviargli comandi, utilizzerà un indice che denota la posizione del Proxy nella tabella. Se uno o più indici non fossero presenti l'esecutore segnalerà un errore prima ancora di eseguire effettivamente lo spartito. In questo modo l'utente che scrive lo spartito non deve preoccuparsi di sapere chi e dove saranno i Proxy che effettivamente riceveranno i comandi da eseguire, ma gli basta sapere che esiste una tabella contenente l'elenco dei Proxy attivi al momento del lancio dello spartito stesso.

# Capitolo 3

## XOL: Un linguaggio di orchestrazione

In questo capitolo introdurremo il concetto di concorrenza e di linguaggio concorrente, spiegando in dettaglio il linguaggio preso di riferimento per la creazione di XOL: il  $\pi$ -calcolo [Mil91]. Quindi descriveremo a fondo il linguaggio, in particolare la sua sintassi e la sua semantica.

### 3.1 I linguaggi concorrenti

La concorrenza è una delle maggiori e affascinanti sfide dell'informatica moderna, sia dal punto di vista teorico che pratico.

Gli scenari concorrenti presentano un insieme di attori (unità di esecuzione) che manipolano delle risorse comuni agendo simultaneamente in maniera, per quanto possibile, indipendente.

Questo approccio porta dei vantaggi in termini di rapidità di esecuzione, e semplicità di programmazione del singolo attore, che si può limitare a svolgere una sola attività tra le tante che porteranno al risultato finale. D'altra parte, progettare un attore in modo da funzionare indipendentemente dalle alterazioni che gli altri attori possono introdurre nell'ambiente presenta notevoli complicazioni.

Distinguiamo quindi tra due famiglie di modelli, che si differenziano nel metodo usato per permettere l'interazione tra i vari attori:

- Memoria Condivisa: gli attori possono utilizzare le stesse “locazioni”, per cui le modifiche fatte da uno possono essere usate dagli altri.
- Scambio di Messaggi: ciascun attore risiede in un ambiente completamente isolato, ma comunica con gli altri attraverso l’invio e la ricezione di messaggi.

A seconda di quale modello si ha in mente servono differenti primitive per la concorrenza, in un caso per modificare la memoria e nell’altro per scambiare messaggi. Questo secondo modello ha portato Robin Milner a creare un linguaggio denominato CCS (Calculus of Communicating System) e il  $\pi$ -calcolo.

### 3.1.1 Il $\pi$ -calcolo

Il  $\pi$ -calcolo è un’algebra di processi capace di descrivere le interazioni di processi concorrenti.

Questo linguaggio si basa sul concetto di *nome*, che sono in quantità infinita, e che indichiamo con  $x, y, \dots \in \mathcal{X}$ . I processi, che indicheremo con  $P, Q, \dots \in \mathcal{P}$ , sono costruiti a partire dai nomi, seguendo la seguente sintassi:

$$P ::= \sum_{i \in I} \pi_i.P_i \mid P|P \mid (vx)P \mid !P$$

dove  $I$  è un insieme finito di indici, e nel caso  $I = \emptyset$  indichiamo la sommatoria con 0. Nella espressione  $\pi.P$  il prefisso  $\pi$  rappresenta la prima azione eseguita da  $\pi.P$ . I prefissi sono di due tipi:

- $x(y).P$ , che corrisponde ad aspettare un nome sul canale che si chiama  $x$ , chiamarlo  $y$  e legare il nome  $y$  in  $P$ ;
- $\bar{x}y.P$ , che significa inviare il nome  $y$  sul canale che si chiama  $x$ .

La semantica informale dei vari costrutti sintattici è la seguente:

- Con  $P_1|P_2$  indichiamo l’operatore di composizione parallela, e significa che  $P_1$  e  $P_2$  sono concorrentemente attivi e possono agire indipendentemente.
- Con  $(vx)P$  indichiamo l’operatore di restrizione che lega il nome  $x$  nel processo  $P$ .

- Con  $!P$  (bang  $P$ ) indichiamo l'operatore di replicazione. Ciò equivale ad eseguire infinite copie di  $P$  in parallelo ( $P|P|\dots$ ) o più semplicemente ad eseguire  $P|!P$ .
- Con  $\sum_{i \in I} \pi_i.P_i$  indichiamo l'operatore di somma guardata dove i vari  $\pi_i$  rappresentano le guardie dei rispettivi processi  $P_i$ . Questo operatore sceglie ed esegue soltanto uno dei vari processi  $\pi_i.P_i$  ignorando gli altri.
- Con  $0$  indichiamo un particolare processo che non fa niente.

## 3.2 La libreria JCSP

JCSP [JCS] è una libreria sviluppata presso l'Università del Kent, che ci permette di scrivere in Java programmi strutturati internamente a processi che comunicano tra loro tramite scambio di messaggi.

Questa libreria ci fornisce:

- un meccanismo per definire processi, in particolare per definire il codice che costituisce il corpo del processo;
- un meccanismo per il lancio in parallelo di un certo numero di processi;
- alcuni meccanismi per realizzare la comunicazione sincrona tra processi.

Sfortunatamente, i meccanismi di comunicazione messi a disposizione da questa libreria possono essere utilizzati esclusivamente su un'unica Java Virtual Machine e non sono progettati per funzionare in un ambiente distribuito.

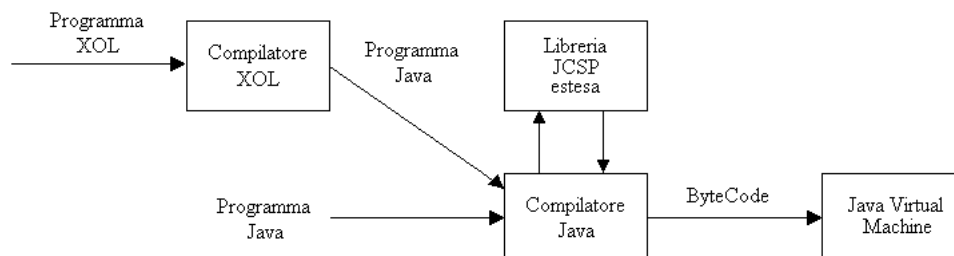
## 3.3 Il linguaggio XOL

XOL significa *XPeer Orchestrating Language*, ed è il linguaggio che vogliamo utilizzare per scrivere gli spartiti da eseguire con il nostro orchestratore. Questo linguaggio nasce prendendo ispirazione dal  $\pi$ -calcolo, aggiungendo strutture importanti, quali gli array, e alcuni costrutti legati a queste strutture.

