Università di Pisa

Facoltà di Scienze Matematiche Fisiche e Naturali

Laurea Specialistica in Tecnologie Informatiche

Anno Accademico 2009/2010

Tesi

Tunneling SSL in Muskel : implementazione e valutazione delle prestazioni

Candidato

Daniele Grotti

Relatore

Prof. Marco Danelutto

INDICE

INTRODUZIONE……..……...pag 4

Capitolo 1 (MUSKEL)

1.1 AMBIENTI A SKELETON IN JAVA PER LA PROGRAMMAZIONE PARALLELA .....pag 9

1.2 IMPLEMENTAZIONE AMBIENTE MUSKEL ……..pag 14

1.3 GRANA E SICUREZZA IN MUSKEL 2.0…….……...…….pag 18

Capitolo 2 (MUSKEL & SSL)

2.1 SICUREZZA NEI PROGRAMMI A SKELETON……...……pag 21

2.2 REMOTE METHOD INVOCATION……...……pag 22

2.3 SECURE SOCKET LAYER…...……...……pag 30

2.4 RMI & SSL IN JAVA ……..…….……..…….pag 39

2.5 PROGETTO LOGICO……...…….…….pag 45

Capitolo 3 (IMPLEMENTAZIONE)

3.1 INTRODUZIONE ……..……pag 49

3.2 IMPLEMENTAZIONE DELLA RICERCA DEGLI HOST NEL MANAGER…….……pag 51

3.3 IMPLEMENTAZIONE "BEST EFFORT" …….....pag 53

3.4 IMPLEMENTAZIONE DELLA GESTIONE TRA MANAGER E CONTROL-THREAD DELLA REMOTE-EXCEPTION…….…….pag 54

3.5 IMPLEMENTAZIONE SERVER RMI CON SSL……..…….pag 59

3.6 IMPLEMENTAZIONE CLIENT RMI CON SSL……..…….pag 62

3.7 GENERAZIONE TRUSTSTORE E KEYSTORE CON KEYTOOL ……..…….pag 64

Capitolo 4 (VALUTAZIONI)

4.1 VALUTAZIONI……..…….pag 69

4.2 IMPATTO DELLA RETE …….……...……..……...pag 77

Capitolo 5 (CONCLUSIONI)

5.1 CONCLUSIONI…….…….……..pag 78

5.2 FUTURE WORK……..……..…….……...…….…….…….…..pag 79

BIBLIOGRAFIA…….……...…….……pag 80

APPENDICE (CODICE)…….....pag 82

INTRODUZIONE

Lo sviluppo di programmi paralleli efficienti è un compito arduo da raggiungere; infatti oltre a codificare tutti i dettagli dell'algoritmo, il programmatore deve anche prendersi cura dei dettagli riguardanti la gestione del parallelismo: inizializzazione delle attività concorrenti, mapping e scheduling, gestione della comunicazione e sincronizzazione, allocazione dei dati ecc.

Queste operazioni sono normalmente a carico del programmatore applicativo ed è difficile che non portino a commettere errori. Lo sforzo richiesto al programmatore cambia, in base al linguaggio/ambiente scelto per lo sviluppo dell'applicazione parallela.

L'ambiente di programmazione Java offre diverse possibilità che possono essere naturalmente sfruttate per utilizzare la rete e per il calcolo distribuito: JVM e bytecode, multi-threading, Remote Method Invocation, Socket e Sicurezza oltre alle più recenti JINI, Java Spaces, Servlets, ecc. [1].

Molte applicazioni parallele e distribuite sono state sviluppate utilizzando queste feature [2].

Per rendere Java un ambiente adatto alla programmazione parallela, sono stati fatti negli anni, molti sforzi.

In particolare sono stati portati avanti progetti per offrire maggiori "features" da usare nello sviluppo di applicazioni Java parallele su una serie di architetture parallele differenti [3,4].

Alcune "features" vengono fornite come "extensions" dei linguaggi base o come librerie di classi. Nel primo caso, sono stati sviluppati e implementati compilatori ad hoc e/o ambienti a runtime.

Lithium è un esempio di libreria Java per la programmazione parallela basata su "algorithmical skeleton".

Gli skeleton sono stati originariamente concepiti da Cole [9] e vengono utilizzati da differenti gruppi di ricerca per modellare la programmazione strutturata parallela in ambienti ad alte prestazioni [10-12].

Dopo essere stati introdotti da Cole, gli algorithmical skeleton hanno portato allo sviluppo di una serie di sistemi a skeleton.

Gli ambienti per la programmazione parallela sfruttano il concetto di skeleton in modi differenti: creando librerie, nuovi linguaggi, una cooperazione di linguaggi e pattern.

P3L [26] e ASSIST [27,28] sono esempi di come questi framework di programmazione possono essere implementati mettendo a disposizione i linguaggi per la programmazione parallela a skeleton. Sono entrambi linguaggi di programmazione progettati e implementati all'Università di Pisa nel '91e nel 2000 rispettivamente.

Altri esempi di librerie che permettono la programmazione parallela a skeleton sono eSkel [29,30], Skipper [31], Muesli [32], e muskel [33].

I primi due sono implementati in C e C++, girano sopra MPI e sono stati progettati rispettivamente da Cole e Kuchen.

Skipper è implementato invece in Ocalm, gira direttamente sopra lo strato di rete TCP/IP, e usa la stessa implementazione macro data flow (MDF) utilizzata da muskel.

Muskel è una libreria derivata da Lithium [7,8,34] scritta completamente in Java che sfrutta RMI per l'implementazione della comunicazione con gli host remoti.

ASSIST e muskel, sono stati sviluppati entrambi con l'obiettivo di essere utilizzati su reti eterogenee di workstation e grid.

In questi ambienti, sono stati presi in considerazione diversi problemi di implementazione: da una parte, sono state prese in considerazione i problemi derivanti da firewall e alte latenze della rete, e dall'altra quelli relativi la sicurezza.

In particolare, i problemi riguardanti la sicurezza crescono quando i programmi a skeleton vengono eseguiti su architetture distribuite i cui nodi remoti e cluster sono interconnessi da reti pubbliche (quindi non dedicate).

In questi casi, codice (memorizzato sui nodi remoti per l'esecuzione) e dati (dati di inputi e risultati della computazione) viaggiano da e verso i nodi remoti attraverso collegamenti potenzialmente soggetti ad attacchi di vario genere.