Gli spartiti scritti con XOL non vengono eseguiti direttamente (non esiste un interprete), ma vengono tradotti in Java con l'aggiunta di una libreria che mette a

disposizione le primitive di comunicazione. Questa libreria è la JCSP, estesa con meccanismi di comunicazione asincroni, utilizzabili in ambiente distribuito.

Una volta tradotti, e ricompilati con il compilatore Java, questi spartiti possono essere eseguiti con una normale Java Virtual Machine (figura 3.1).



**Figura 3.1:** *Esecuzione di un programma XOL*

L'ambiente iniziale in cui lo spartito tradotto viene eseguito non è vuoto ma contiene uno speciale array, denominato *peers*, contenente i canali di tutti i Proxy attivi al momento del lancio dello spartito.

### 3.3.1 Sintassi

La sintassi, che definiamo mediante una grammatica con  $P$  come simbolo distinto, rappresenta l'insieme dei costrutti necessari per la scrittura dei nostri spartiti, ed è così definita:

$$P ::= !c(m) \mid ?c(ide).P \mid native(J) \mid 0$$

$$P ::= new\ ide\ in\ P$$

$$P ::= new\ ide[N]\ in\ P$$

$$P ::= let\ ide = V^+ in\ P$$

$$P ::= let\ ide[] = \{V^+, \dots, V^+\} in\ P$$

$$P ::= process\ ide(W^+) = P in\ P$$

$$P ::= P \mid P \mid P.P \mid P^\$ \mid P^\$n$$

$$\begin{aligned}
P &::= \sum_{i=2}^n G_i.P_i \\
P &::= \left( \sum_{i=2}^n G_i.P_i \right)^* \\
P &::= \text{call } ide(V^+) \\
P &::= \text{forpar } (ide = E; E; E) P \\
G &::= ?c \mid \text{timeout}(N)
\end{aligned}$$

Dove  $J$  rappresenta un qualunque comando Java,  $W^+$  è una tupla di identificatori, i parametri formali del processo parametrico,  $V^+$  rappresenta una tupla di valori qualunque  $E$  è una espressione numerica con identificatori e  $N$  è un qualunque numero naturale.

### 3.3.2 Semantica

Questa semantica ha lo scopo di descrivere sommariamente i costrutti principali di XOL, senza darne una definizione formale mediante regole di inferenza. Inoltre, dopo aver visto alcuni casi d'uso illustreremo le regole di compilazione per costruire un programma Java partendo da un programma XOL.

La semantica informale è così definita:

- Con  $!c(m)$  indichiamo l'invio di un messaggio  $m$  sul canale  $c$ .
- Con  $?c(ide).P$  indichiamo l'attesa di un messaggio dal canale  $c$ . Questo prende il nome  $ide$ . Successivamente si prosegue con  $P$  dove  $ide$  è legato.
- Con  $native(J)$  indichiamo l'inserimento del codice Java "nativo"  $J$  per aggiungere agli spartiti tutta la potenza espressiva di Java.
- Con  $0$  indichiamo l'azione che non fa niente.
- Con  $new\ ide\ in\ P$  indichiamo la creazione di un nuovo canale di nome  $ide$  che può essere utilizzato nel processo  $P$ .
- Con  $new\ ide[N]\ in\ P$  indichiamo la creazione di un array con  $N$  nuovi canali di nome  $ide$  che può essere utilizzato nel processo  $P$ .

- Con  $let\ ide = V^+ in P$  indichiamo la creazione di una costante denominata  $ide$  legata alla tupla di valori  $V^+$ , utilizzabile nel processo  $P$ .
- Con  $let\ ide[] = \{V_1^+, \dots, V_N^+\} in P$  indichiamo la creazione di un array di costanti denominato  $ide$  contenenti le tuple di valori  $V_1^+, \dots, V_N^+$ , utilizzabili nel processo  $P$ .
- Con  $process\ ide(W^+) = P_1 in P_2$  indichiamo la creazione di un processo parametrico denominato  $ide$ , con la tupla di parametri formali  $W^+$ , che denota il processo  $P_1$  dove gli elementi appartenenti a  $W^+$  verranno sostituiti con i parametri attuali. Questo processo parametrico può essere richiamato con la  $call$  soltanto nel processo  $P_2$ .
- Con  $P_1|P_2$  ( $P_1$  parallelo  $P_2$ ) indichiamo l'esecuzione concorrente di due processi  $P_1$  e  $P_2$ .
- Con  $P_1.P_2$  ( $P_1$  sequenziale  $P_2$ ) indichiamo l'operatore di *join*, cioè eseguiamo prima il processo  $P_1$  se e soltanto quando tutti i processi che compongono  $P_1$  saranno terminati eseguiremo il processo  $P_2$ .
- Con  $P^s$  ( $P$  ripetuto) indichiamo l'esecuzione di infiniti processi  $P$ , uno dopo l'altro.
- Con  $P^s N$  ( $P$  ripetuto  $N$  volte) indichiamo lo stesso effetto di  $P'$  ma un numero finito  $N$  di volte.
- Con  $call\ ide(V^+)$  indichiamo l'esecuzione del processo parametrico precedentemente definito come  $ide$ , passando la lista di valori  $V^+$  come parametri attuali.
- Con  $forpar\ (ide = E_1; E_2; E_3) P$  indichiamo la stessa semantica del costrutto *for* di un linguaggio imperativo ma le diverse istanze di  $P$  vengono eseguiti in modo parallelo e non in modo sequenziale.
- Con  $?c$  indichiamo una guardia, che è vera quando sul canale  $c$  c'è un messaggio da leggere, e falsa quando questo messaggio non è presente.

- Con  $timeout(N)$  indichiamo un'altra guardia, che è vera dopo che sono passati  $N$  millisecondi dall'attivazione, e falsa se questo periodo non è ancora trascorso.
- Con  $\sum_{i=2}^n G_i.P_i$  indichiamo un alternatore “con guardia”, in grado di scegliere il processo  $P_i$  se la rispettiva guardia  $G_i$  è vera.
- Con  $(\sum_{i=2}^n G_i.P_i)^*$  indichiamo la replicazione di un alternatore deterministico. Ciò equivale a eseguire infinite copie del processo  $\sum_{i=2}^n G_i.P_i$  in parallelo.

### 3.3.3 Casi d'uso

Vediamo adesso degli esempi di spartiti (casi d'uso) per verificare l'effettiva utilità dei costrutti del nostro linguaggio precedentemente definiti. Questi spartiti avranno lo scopo di testare le varie funzionalità del sistema XPeer.

Possiamo raggruppare questi casi d'uso in tre classi principali:

- i test relativi alla *costruzione di una rete*;
- i test relativi alle *misurazioni* delle operazioni di XPeer, quali la compilazione o l'esecuzione di interrogazioni;
- i test relativi alla *gestione dei fallimenti* di peer o superpeer vicini.

#### Costruzione della rete

Con questo esempio vogliamo creare uno spartito che costruisce una rete di tre peer, eleggerne uno a superpeer, e connettere tutti e tre i peer al superpeer appena creato, costruendo così una configurazione iniziale del sistema XPeer.

Per prima cosa vorremmo definire alcune procedure utili per comunicare con i vari peer; essi, oltre ad aspettarsi messaggi ben precisi, rispondono sempre ad ogni messaggio che viene loro inviato con un avviso di terminazione o messaggio di errore. Per farlo, ci rendiamo subito conto che abbiamo bisogno di un costrutto (come quello che abbiamo definito) in grado di definire processi parametrici in grado di comportarsi come le procedure di un linguaggio imperativo.

Le procedure utili possono essere così definite:

```

process Send(canale, comando) =
  !canale(comando)
in
process ReceiveAndPrint(canale) =
  ?canale(messaggio) . native(System.out.println(messaggio.toString()))
in
process SendCommand(peer, comando, canaleDiRisposta)
  call Send(peer, comando) |
  call ReceiveAndPrint(canaleDiRisposta)
in P

```

Iniziamo con il creare i tre peer assumendo di avere l'array predefinito denominato `peers` che contiene i canali necessari a inviare i comandi ai vari peer.

La creazione può quindi essere fatta con il seguente codice:

```

new canaleDiRispostaStart[3] in
process SendStartCommand(n, params) =
  call SendCommand(peers[n], ("start", params,
    canaleDiRispostaStart[n]), canaleDiRispostaStart[n])
in
  let params[] = {
    "Peer1 2001 .", "Peer2 2002 .", "Peer3 2003 ."
  } in
  forpar (i=0;i<3;i++)
    call SendStartCommand(i, params[i])

```

A questo punto creiamo la struttura della rete desiderata, iniziando ad eleggere un peer a superPeer di livello 1:

```

new canaleDiRispostaBecomeSP in
  call SendCommand(peers[0], ("becomeSP", "1",
    canaleDiRispostaBecomeSP), canaleDiRispostaBecomeSP)

```

e facciamo connettere tutti gli altri peer al nostro superPeer appena creato

```

new canaleDiRispostaConnect[3] in
process SendConnectCommand(n) =
  call SendCommand(peers[n], ("connect", "2001 "+peers[0],
    canaleDiRispostaConnect[n]), canaleDiRispostaConnect[n])
in
  forpar (i=0;i<3;i++)
    call SendConnectCommand(i)

```

Mettendo insieme i pezzi di codice sopra descritti otteniamo lo spartito completo, che ci permette di costruire la rete con un superpeer e tre peer. Lo spartito integrale è il seguente:

```

process Send(canale, comando) =
  !canale(comando)
in
process ReceiveAndPrint(canale) =
  ?canale(messaggio) . native(System.out.println(messaggio.toString()))
in
process SendCommand(peer, comando, canaleDiRisposta)
  call Send(peer, comando) |
  call ReceiveAndPrint(canaleDiRisposta)
in
(
(
new canaleDiRispostaStart[3] in
process SendStartCommand(n, params) =
  call SendCommand(peers[n], ("start", params,
    canaleDiRispostaStart[n]), canaleDiRispostaStart[n])
in
let params[] = {
  "Peer1 2001 .", "Peer2 2002 .", "Peer3 2003 ."
} in
forpar (i=0;i<3;i++)
  call SendStartCommand(i, params[i])
) .
(
new canaleDiRispostaBecomeSP in
  call SendCommand(peers[0], ("becomeSP", "1",
    canaleDiRispostaBecomeSP), canaleDiRispostaBecomeSP)
) .
(
new canaleDiRispostaConnect[3] in
process SendConnectCommand(n) =
  call SendCommand(peers[n], ("connect", "2001 "+peers[0],
    canaleDiRispostaConnect[n]), canaleDiRispostaConnect[n])
in
forpar (i=0;i<3;i++)
  call SendConnectCommand(i)
)
)

```

## Misurazione

In questo esempio, dopo aver costruito la rete con lo spartito precedente, vogliamo eseguire una interrogazione e misurarne i tempi di compilazione e di esecuzione.