Dati e codice attraversando il collegamento insicuro possono essere facilmente "snooped and spoofed" da utenti che non hanno i diritti per farlo. E' necessario quindi assicurare il flusso nelle connessioni in reti non sicure.

Questo naturalmente ha un costo che viene pagato sia nella fase di invio e ricezione tra le macchine (tempo speso nella cifratura e decifratura di codici e dati), che in termini di banda della rete (le connessioni cifrate possono richiedere più comunicazioni e una diversa grandezza dei messaggi scambiati rispetto allo stesso tipo di comunicazione, ma in chiaro).

Gli skeleton sono un comune modello di astrazione, utilizzabile nella definizione di pattern paralleli.

Come dicevamo, gli skeleton vengono messi a disposizione al programmatore in forma di "language constructs" [10,11] o come librerie [13-15], e possono anche essere annidati per creare applicazioni parallele complesse [14,16].

Per scrivere applicazioni parallele utilizzando ambienti paralleli basati su skeleton, il programmatore deve, di solito, seguire alcuni passi distinti:

- deve dichiarare espressamente la strutturare parallela dell'applicazione utilizzando skeleton opportunamente annidati;
- deve quindi scrivere la parte di codice sequenziale per l'applicazione specifica;
- infine deve semplicemente compilare e linkare il codice risultante per ottenere l'eseguibile parallelo.

Lithium permette al programmatore di utilizzare pienamente un ambiente Java per la programmazione parallela basata su skeleton. La libreria supporta gli skeleton più comuni, inclusi pipeline, task farm nonchè skeleton, data parallel e iterativi.

Utilizzando Lithium, il programmatore può inizializzare una applicazione parallela istanziando gli skeleton ed eventualmente annidandoli, e può poi fornire dei task per la computazione (input data), richiedendo l'esecuzione parallela del programma risultante tra una serie di workstation interconnesse. Alla fine otterrà i risultati della sua computazione.

Ad esempio, il programmatore può esprimere l'applicazione parallela come un pipeline a stadi sequenziali o che esprimono un certo parallelismo.

Quando l'applicazione è stata realizzata e testata, il programmatore può cercare di affinare il programma migliorandone le prestazioni. Durante questo processo di

ottimizzazione egli può lavorare o sulla struttura dello skeleton o sui parametri, in modo che i colli di bottiglia nelle performance vengano eliminate o ridotte [16].

L'implementazione di Lithium sfrutta gli RMI di Java per distribuire la computazione fra i vari nodi di una certa architettura. Le "Java reflection features" sono inoltre sfruttate per rendere semplici le API di Lithium.

Ultimo, ma non meno importante, è la struttura pulita e orientata agli oggetti (OO) del codice di Lithium, che permette anche l'esecuzione sequenziali di programmi paralleli su una sola macchina. Quest'ultima è una feature molto utile soprattutto in fase di debugging.

Lithium rappresenta un affinamento consistente e uno sviluppo di un vecchio lavoro riportato in [17] e le regole di ottimizzazione implementate in Lithium estendono quelle discusse in [18,19].

Muskel è basato su *macro data flow* (***MDF***), e per l'implementazione delle applicazioni parallele sfrutta sia il *multi-threading*, localmente alla macchina dell'utente, che la remote method invocation (***RMI***), per l'esecuzione in parallelo dell'applicazione.

L'obiettivo di questa tesi è quello introdurre nuove feature in Muskel (versione 2.0), in particolare, una forma di comunicazione sicura con i nodi remoti distribuiti, tra i quali nodi, considerati "sicuri", e nodi considerati "insicuri", per via del tipo di collegamento implementato attraverso reti pubbliche.

Dopo una introduzione sul funzionamento di Muskel, verranno discussi i dettagli di alcune delle nuove feature introdotte in questa nuova versione, ad esempio, quelle che utilizzano la tecnologia SSL (Secure Socket Layer).

Seguirà una breve spiegazione su RMI, Secure Sockets Layer e sui costi della sicurezza nei sistemi a skeleton, che evidenzierà come questo sia stato implementato all'interno dell'interprete distribuito Muskel, discutendone le implicazioni sulle performance.

Verranno valutati, nel caso della feature relative all'utilizzo di SSL, i costi relativi alla sicurezza.

Prima si considererà un utilizzo delle tecniche di comunicazione sicura su tutte le macchine a disposizione e poi solo su un sottoinsieme di macchine considerate non sicure, tra tutte quelle a disposizione.

Verrà valutato quindi il “tempo di completamento” e il “tempo di setup” per diverse configurazioni con diverse grane di calcolo.

I risultati saranno valutati tenendo presente gli obiettivi di funzionalità, scalabilità ed efficienza oltre naturalmente agli obiettivi di sicurezza.

Nel Capitolo 1 verranno introdotti gli ambienti a skeleton e in particolare l’ambiente Java Muskel per la programmazione parallela.

Nel Capitolo 2 dopo una introduzione sulla sicurezza negli ambienti a skeleton, si introdurrà la Remote Method Invocation (RMI), utilizzata in Muskel per le comunicazioni con i worker remoti, e il Secure Socket Layer (SSL) per la parte relativa alla sicurezza che verrà introdotta nella nuova versione di muskel.

Nel Capito 3 verranno descritti alcuni dettagli delle feature implementate nella nuova versione di muskel.

Nel Capitolo 4 si valuteranno i risultati ottenuti con alcuni esperimenti fatti eseguendo una applicazione parallela “sintetica” con la nuova versione di muskel, per testare principalmente la parte relativa alla sicurezza.

Verranno quindi tratte le conclusioni nel Capitolo 5.

Capitolo 1

MUSKEL

1.1

AMBIENTI A SKELETON IN JAVA PER LA PROGRAMMAZIONE PARALLELA

Lithium e Muskel sono ambienti Java strutturati per la programmazione parallela basati su skeleton.

Con gli algorithmic skeleton si è raggiunto un buon compromesso tra la potenza espressiva e l'efficienza nel campo della programmazione parallela/distribuita.

Un *algorithmic skeleton* non è altro che un modello parametrico di implementazione del parallelismo, che può essere personalizzato dal programmatore utilizzando opportuni parametri, in base alle necessità che una certa applicazione richiede.

Normalmente è permesso usare skeleton in modo annidato, di modo che utilizzando la composizione di skeleton semplici, i programmatori possano realizzare applicazioni arbitrariamente complesse.