Iniziamo preparando un processo parametrico che aspetta un messaggio da un canale e, successivamente, lo appende in fondo a un file. Potremmo anche scrivere una procedura più precisa, ma assumiamo di aver già scritto un processo

parametrico, chiamato *append*, che prende un messaggio e lo inserisce alla fine di un file. Questo processo parametrico *append* è scritto utilizzando codice Java.

```
process ReceiveAndAppend(canale, file) =
  ?canale(messaggio) . call append(messaggio, file)
in
```

Dobbiamo inoltre ordinare al peer di inviarci le misurazioni, comunicandogli il canale sul quale vogliamo che tali messaggi vengano inviati. Vogliamo inoltre che questi messaggi siano scritti in un file per poterli elaborare successivamente (ad esempio con strumenti come Matlab).

```
new canaleDiRispostaSubscribe[3] in
new canaleMeasure[3] in
process SendSubscribeCommand(n) =
  call SendCommand(peers[n], ("subscribe", "measure" +
    canaleMeasure[n], canaleDiRispostaSubscribe[n]),
    canaleDiRispostaSubscribe[n])
in
let xmlfile[] = {
  "./measure/mes1.dat", "./measure/mes2.dat", "./measure/mes3.dat"
} in
forpar (i=0;i<3;i++) (
  call SendSubscribeCommand(i) |
  (call ReceiveAndAppend(canaleMeasure[i], fileMeasure[i]))$
)
```

Definiamo adesso un processo parametrico per inviare una query da far compilare ed eseguire a un peer.

Da notare che viene anche inviato, come parametro del comando, il canale sul quale inviare i risultati e un flag che indica se vogliamo ricevere i risultati mano a mano che vengono generati (*false*) oppure se li vogliamo ricevere in un unico messaggio al termine dell'esecuzione (*true*).

In parallelo all'invio di questo comando, mettiamo un processo che si occupa della ricezione dei risultati sul canale precedentemente creato; se non viene ricevuto alcun messaggio entro cinque minuti assumiamo che nessun risultato sia stato generato dall'esecuzione, e lo segnaliamo.

```
new canaleDiRispostaExecute[3] in
new canaleRisultatiEsecuzione[3] in
process SendExecuteCommand(n, query, file) =
  (call SendCommand(peers[n], ("execute", query +
    canaleRisultatiEsecuzione[n]+"true",
    canaleDiRispostaExecute[n]), canaleDiRispostaExecute[n])) |
```

```

(?canaleRisultatiEsecuzione .
  (call ReceiveAndAppend(canaleRisultatiEsecuzione[n], file)) +
  timeout(5000) .
  native(System.err.println("Attenzione, nessuna risposta")))
in
let files[] = {
  "./execute1.dat", "./execute2.dat", "./execute3.dat"
} in
let queries[] = {
  "for \b in \ds1/bib return \b",
  "for \b in \ds1//book return \b",
  "for \b in \ds1/bib/book let \saa := \b//author return \b",
} in
forpar (i=0;i<3;i++)
  call SendExecuteCommand(i, queries[i], files[i])

```

Mettendo tutto insieme, e aggiungendo i soliti processi di utilità precedentemente definiti, otteniamo lo spartito:

```

new canaleDiRispostaSubscribe[3] in
new canaleMeasure[3] in
new canaleDiRispostaExecute[3] in
new canaleRisultatiEsecuzione[3] in
process SendSubscribeCommand(n) =
  call SendCommand(peers[n], ("subscribe", "measure" +
    canaleMeasure[n], canaleDiRispostaSubscribe[n]),
    canaleDiRispostaSubscribe[n])
in
process SendExecuteCommand(n, query, file) =
  (call SendCommand(peers[n], ("execute", query +
    canaleRisultatiEsecuzione[n]+"true",
    canaleDiRispostaExecute[n]), canaleDiRispostaExecute[n])) |
  (?canaleRisultatiEsecuzione .
    (call ReceiveAndAppend(canaleRisultatiEsecuzione[n], file)) +
    timeout(5000) .
    native(System.err.println("Attenzione, nessuna risposta")))
in
let xmlfile[] = {
  "./measure/mes1.dat", "./measure/mes2.dat", "./measure/mes3.dat"
} in
let files[] = {
  "./execute1.dat", "./execute2.dat", "./execute3.dat"
} in
let queries[] = {
  "for \b in \ds1/bib return \b",
  "for \b in \ds1//book return \b",
  "for \b in \ds1/bib/book let \saa := \b//author return \b",
} in

```

```

(
  forpar (i=0;i<3;i++)
    call SendMeasureCommand(i)
) . (
  (
    forpar (i=0;i<3;i++)
      (call SendExecuteCommand(i, queries[i], files[i]))$
    ) | (
      forpar (i=0;i<3;i++)
        (call ReceiveAndAppend(canaleMeasure[i], fileMeasure[i]))$
      )
    )
  )
)

```

### Fallimenti di sistema

In questo ultimo esempio, ancora una volta dopo aver costruito la rete con lo spartito di costruzione, vogliamo vedere come si comporta un'esecuzione da parte di un peer simulando il sovraccarico del superpeer.

Iniziamo con l'invio del messaggio di blocco al superpeer, che nello spartito per costruire la rete era quello di indice zero.

```

new canaleDiRispostaFreeze in
  call SendCommand(peers[0], ("freezePeer", "",
    canaleDiRispostaFreeze), canaleDiRispostaFreeze)

```

A questo punto è sufficiente eseguire uno spartito di Misurazione, come lo spartito precedentemente mostrato per eseguire una interrogazione, e raccogliere i dati relativi alle misurazioni in tempi e risultati delle interrogazioni stesse. In particolare, in questo esempio, per ogni peer ci aspettiamo di non ricevere nessun dato relativo agli altri peer, avendo bloccato l'unico superpeer della nostra rete.

In uno scenario più complicato, dove sono presenti numerosi superpeer, è interessante vedere quanto tempo impiega il sistema ad autoamministrarsi per tornare a eseguire correttamente e in tempo accettabile le interrogazioni proposte.

### 3.3.4 Regole di compilazione

Lo spartito scritto in Java prevede che il codice di ogni processo parametrico sia scritto in una classe che implementa l'interfaccia *CSProcess* della libreria JCSP e quindi che abbia disponibile un particolare metodo denominato *run*.

Ai fini dell'esecuzione uno spartito viene visto come un particolare processo parametrico che prende come parametro l'array di Proxy attivi denominato *peers*. Quindi il compilatore deve creare una classe con lo stesso nome del file che contiene lo spartito e che implementa l'interfaccia *CSProcess*. A questa classe il compilatore aggiunge altre informazioni che sono:

- il massimo numero di Proxy utilizzati nello spartito (*nPeer*);
- gli *import* necessari per la corretta esecuzione del codice Java.