Tipici esempi di skeleton sono il Farm (che modella computazioni parallele di task indipendenti su stream), Pipeline (che modella le computazioni organizzate in stadi), Map, Reduce e Prefix (che modellano le classiche computazioni data parallel apply-to-all e reduce) nonchè diversi skeleton iterativi (che modellano diversi schema di loop).

Una caratteristica dei modelli di programmazione basati su skeleton è quella di non consentire al programmatore di scrivere applicazioni al livello "assembly", interagendo direttamente con l'ambiente di esecuzione distribuito tramite comunicazione, con primitive di accesso alla memoria condivisa e/o tramite schedulazione e mappatura esplicite del codice.

Piuttosto, con questi modelli è possibile sfruttare il parallelismo parametrico, incapsulando e astraendo, mediante gli skeleton, tutti i dettagli relativi all'utilizzo del parallelismo.

Ad esempio, per implementare una semplice applicazione parallela tutti i dati in ingresso sono passati utilizzando uno stream di input o un file, ed il programma

semplicemente istanzia copie di una composizione di skeleton fornite dall'utente per produrre o istanziare tutto il codice necessario a elaborare ogni singolo dato dell'input stream.

Il sistema, il compilatore o le librerie a "runtime", si occuperanno poi di allocare e gestire in modo adeguato le risorse disponibili, di schedulare i task in maniera opportuna tra le risorse utilizzate e di distribuire gli "input task" e ottenere i relativi risultati in base al mapping utilizzato.

Nei sistemi tradizionali, invece, il programmatore avrebbe dovuto inserire esplicitamente tutto il codice necessario per la distribuzione e la schedulazione dei processi verso le risorse disponibili nonchè per il trasferimento dei dati di input e output tra gli elementi coinvolti nella computazione.

Il costo di questo approccio ad alto livello nell'utilizzo dei programmi paralleli è pagato in termini di libertà di programmazione; al programmatore infatti non è permesso, di solito, fornire modelli arbitrari di esplicitazione del parallelismo.

Egli può usare solo quei modelli forniti dal sistema, cioè i modelli forniti dagli skeleton messi a disposizione e dalle loro composizioni.

Tali modelli, peraltro, includono normalmente tutti quei modelli riutilizzabili che servono per realizzare implementazioni distribuite efficienti delle applicazioni parallele più comuni.

Tutto questo avviene principalmente per togliere la possibilità ai programmatori di scrivere codice inefficiente di persè e che potenzialmente potrebbe compromettere l'efficienza dell'implementazione fornita per gli skeleton messi a disposizione.

In Muskel, per implementare gli skeleton vengono utilizzati i macro-data flow (***MDF***).

Un modo abbastanza intuitivo di elaborazione su stream e modalità di parallelizzazione è quello che fa riferimento appunto al modello computazionale detto ***data-flow***.

Partendo da una computazione sequenziale, questa viene trasformata in un ***grafo data-flow*** applicando le condizioni di Bernstein che permettono di determinare un ordinamento parziale tra le operazioni della computazione stessa: le relazioni di precedenza tra operazioni dipendono esclusivamente dalla disponibilità dei dati secondo uno schema di tipo *produttore-consumatore*.

Nel grafo data-flow i *nodi* corrispondono, dunque, ad operazioni e gli *archi* a canali mediante i quali trasmettere valori dei dati tra operazioni.

Un programma a skeleton viene compilato per ottenere un ***grafo MDF***.

Un grafo MDF viene definito creando alcune istruzioni ***MDF*** (*macro data flow instruction*) e connettendole all'interno di uno grafo MDF.

Un grafo MDF può essere definito per estendere l'insieme degli skeleton.

In Muskel, la prima istruzione MDF del grafo deve avere un unico ***token*** di input e l'ultima istruzione MDF del grafo deve avere un unico ***token*** di output.

Un esempio di codice muskel per la creazione di un grafo MDF, che imposta la prima e l'ultima istruzione del grafo attraverso i metodi ***setInputInstruction*** e ***setOutputInstruction*** e inserisce le istruzioni nel grafo richiamando il metodo ***addIstruction*** è riportato di seguito:

```
MdfGraph graph = new MdfGraph();

graph.setInputInstruction(i1);
graph.addInstruction(i1);
graph.addInstruction(i2);
graph.addInstruction(i3);
graph.addInstruction(i4);
graph.setOutputInstruction(i4);
```

Le istruzioni all'interno del grafo rappresentano esecuzioni delle parti sequenziali del codice.

Le istruzioni MDF diventano ***fireable*** soltanto quando sono presenti tutti i ***token*** necessari perché l'istruzione venga calcolata.

Il flusso dei dati (gli ***archi*** del ***grafo MDF***) deriva direttamente dalla struttura annidata dello ***skeleton*** [20,21].

Il programma a skeleton viene eseguito caricando un processo server su ogni macchina disponibile e un taskpool manager sulla macchina locale dell'utente.

I grafi data flow possono essere usati in Muskel nelle situazioni dove potrebbero essere usati gli skeleton classici, permettendo di integrare nello stesso tempo entrambe le forme di sfruttamento del parallelismo all'interno di uno stesso programma .

La definizione da parte dell'utente dei grafi data flow offre la possibilità di programmare nuovi modelli per lo sfruttamento del parallelismo.

L'utente può utilizzare una serie di skeleton primitivi che possono essere nidificati per il calcolo in parallelo di dati in ingresso che appaiono su stream di input, come pipeline e farm.

L'implementazione degli skeleton, come dicevamo, è fatta compilando i programmi a skeleton forniti all'utente, in grafi "*macro data flow*" (MDF), dove ogni task che deve essere calcolato viene utilizzato come ***input token*** di un'istanza del grafo compilato a partire dal programma a skeleton fornito dal programmatore.

Le istruzioni ***fireable*** disponibili nel grafo, vengono quindi schedulate per l'esecuzione nei nodi remoti dove è implementato un ***interprete data flow***, e i risultati ottenuti, sono utilizzati per generare nuove istruzioni o come risultati dell'esecuzione del programma.

I server remoti sono in grado di eseguire ciascuna delle istruzioni ***fireable*** contenute nel grafo.

Il taskpool manager, in esecuzione sulla macchina dell'utente che lancia il programma, si occupa di tenere un ***MDF repository***, dove vengono memorizzate tutte le istruzioni MDF relative al grafo che viene gestito, e di fornire i server remoti con le istruzioni ***MDF fireable*** che devono essere eseguite.

Logicamente, ogni ***input task*** disponibile viene utilizzato come un ***token*** di input per una istanza di un grafo MDF che viene memorizzato nel taskpool.