Il codice prodotto dal compilatore si presenta sempre in questa forma:

```
import jcsp.lang.*;
...

public class NomeFileSpartito implements CSProcess {
    public static int nPeer = N;
    private CSProcess process;

    public NomeFileSpartito(final SocketChannelOutput [] peers) {
        this.process = CodiceCompilato;
    }

    public void run() {
        this.process.run();
    }
}
```

Il codice compilato viene prodotto attraverso l'applicazione di una funzione  $\mathcal{P}$ , che dato un costrutto della grammatica restituisce il corrispondente codice Java. Questo codice è sempre un oggetto della classe *CSProcess* dotato di un metodo *run* che viene definito utilizzando le classi anonime di Java e viene invocato al momento della sua esecuzione.

La funzione  $\mathcal{P}$  è così definita:

- Per il costrutto  $!c(m)$  creiamo un processo che nel metodo *run* invoca il metodo *write* del canale *c*, che a sua volta invia il messaggio *m* sul canale. Se *c* non è un canale o non è stato dichiarato il compilatore Java segnalerà

un errore di compilazione:

```

 $\mathcal{P}[\![c(m)]\!] =$ 
    new CSProcess(){
        public void run(){
            c.write(m);
        }
    }

```

- Per il costrutto  $?c(ide).P$  creiamo un processo che nel metodo run crea un oggetto che si chiama *ide* al quale assegnamo il messaggio ricevuto dal canale *c* tramite l'invocazione del metodo *receive*. Successivamente invochiamo il metodo run del processo risultante dalla compilazione di *P* dove può essere utilizzato l'oggetto *ide* in quanto dichiarato:

```

 $\mathcal{P}[\![?c(ide).P]\!] =$ 
    new CSProcess(){
        public void run(){
            Object ide = c.receive();
             $\mathcal{P}[\![P]\!]$ .run();
        }
    }

```

- Per il costrutto *native(J)* creiamo un processo che nel metodo run ha esattamente il codice Java *J*:

```

 $\mathcal{P}[\![native(J)]\!] =$ 
    new CSProcess(){
        public void run(){
            J;
        }
    }

```

```

    }
}

```

- Per il costrutto 0 creiamo un processo che nel metodo run non esegue nessuna operazione:

```

 $\mathcal{P}[[0]] =$ 
    new CSProcess(){
        public void run(){
        }
    }
}

```

- Per il costrutto *new ide in P* creiamo un processo che nel metodo run dichiara un canale di nome *ide*, lo inizializza invocandone il costruttore e invoca il metodo run del processo risultante dalla compilazione di *P*:

```

 $\mathcal{P}[[new\ ide\ in\ P]] =$ 
    new CSProcess(){
        public void run(){
            final SocketChannel ide =
                new SocketChannel(cm, "ide");
             $\mathcal{P}[[P]].run()$ ;
        }
    }
}

```

- Per il costrutto *new ide[N] in P* creiamo un processo che nel metodo run dichiara e inizializza un array di canali di lunghezza *N* e invoca il metodo

run del processo risultante dalla compilazione di  $P$ :

```

 $\mathcal{P}[\text{new } ide[N] \text{ in } P] =$ 
  new CProcess(){
    public void run(){
      final SocketChannel[] ide =
        SocketChannel.create(cm, "ide", N);
       $\mathcal{P}[P].run()$ ;
    }
  }

```

- Per il costrutto  $\text{let } ide = V^+ \text{ in } P$  creiamo un processo che nel metodo run dichiara e inizializza un oggetto di nome  $ide$  che rappresenta la tupla di valori  $V^+$ :

```

 $\mathcal{P}[\text{let } ide = V^+ \text{ in } P] =$ 
  new CProcess(){
    public void run(){
      final Object ide =  $V^+$ ;
       $\mathcal{P}[P].run()$ ;
    }
  }

```

- Per il costrutto  $\text{let } ide[] = \{V_1^+, \dots, V_N^+\} \text{ in } P$  creiamo un processo che nel metodo run dichiara e inizializza un array di nome  $ide$  che contiene le tuple di valori  $V_i^+$ :

```

 $\mathcal{P}[\text{let } ide[] = \{V_1^+, \dots, V_N^+\} \text{ in } P] =$ 
  new CProcess(){
    public void run(){

```

```

        final Object[] ide = {
            V1+, ..., VN+
        }
        P[[P]].run();
    }
}

```

- Per il costrutto  $process\ ide(W^+) = P_1\ in\ P_2$  creiamo un processo che nel metodo run dichiara una nuova classe *ide* che implementa CSProcess e che contiene il risultato della compilazione del processo  $P_1$ . Successivamente invoca il metodo run del processo risultante dalla compilazione di  $P_2$ :

```

P[[process ide(W+) = P1 in P2]] =
    new CSProcess(){
        public void run(){
            class ide implements CSProcess {
                private CSProcess process;
                public ide(W+) {
                    this.process = P[[P1]];
                }
                public void run(){
                    this.process.run();
                }
            };
            P[[P2]].run();
        }
    }
}

```

- Per il costrutto  $P_1|P_2$  creiamo un processo *parallelo* della libreria JCSP

passandogli come parametro un array di processi contenente il risultato della compilazione del processo  $P_1$  e del processo  $P_2$ :

$$\mathcal{P}[[P_1|P_2]] =$$

```
new Parallel(new CSProcess[] { $\mathcal{P}[[P_1]]$ ,  $\mathcal{P}[[P_2]]$ })
```

- Per il costrutto  $P_1.P_2$  creiamo un processo *sequenziale* della libreria JCSP passandogli come parametro un array di processi contenente il risultato della compilazione del processo  $P_1$  e del processo  $P_2$ :

$$\mathcal{P}[[P_1.P_2]] =$$

```
new Sequence(new CSProcess[] { $\mathcal{P}[[P_1]]$ ,  $\mathcal{P}[[P_2]]$ })
```