Quando l'istruzione diventa ***fireable***, perchè tutti i token sono disponibili, può quindi essere inviata ad uno dei server remoti per essere eseguita.

Il server remoto calcola l'istruzione MDF e invia il risultato ad una o più istruzioni MDF nel taskpool, o all'output stream.

Alcune istruzioni MDF, intanto, possono diventare ***fireable*** e il processo viene ripetuto fino a quando esiste almeno una istruzione MDF***fireable*** all'interno del taskpool.

L'ultima istruzione MDF viene quindi trovata e rimossa dal taskpool.

Tutto questo processo è completamente trasparente all'utente.

Un esempio del processo di compilazione ed esecuzione di un programma a skeleton basato su ***data flow*** è riportato nella *Figura 1.1*.

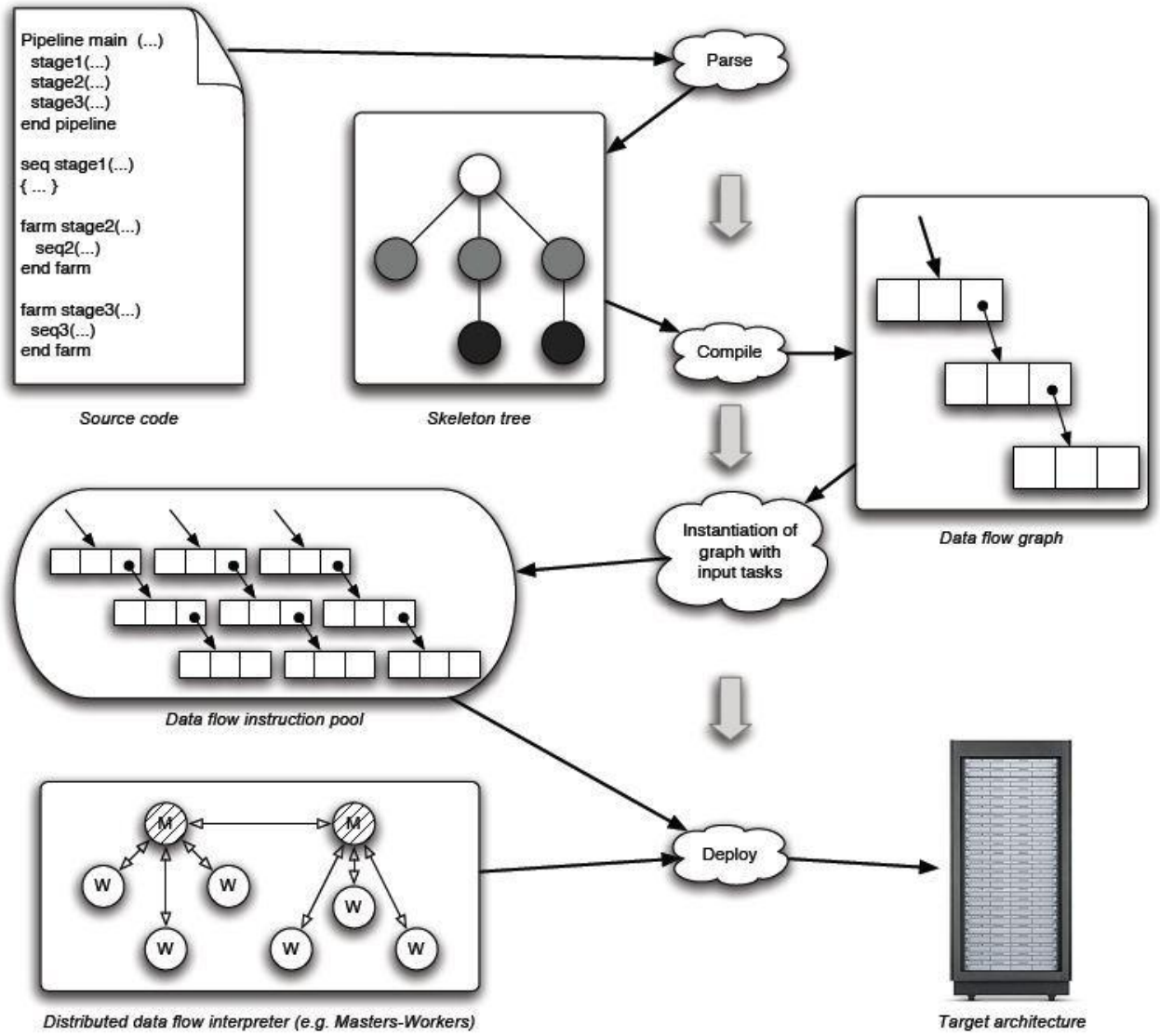


Fig 1.1 Skeleton program execution according to the data flow approach.

In passato l'implementazione del supporto RMI si è dimostrata poco efficiente [22], ma l'implementazione attualmente presente nel JDK permette di raggiungere una buona efficienza e ottime prestazioni nell'esecuzione di programmi basati su skeleton.

Il server remoto RMI deve essere creato a mano, (attraverso ad esempio un comando ssh) o tramite appositi script in Perl forniti insieme all'ambiente Muskel.

Nel taskpool manager, locale, viene lanciato un thread per ogni nodo remoto utilizzato per il programma a skeleton. Tale thread ottiene un riferimento locale ad un oggetto "RMI Server", remoto.

Verrà adesso discussa l'implementazione dell'ambiente muskel basato su macro data flow.

1.2

IMPLEMENTAZIONE AMBIENTE MUSKEL

L'ambiente Muskel, per la programmazione parallela basata su skeleton, usa un oggetto della classe ***Manager*** per gestire le computazioni.

L'oggetto ***manager*** prende come input: un programma a skeleton, un input manager, un output manager e un contratto di performance, che serve per impostare, ad esempio, il grado di parallelismo (come nell'*Esempio 1* pag 17*)*.

Il ***manager*** cerca poi di reclutare il numero di interpreti remoti richiesto dall'utente con il contratto di performance.

La ricerca degli elementi di calcolo disponibili viene fatta attraverso un semplice protocollo basato sul multicast "peer-2-peer", reclutando il numero di interpreti remoti richiesti, con il contratto di performance.

Il ***manager*** per la ricerca degli interpreti remoti attraverso scambio di messaggi multicast usa la classe ***DiscoveryService***.

La classe ***DiscoveryService*** viene usata per mantenere una lista di interpreti disponibili, scoperti attraverso lo scambio di messaggi multicast con le macchine che stanno eseguendo un oggetto RMI della classe ***RemoteInterpreter***.

I ***RemoteInterpreter*** in particolare useranno un thread della classe ***PresenceThread*** per rispondere ai messaggi multicast.

Il ***DiscoveryService*** utilizza un thread della classe ***DiscoveryThread*** per realizzare effettivamente la fase di ricerca degli interpreti remoti.

Il ***DiscoveryThread*** controlla continuamente la presenza di nuovi worker, e ogni volta che ne trova uno lo comunica al ***DiscoveryService*** che si occuperà di aggiungerlo alla lista.

Viene quindi istanziato un oggetto della classe ***ControlThread*** per ogni interprete che si riesce a trovare.

L'oggetto ***controlthread*** effettua un loop nel quale recupera una istruzione ***fireable*** disponibile nel MDF graph ***repository*** (***MdfiPool***), da consegnare ad un ***RemoteInterpreter.***

Si ottiene quindi il risultato della computazione remota e eventualmente, o si consegna il risultato come ***token*** nel grafo MDF o, nel caso che questo sia un risultato finale, si consegna all'***OutputManager***.

Il ***Manager*** si occupa di tenere una copia aggiornata del grafo nell' ***MDF graph repository*** e per ogni task prodotto dall'***InputManager*** inserisce il ***task*** come ***token*** nella corretta istruzione MDF iniziale, all'interno di una nuova istanza del grafo completo una volta per tutte a partire dal programma utente.

Nel caso si verifichi un problema con un interprete remoto (fallimento di un nodo remoto o problemi di rete), il ***controlthread*** informa il ***manager*** e termina.

Quindi il ***manager*** cerca di recuperare la situazione reclutando un altro ***RemoteInterpreter*** e reinserendo l'istruzione non eseguita nel ***repository*** di istruzioni MDF.

Il repository è rappresentato da un oggetto ***pool*** della classe ***MdfiPool***, che viene creato nel ***manager*** e viene condiviso con i ***controlthread***..

Sull'oggetto ***pool*** è possibile richiamare i metodi:

- ***insert (Mdfi istr)*** per inserire una istruzione MDFi nel ***pool***
- ***insert (Mdf graph)*** per inserire un grafo MDF nel ***pool***
- ***extract (IstructionTag instrtag)*** per estrarre l'istruzione Mdfi corrispondente ad un dato ***instructioTag***

- ***extract (InstructionTag instrtag, GraphId graph)*** per estrarre l'istruzione Mdfi corrispondente ad un dato ***InstructionTag*** e ad un dato ***GraphId***.

Il ***manager*** quando crea un ***controlthread*** per ogni ***RemoteInterpreter***, trovato utilizzando il ***DiscoveryService***, gli passa tra i parametri di input anche un riferimento all'oggetto ***pool***.

Il ***controlthread*** potrà quindi utilizzarlo, per inserire o estrarre i ***task***, richiamando i metodi opportuni su tale oggetto condiviso.

Gli interpreti remoti vengono lanciati sui nodi remoti, ad esempio tramite comandi ssh o tramite script forniti con l'ambiente muskel.

Essi sono infatti ***remote objects*** Java, che girano come processi standalone o come Java ***Activable objects*** e sono specializzati nell'eseguire istruzioni MDF gestite dai ***controlthread,*** istanziati dal ***manager***.

Il ***controlthread*** manda agli interpreti remoti la versione serializzata della classe ***Compute*** appena prima di iniziare la consegna delle istruzioni ***Mdfi fireable*** disponibili.

Il programmatore usando Muskel può esprimere il parallelismo della computazione usando esclusivamente le classi *Farm* e *Pipeline*, visto che questi sono gli unici due skeleton messi a disposizione.

Il funzionamento di muskel è rappresentato nella *Figura 1.2*.

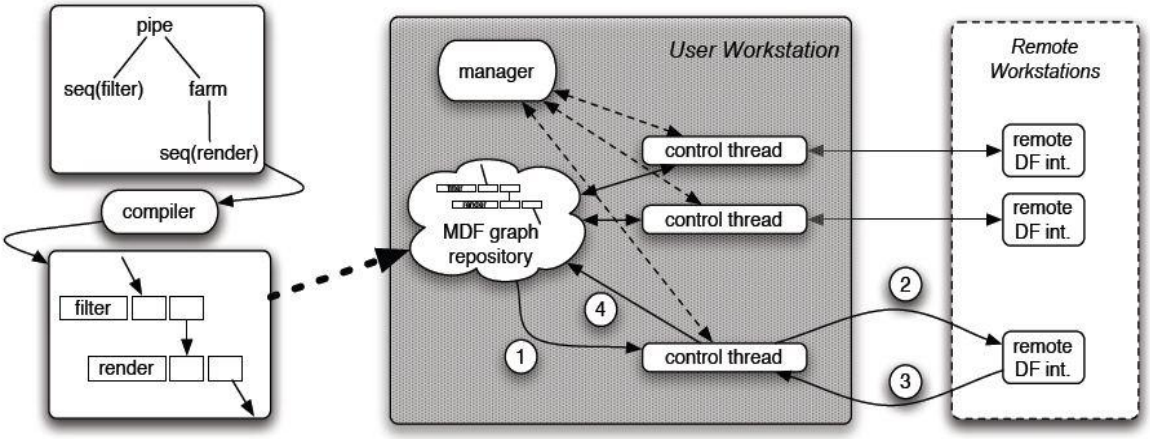


Figura 1.2

L'Esempio 1, che segue, fa vedere il codice necessario per programmare un pipeline a due stadi, con il secondo stadio implementato con un farm, per il filtraggio di immagini mediche e le loro successive renderizzazione.

```
package muskel.*;
public class SampleCode {
public static void main(String[] args) {

//first of all define the program to be computed
Compute filter = new Filter();//first stage is filtering
Compute render = new Render();//second stage is rendering
Compute farm = new Farm(render); //as rendering is heavier than filtering,
le'ts farm it out in the pipe
Compute main = new Pipeline(filter,farm); //this is the program we eventually
compute

//then arrange to provide input "task" stream and some way to manage
resultsfirst stage is filtering
InputManager inM = //provide input image stream (implement hasNext/next
abstraction to get images)
      new ImageStreamInputManager("sample_mage.da");
OutputManager outSM = //store results (provide deliver method to handle each
result computed)
      new FileOutputManager("sample_result.dat");

//now arrange to declare a manager: it will completely take care of the parallel
computation
Manager mng = new Manager(main, inM, outM); //declare the manager
mng.setContract(new ParDegree(Integer.parseInt(args[0]))); //ask to use args[0]
remote PEs

//now ask to compute the program
mng.compute();//start the computation and wait for termination
      }
}
```

Esempio 1

Nell'esempio si assume che esistano due classi Java che processano immagini mediche provenienti da un certo tipo di scanner (PET, CAT, MNR) per filtrare (class ***`Filter`***) le immagini e quindi visualizzarle (class ***`Render`***) una volta filtrate.