- Per il costrutto  $P^{\$}$  creiamo un processo *ripetuto* della libreria JCSP passandogli come parametro il risultato della compilazione del processo  $P$ :

$$\mathcal{P}[[P^{\$}]] =$$

```
new Repeated( $\mathcal{P}[[P]]$ )
```

- Per il costrutto  $P^{\$}N$  creiamo un processo *ripetuto* della libreria JCSP passandogli come parametro il risultato della compilazione del processo  $P$  e il numero di volte  $N$  che deve ripetere il processo:

$$\mathcal{P}[[P^{\$}N]] =$$

```
new Repeated( $\mathcal{P}[[P]]$ ,  $N$ )
```

- Per il costrutto *call ide*( $V^+$ ) creiamo un processo della classe *ide* passandogli i parametri attuali  $V^+$ :

$$\mathcal{P}[[\textit{call ide}(V^+)]] =$$

```
new ide( $V^+$ )
```

- Per il costrutto  $forpar (ide = E_1; E_2; E_3) P$  creiamo un processo che nel metodo `run` costruisce un processo *parallelo* tramite un *for* che utilizza l'espressioni  $E_1$ ,  $E_2$  e  $E_3$  e il processo  $P$ . Successivamente invoca il metodo `run` di questo processo *parallelo* per eseguire concorrentemente tutti i processi:

```

 $\mathcal{P}[\![forpar (ide = E_1; E_2; E_3) P]\!] =$ 
    new CSProcess(){
        private Parallel process
            = new Parallel();
        public void run(){
            for (int ide= $E_1; E_2; E_3$ ) {
                this.process.addProcess(
                     $\mathcal{P}[\![P]\!]$ 
                );
            }
            this.process.run();
        }
    }

```

- Per il costrutto  $\sum_{i=2}^n G_i.P_i$  creiamo un processo *alternativo* della libreria JCSP passandogli come parametro un array di guardie contenente il risultato della compilazione dei vari  $G_i$  e un array di processi contenente il risultato della compilazione dei vari  $P_i$ :

```

 $\mathcal{P}[\![\sum_{i=2}^n G_i.P_i]\!] =$ 
    new Alternator(
        new Guard[] { $\mathcal{G}[\![G_1]\!]$ , ...,  $\mathcal{G}[\![G_n]\!]$ },
        new CSProcess[] { $\mathcal{P}[\![P_1]\!]$ , ...,  $\mathcal{P}[\![P_n]\!]$ })

```

- Per il costrutto  $(\sum_{i=2}^n G_i.P_i)^*$  creiamo un processo *replicato* della libreria JCSP contenente un processo *alternativo* creato come nel costrutto precedente:

$$\mathcal{P}\left[\left(\sum_{i=2}^n G_i.P_i\right)^*\right] =$$

```

new Replicated(
    new Alternator(new Guard[] {G[G1], ..., G[Gn]},
        new CSProcess[] {P[P1], ..., P[Pn]})
)

```

Nella funzione  $\mathcal{P}$  utilizziamo la funzione  $\mathcal{G}$ , necessaria per compilare le guardie dell'alternatore deterministico, così definita:

- Per il costrutto  $?c$  creiamo una guardia rappresentata dal canale stesso, la guardia risulterà pronta quando un messaggio è presente sul canale:

$$\mathcal{G}[?c] =$$

$$c$$

- Per il costrutto  $timeout(N)$  creiamo un *timer* della libreria JCSP che sarà pronto dopo  $N$  millisecondi dalla sua attivazione:

$$\mathcal{G}[timeout(N)] =$$

```

new CSPTimer(N)

```

Una volta applicata la funzione  $\mathcal{P}$  lo spartito è pronto per essere compilato con un normale compilatore Java e successivamente eseguito dall'orchestratore.

# Capitolo 4

## Aspetti implementativi

In questo capitolo parleremo degli aspetti più rilevanti dal punto di vista dell'implementazione del sistema descrivendone le scelte principali, le funzionalità secondarie e le configurazioni richieste per il suo corretto funzionamento.

### 4.1 La libreria JCSP

Come abbiamo detto nei capitoli precedenti la libreria che utilizziamo per gli spartiti scritti in Java è la JCSP. Questa libreria presenta due problemi fondamentali:

- i canali funzionano esclusivamente se si trovano tutti su una singola Java Virtual Machine, quindi non possiamo utilizzarli nel nostro ambiente distribuito;
- alcune operazioni che abbiamo introdotto non sono presenti nella libreria ovvero: la replicazione e la ripetizione.

La mancanza di questi due aspetti ha reso necessario l'estensione di questa libreria per darci la possibilità di eseguire correttamente sia nostri spartiti, che l'intero sistema di orchestrazione.

#### 4.1.1 I canali

Un canale è caratterizzato da tre attributi: il nome del canale, il nome dell'host su cui si trova il canale e la porta adibita allo scambio dei messaggi.

I canali implementati utilizzano un gestore dei canali presente nella classe *ChannelManager* che si occupa di memorizzare i dati comuni relativi ai canali gestiti (hostname e porta) e di mettere a disposizione delle primitive ad alto livello per inviare e ricevere messaggi. Questo gestore è suddiviso in tre livelli:

- **Gestore dei messaggi:** astrae la gestione dei socket TCP;
- **Gestore delle code:** che introduce l'astrazione delle code per inviare e ricevere messaggi;
- **Gestore dei canali:** che si occupa di astrarre il concetto di canale utilizzando le code e memorizzando gli aspetti comuni di tutti i canali che gestisce.

I canali di output sono presenti nella classe *SocketChannelOutput* e implementano l'interfaccia *ChannelOutput* che contiene un metodo per la scrittura di un oggetto sul canale. Essi possono essere inviati come messaggi su altri canali tramite una serializzazione del canale stesso e la sua ricostruzione una volta arrivato a destinazione.

I canali di input/output sono presenti nella classe *SocketChannel* e hanno a disposizione sia il metodo per scrivere un oggetto sul canale sia il metodo per leggere un oggetto da un canale. Questi canali sono quelli che vengono creati tramite il comando *new* del linguaggio presentato precedentemente, e anche questi possono essere inviati tramite la serializzazione del canale stesso. A differenza dei canali di output, questa seconda tipologia di canale non può essere ricostruita a destinazione perché da un lato non è possibile serializzare il gestore dei canali e dall'altro perché vogliamo che l'unico ricevente del canale sia colui che lo ha creato. Una volta arrivato a destinazione il canale deve essere ricostruito come canale di output.

## 4.2 Implementazione del Proxy

Durante l'implementazione del Proxy l'attenzione è stata catalizzata sulla possibilità di questa parte del sistema a una semplice estendibilità con ulteriori operazioni da svolgere. In particolare è presente una lista di comandi definita nella

classe *Command*, l'esecutorthread (descritto nel paragrafo 2.3) controlla se ogni comando ricevuto appartiene a questa lista tramite una serie di condizionali e va ad eseguire il metodo relativo all'operazione del comando ricevuto.

Per estendere un Proxy con nuove funzionalità è sufficiente aggiungere un comando alla lista, aggiungere il controllo e creare un metodo opportuno per l'operazione che si vuole svolgere.

### 4.2.1 Configurazione del Proxy

Per eseguire un Proxy, è necessario invocare il metodo *ProxyMonitor.main* passando come parametro rispettivamente: la porta sul quale ricevere i comandi, il nome del canale del Proxy, l'hostname dell'orchestratore, la porta dell'orchestratore e il nome del canale dell'orchestratore. Se una o più di queste informazioni non sono presenti viene utilizzato un valore predefinito.

La cartella di lavoro del Proxy deve essere una qualunque cartella dentro la cartella run del progetto. In questa cartella vengono creati tutti i file usati dal sistema XPeer comprensivi del file di log.

La serializzazione dei comandi è stata fatta usando la codifica dei caratteri di Windows che risulta diversa da quella di altri sistemi operativi. Per rendere il Proxy portabile è sufficiente aggiungere come argomento alla Virtual Machine la stringa *-Dfile.encoding=cp1252*. In questo modo la Virtual Machine di Java utilizza la codifica di Windows anche su altri sistemi operativi.

Alcune funzionalità del sistema XPeer richiedono la presenza di una variabile d'ambiente di nome *path* che va impostata prima del lancio di un Proxy. Il valore da assegnare a questa variabile è *\${env\_var:path};lib/win*.

L'analisi degli aspetti relativi alla gestione della memoria è stata effettuata utilizzando un *Profiler*, ovvero uno strumento in grado di analizzare l'esecuzione di un programma fornendo informazioni relative alle prestazioni e alla gestione delle risorse. Prima di utilizzare questo strumento è necessario aggiungere come parametro alla Virtual Machine la stringa *-Dcom.sun.management.jmxremote*. Successivamente è sufficiente eseguire il programma *jconsole*, presente nelle librerie di Java, e tramite una apposita interfaccia selezionare i Proxy che si vuol monitorare.

### 4.3 Implementazione dell'Orchestratore

La base di partenza per l'implementazione dell'orchestratore sul quale sono state aggiunte tutte le funzionalità descritte nel capitolo 2.4 è MiniPad: un semplice editor per il linguaggio Java [Gin00a, Gin00b] che risolve tutti gli aspetti inerenti alla gestione del documento (creazione, caricamento, salvataggio, modifica) e alla sua compilazione. Per quanto riguarda l'esecuzione non è sufficiente lanciare direttamente uno spartito; esso non è dotato di un metodo *main* e necessita di una fase iniziale di elaborazione.

Le operazioni di compilazione e esecuzione si svolgono creando un processo separato che richiama rispettivamente il compilatore e l'esecutore Java della *Sun*. Per la compilazione non è fondamentale creare un nuovo processo ma è stata adottata questa soluzione per garantire una sostituzione più semplice qualora sia implementato anche il compilatore di XOL. Per quanto riguarda l'esecuzione la soluzione di creare un processo separato è necessaria per evitare che il codice eseguito in uno spartito vada a modificare le strutture dati presenti nella memoria dell'orchestratore e viceversa.

Il meccanismo utilizzato in Java per creare processi separati è attraverso l'invocazione del metodo *Runtime.exec()* che dà la possibilità di eseguire programmi esterni e funzioni specifiche di un sistema operativo. Con l'uso di questo meccanismo è possibile perdere la portabilità dell'applicazione; nel nostro caso l'orchestratore necessita del compilatore e dell'esecutore Java della *Sun* e non di una funzione specifica del sistema operativo quindi il sistema resta portabile per quelle piattaforme dove questi due strumenti sono presenti.

Il processo di esecuzione di uno spartito avviene in tre passi:

- al primo passo l'orchestratore lancia un nuovo processo che esegue l'esecutore passando come argomento il nome della classe che contiene lo spartito compilato e la lista dei Proxy attivi poi si mette in attesa;
- al secondo passo l'esecutore costruisce un gestore dei canali e crea tanti canali di output quanti sono gli elementi della lista dei Proxy, carica la classe dello spartito tramite la *reflection* e esegue lo spartito;

- al terzo passo lo spartito termina e notifica all'orchestratore l'eventuale presenza di errori di esecuzione.

L'utente ha a disposizione la possibilità di bloccare il processo di esecuzione in qualunque momento tramite un apposito bottone dell'interfaccia grafica.

### 4.3.1 L'interfaccia grafica

L'interfaccia grafica è suddivisa in due pannelli che forniscono diverse interazioni con l'orchestratore. Il primo pannello rappresenta l'editor degli spartiti che dà la possibilità di modificare lo spartito stesso e visualizzare sia il canale di output che il canale di errore durante l'esecuzione. Il secondo pannello visualizza la lista di Proxy attivati che si sono connessi all'orchestratore ordinata rispetto all'utilizzo nella prossima esecuzione di uno spartito. L'ordinamento può essere modificato tramite un semplice drag and drop all'interno della tabella.

Dal menu dell'interfaccia grafica è possibile eseguire alcuni spartiti che si rivelano utili in numerose occasioni e che per questo sono stati inseriti nel menu *process* come spartiti predefiniti. Gli spartiti eseguibili sono i seguenti:

- *Show Tree*: che mostra graficamente la struttura della rete ricostruita dopo aver interrogato tutti i Proxy;
- *Show Peer Table*: che mostra una tabella contenente le informazioni sullo stato dei nodi contenuti nei Proxy;
- *Kill All Proxies*: che invia un comando di terminazione a tutti i Proxy attualmente attivi.