Lo stream di immagini da processare viene letto da un file sfruttando adeguatamente i metodi ***`boolean hasNext()`*** e ***`Object next()`*** implementati nella classe ***`InputManager`*** definita dall'utente.

Le immagini elaborate, saranno eventualmente memorizzate in un altro file, invocando il metodo ***`void deliver(Object r)`*** implementato nella classe ***`OutputManager`*** definita dall'utente.

I metodi *hasNext*, *next* e *deliver* vengono utilizzati dall'interprete Muskel e vengono forniti dall'utente che sa come sono fatti i dati di input e come vengono generati i risultati.

Inoltre, l'utente sa che la fase di rendering è sensibilmente più lunga di quella di filtering, e quindi decide di eseguire il rendering in parallelo, scrivendo il secondo stadio del pipeline come un farm.

1.3

GRANA E SICUREZZA IN MUSKEL 2.0

La versione 2.0 di Muskel è stata testata su diverse configurazioni di workstation inclusi cluster dedicati, reti locali che utilizzano workstation di produzione, reti su scala geografica che gestivano lo stesso tipo di macchine in due zone separate da firewall ecc.

In tutti questi casi, è stata raggiunta una quasi perfetta scalabilità nell'esecuzione di programmi a grana medio/alta.

E' stato dimostrato che le configurazioni su rete locale presentano buona scalabilità quando le computazioni hanno una ***`grana`*** (rapporto tra il tempo speso nella computazione di una istruzione MDF e il tempo speso nella consegna dell' input token e nel recupero dell'output token) intorno ai 100.

Le reti su scala geografica, invece richiedono una computazione con una grana sensibilmente maggiore (1 o 2 ordini di grandezza maggiori rispetto alla rete locale).

Quando *Muskel* viene eseguito su nodi remoti e cluster interconnessi tramite reti non dedicate o pubbliche, alcune accortezze andrebbero prese se i dati oggetto della computazione fossero dati sensibili.

In questi casi, infatti codice e dati viaggiano da e verso il nodo remoto, attraverso un collegamento potenzialmente pericoloso.

Codice e dati che attraversano il collegamento non sicuro possono essere facilmente oggetto di intercettazioni o "spoofing" da parte di persone che non hanno i diritti per gestire la computazione parallela.

Le conseguenze di questo tipo di intrusione possono portare il nodo remoto a eseguire una computazione che non avrebbe dovuto calcolare, rendendo di fatto così, il risultato globale non corretto.

Proprio per questo motivo in questa tesi verranno introdotte una serie di nuove feature per muskel, tra le quali l'utilizzo di forme di sicurezza nella comunicazione con gli host remoti, cercando di garantire autenticazione, integrità e cifratura dei dati.

Uno degli scopi sarà quindi rendere sicuro il flusso di dati su reti non protette, ma tale obiettivo dovrà essere bilanciato con il costo, che viene pagato sia, nella fase di invio che di ricezione tra le macchine, nonchè in termini di banda della rete.

Si dovrà valutare l'incidenza dell'implementazione della sicurezza sul costo totale della computazione, sia con un diverso numero di nodi, che per una grana del calcolo più o meno grande.

Sarà quindi valutata la scalabilità e l'efficienza con e senza l'implementazione della sicurezza o con una implementazione parziale, considerando alcuni nodi come sicuri, configurati opportunamente, e lasciando sugli altri le impostazioni iniziali.

Sarà compito del programmatore configurare opportunamente le macchine prima dell'avvio della computazione, settandole come sicure ad esempio, direttamente da linea di comando al momento dell'esecuzione del ***`RemoteInterpreter`***.

Un esempio di scalabilità ed efficienza dei ***RemoteInterpreter*** nella versione di ***muskel 2.0*** è rappresentato nella Fig 1.3.

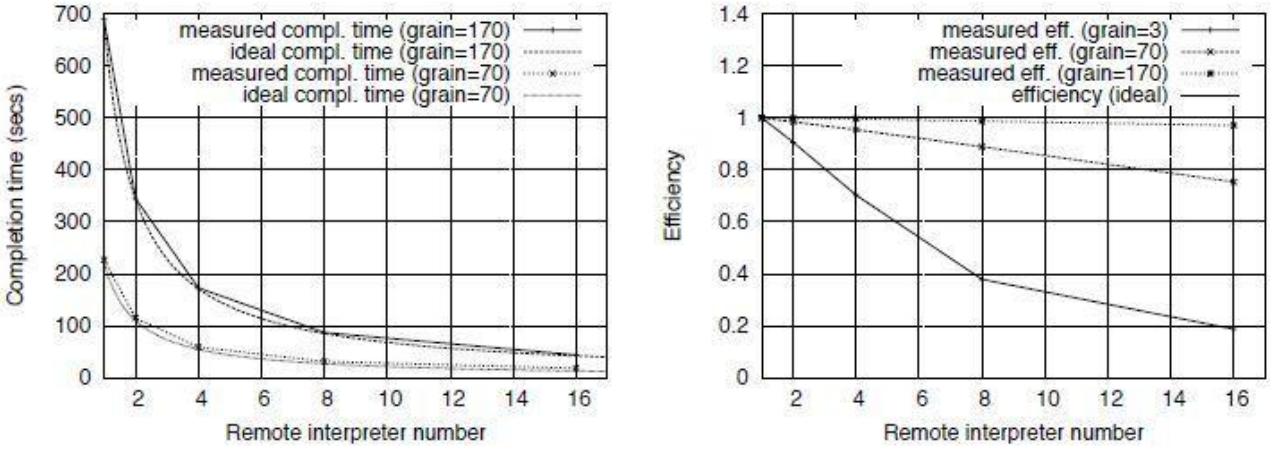


Fig 1.3 Scalability of muskel prototype and effect of computation grain

Capitolo 2

MUSKEL & SSL

2.1

SICUREZZA NEI PROGRAMMI A SKELETON

Quando eseguiamo programmi paralleli usando nodi che sono interconnessi mediante una rete pubblica, ci sono dei rischi dovuti al fatto che le comunicazioni possono essere intercettate e che i dati importanti possono finire nelle mani di utenti non autorizzati.