Sempre dal menu dell'interfaccia grafica è possibile accedere a una lista di proprietà modificabili dall'utente. La lista di queste proprietà è descritta nella tabella 4.1.

### 4.3.2 Configurazione dell'Orchestratore

Per eseguire l'orchestratore v'è invocato il metodo *XOrch.main* senza passare nessun parametro.

Nome	Significato
ProcessLoader	Il nome della classe dell'esecutore di spartiti
Log4jJar	La cartella contenente la libreria di log
StoreLoc	La cartella contenente gli spartiti compilati
XPeerClass	La cartella contenente le classi compilate del sistema XPeer
JavaVMArgs	Gli argomenti da passare alla Virtual Machine durante l'esecuzione di uno spartito
LocalHostName	Il nome dell'host della macchina
Port	La porta dove l'orchestratore aspetta la connessione dei Proxy
XOrchClass	La cartella contenente le classi compilate dell'orchestratore
FileChooseDir	La cartella proposta quando si sceglie di salvare/caricare un file
ImagesDir	La cartella contenente le immagini visualizzate per L'orchestratore
ClassPath	Ulteriori classpath da aggiungere durante la compilazione/esecuzione di spartiti
LoaderPort	La porta dove l'esecutore crea il gestore dei canali

**Tabella 4.1:** *Proprietà modificabili nell'orchestratore*

La cartella di lavoro dell'Orchestratore è la cartella `/run/xOrch` dentro la cartella del progetto. In questa cartella vengono create tutte le classi compilate e i file di log relativi all'orchestratore e agli spartiti eseguiti.

Le configurazioni da impostare per la Virtual Machine sono le stesse del Proxy, in particolare l'utilizzo della codifica dei caratteri di Windows e del Profiler.

# Capitolo 5

## Conclusioni

In questa relazione sono stati presentati i principali aspetti progettuali e implementativi che hanno riguardato il lavoro svolto durante il tirocinio; questo tirocinio si colloca all'interno del progetto di ricerca XPeer del gruppo di Basi di Dati del Dipartimento di Informatica dell'Università di Pisa.

Durante il tirocinio ho progettato il sistema di orchestrazione che costruisce in un ambiente distribuito una rete del sistema XPeer al fine di scoprire errori latenti, compiere misurazioni sulle prestazioni del sistema o creare una rete iniziale del sistema stesso. Il sistema di orchestrazione risparmia l'utente dal dover eseguire direttamente per tutti i test effettuati tutti i comandi su tutte le singole istanze dei nodi grazie agli spartiti che una volta scritti possono essere eseguiti molte volte.

Il sistema implementato offre tutte le funzionalità descritte tranne il compilatore per il linguaggio XOL. La creazione di questo compilatore rappresenta uno sviluppo futuro per utilizzare il linguaggio da me creato non solo come linguaggio di specifica ma come linguaggio vero e proprio con il quale si scrivono gli spartiti.

### **Esperienze acquisite**

L'esperienza formativa acquisita è relativa alla progettazione, realizzazione e verifica di una piattaforma per la verifica e misurazione di applicazioni distribuite.

Durante questo tirocinio ho avuto modo di imparare molto sui linguaggi di programmazione sia dal punto di vista teorico che pratico, scoprendo varie diffi-

coltà che possono celarsi anche dietro la creazione di un linguaggio apparentemente semplice. Inoltre, aver avuto la possibilità di lavorare con un gruppo di persone esperte e disponibili al fine di risolvere i problemi che sono emersi durante il periodo di tirocinio ha reso questa esperienza utile e piacevole.

# Bibliografia

- [Bat05] Giovanni Battaglia. XPeer: robustezza rispetto ai fallimenti, 2005. Relazione di tirocinio.
- [BL95] Gérard Boudol and Cosimo Laneve.  $\lambda$ -Calculus, Multiplicities and the  $\pi$ -Calculus, 1995.
- [Cec02] Alessia Ceccato. Analisi statica dello Spi-calcolo, 2002. Tesi di laurea.
- [CL06] Samuele Carpineti and Cosimo Laneve. A basic contract language for web services, 2006.
- [CLM05] Samuele Carpineti, Cosimo Laneve, and Paolo Milazzo. BoPi - a distributed machine for experimenting Web Services technologies, 2005.
- [FF07] Donato Ferrante and Michele Freschi. Completamento Compilazione ed Esecuzione Query in XPeer, 2007. Relazione di tirocinio.
- [Gin00a] Andrea Gini. MiniPad Come progettare e implementare un editor Java con compilatore in linea (Parte I). *Mokabyte*, 2000. <http://www.mokabyte.it/2000/04/minipad1.htm>.
- [Gin00b] Andrea Gini. MiniPad Come progettare e implementare un editor Java con compilatore in linea (Parte II). *Mokabyte*, 2000. <http://www.mokabyte.it/2000/05/minipad2.htm>.
- [Gio05] Nicola Gioia. Un sistema Peer-to-Peer per l'interrogazione distribuita di dati XML, 2005. Tesi di laurea.

- 
- [GLW03] Philippa Gardner, Cosimo Laneve, and Lucian Wischik. Linear Forwarders, 2003.
- [JCS] JCSP. <http://www.cs.kent.ac.uk/projects/ofa/jcsp/>.
- [Mil91] Robin Milner. The Polyadic  $\pi$ -Calculus: a Tutorial, 1991.
- [PT06] Giovanni Pardini and Paolo Tomei. La gestione delle interrogazioni in XPeer, 2006. Tesi di laurea.
- [SMGC04] Carlo Sartiani, Paolo Manghi, Giorgio Ghelli, and Giovanni Conforti. XPeer: A Self-organizing XML P2P Database System, 2004.
- [SMGC07] Carlo Sartiani, Paolo Manghi, Giorgio Ghelli, and Giovanni Conforti. Scalable Query Dissemination in XPeer, 2007.