Inoltre, i dati posso essere intercettati e sostituiti con altri dati sbagliati o ingannevoli, sfruttando le tecniche di "spoofing", determinando conseguentemente dei calcoli errati.

Un esempio di programma che soffre di questo particolare problema è SETI@home [24], un esperimento scientifico che utilizza i computer connessi ad internet per la ricerca di Intelligenze Extratterresti.

Nelle prime versioni di questo programma per il calcolo distribuito, infatti, i dati non venivano cifrati e gli utenti non autenticati, di modo che molti utenti malintenzionati hanno introdotto dati corrotti nel database generale eseguendo di fatto computazioni errate.

Inoltre, molti singoli individui ed aziende hanno fatto cambiamenti non ufficiali alla parte distribuita del software per provare ad ottenere risultati più velocemente, ma questo ha compromesso l'integrità di tutti i risultati.

Prendiamo in considerazione quello che avviene in ***Muskel***: il codice serializzato viene inviato ad un ***RemoteInterpreter*** che viene usato per calcolare sul nodo remoto l'istruzione MDF relativa al codice utente di quello ***skeleton***.

Se il codice viene cambiato, il nodo remoto può venire usato per calcolare cose che non doveva calcolare.

Quindi è fondamentale, per eliminare problemi di codice e dati che l'accesso al ***RemoteInterpreter*** sia autenticato in modo sicuro, e che il codice stesso sia cifrato prima di essere inviato al nodo remoto.

L’autenticazione e la cifratura del codice può essere implementata usando l’estensione JSSE [23] di Java, inclusa nel JDK standard a partire dalla versione 1.4.

Pertanto in questa tesi, oltre ad altre feature, verranno introdotti in ***Muskel*** meccanismi per prevedere l’autenticazione, la privacy e l’integrità nelle comunicazioni tra i ***controlthread*** in esecuzione sulla macchina utente e l’***interprete remoto*** in esecuzione sulla macchina remota.

In particolare, verrà realizzata una versione di Muskel (estendendo muskel originale) che sfrutta la libreria SSL per le comunicazione riguardanti i nodi remoti considerati non sicuri.

SSL fornisce esattamente autenticazione, usando chiavi asimmetriche, privacy usando chiavi di sessione simmetriche, e integrità.

Capiamo prima meglio cosa sono gli ***RMI*** [25], che vengono utilizzate per le comunicazione in *Muskel*, e il ***Secure Socket Layer*** (SSL), e poi vediamo come vengono implementate in Java e nell’ambiente Muskel per la programmazione parallela.

2.2

REMOTE METHOD INVOCATION

Remote Method Invocation o **RMI** è una tecnologia che consente agli oggetti Java di effettuare chiamate di metodi remoti. Questa tecnologia include una API (*application programming interface*) il cui scopo esplicito è quello di rendere trasparenti al programmatore quasi tutti i dettagli della comunicazione su rete.

Essa consente infatti di invocare un metodo di un oggetto remoto (cioè appartenente a un diverso processo, potenzialmente su una diversa macchina) *come se* tale oggetto fosse "locale" (ovvero appartenente allo stesso processo in cui viene eseguita l'invocazione).

L'utilizzo di un meccanismo di invocazione remota di metodi in un sistema *object-oriented* comporta notevoli vantaggi di omogeneità e simmetria nel progetto, poiché consente di modellare le interazioni fra processi distribuiti usando lo stesso strumento

concettuale che si utilizza per rappresentare le interazioni fra i diversi oggetti di una applicazione, ovvero la chiamata di metodo.

Il termine ***RMI*** identifica ufficialmente sia la API messa a disposizione del programmatore sia il protocollo di rete usato "dietro le quinte" per il dialogo fra le macchine virtuali Java coinvolte nella comunicazione.

Dell'API e del protocollo esistono due implementazioni di uso comune. La prima (meno recente) ha nome **JRMP** (Java Remote Method Protocol) ed è implementata sul protocollo TCP; la seconda, nota come **RMI-IIOP**, è invece basata sul protocollo IIOP della piattaforma middleware CORBA, che però non è distribuito ufficialmente.

RMI è una libreria Java che consente di interagire con oggetti remoti che girano sulla Java Virtual Machine di host remoti sulla rete.

Con RMI è possibile invocare metodi su oggetti che competono a processi remoti come se fossero oggetti locali.

Viceversa è possibile esportare un oggetto come remoto in modo che processi remoti possano avere accesso diretto ad esso senza dover definire un protocollo e un formato di trasmissione delle informazioni,

RMI si presta bene ad essere utilizzato tra sistemi omogenei (full Java).

Per la comunicazione remota tra sistemi eterogenei è possibile utilizzare:

- o RMI facendo wrapping degli oggetti non Java (utilizzando la Java Native Interface),
- o un altro schema per gestire oggetti distribuiti language-indipendent ad esempio CORBA.
- o RMI/IIOP che consente agli oggetti RMI di comunicare direttamente con oggetti remoti corba su IIOP (Internet inter-ORB protocol), ma come dicevamo questo non viene distribuito ufficialmente.

L’architettura RMI è quella riportata nella figura Fig 2.1.

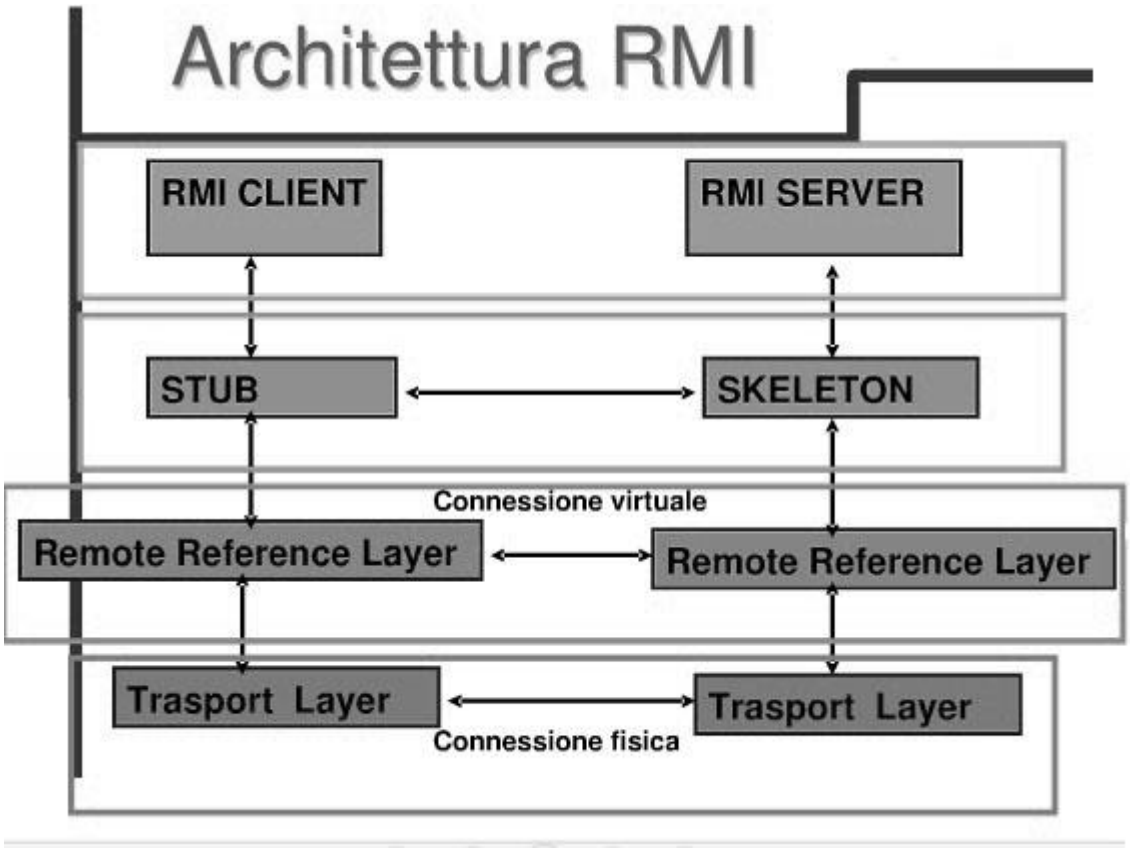


Fig 2.1

L'architettura RMI è strutturata su tre strati:

Stub/skeleton layer: fornisce le interfacce usate dal client e server per le loro interazioni

RemoteReference layer : fornisce un oggetto **RemoteRef** che rappresenta il link all'oggetto remoto che realizza il servizio.

Il RemoteReference layer installa un collegamento virtuale tra i due lati, decodifica le richieste del client e le invia al server, decodifica le richieste e le inoltra allo skeleton.

Transport layer: strato sul quale è realizzato il collegamento fisico; si perde la concezione di oggetto remoto/locale. I dati sono sequenze di byte.

Il collegamento è di tipo sequenziale, per questo si richiede la serializzazione dei parametri da passare ai metodi.

Il tranport layer gestisce il protocollo di conversione delle invocazioni remote dei metodi e l'impacchettamento dei riferimenti ai vari oggetti.

Stub/Skeleton forniscono una duplice rappresentazione dell'oggetto remoto. Lo **stub** rappresenta una simulazione locale sul client dell'oggetto remoto che vive e viene eseguito sul server.

Lo **stub** è un proxy per il reale oggetto remoto del server e fa apparire l'oggetto remoto come locale e, non implementa il metodo ma tutti i meccanismi necessari per consentire al client di interagire con il server, al fine di eseguire il metodo (in modo trasparente).

Lo **skeleton**, speculare dello stub lato server, implementa il lato server dell'invocazione, contiene il codice per ricevere e gestire l'invocazione dallo stub del client e invocare l'oggetto corretto residente sul server.

Il **protocollo RMI** prevede che il client riceva un riferimento (**reference**) dell'oggetto remoto (lo stub dell'oggetto). Invia l'invocazione di un metodo remoto chiamando il metodo su un oggetto stub (esegue i metodi messi a disposizione per l'invocazione remota su un oggetto come se fosse locale).

Lo skeleton realizza la **serializzazione** e **marshalling** degli argomenti dei metodi passati al server in modo da produrre dati gestibili dal Trasport Layer e utilizzabili dall'oggetto remoto, e chiede al Remote Reference Layer di instradare la richiesta verso l'oggetto remoto invocato.

Lato server, il remote reference layer riceve la richiesta dal Trasport Layer e la converte in una richiesta per lo skeleton del server compatibile con l'oggetto riferito.

Lo skeleton converte la richiesta remota nella chiamata corretta ad un metodo sul oggetto reale residente sul server (la conversione consiste nell'**unmarshalling** degli argomenti della chiamata in formato adatto alle caratteristiche dell'oggetto che gira sul server).

Se l'esecuzione del metodo richiede la comunicazione indietro con il client, lo skeleton si occupa di serializzare i dati di ritorno e di rinviarli.

RMI fornisce alcuni servizi base per la gestione delle applicazioni distribuite: **Naming/Registry service**, **Remote Object Activation service**, **Distributed Garbage collection**.

Naming Registry Service. Quando un processo server-side esporta un servizio basato su RMI deve registrare uno o più oggetti RMI con il suo Registry locale (rappresentato da una interfaccia specifica: la **Registry Interface**).

Ciascun oggetto è registrato con il suo nome logico che un client può utilizzare come riferimento. La gestione della coppia oggetto remoto/nome logico è gestita tramite un'interfaccia specifica: la **Naming interface**.

Un client può ottenere un riferimento stub all'oggetto remoto invocando l'oggetto per nome tramite questa interfaccia.

Il metodo ***`Naming.lookup()`*** prende il nome dell'oggetto remoto e lo localizza nella rete.

Quando il metodo ***`lookup()`*** individua l'host dell'oggetto desiderato, consulta il ***`registry`*** RMI dell'host e richiede l'oggetto per nome.

Se il registry trova l'oggetto, genera un riferimento remoto all'oggetto e lo fornisce al processo client.

Il riferimento viene convertito dal processo client in un riferimento stub che è restituito al chiamante.

Una volta ricevuto il riferimento il client può iniziare la conversazione con il server.

Remote Object Activation Service. Il servizio di attivazione di un oggetto remoto: fornisce un modo per consentire l'attivazione di un oggetto sul server sulla base delle necessità del client che lo invoca.

Un server object è registrato con un RMI registry service all'interno di una JVM attiva ed è disponibile per il tempo in cui la JVM è in vita, se si ferma non si ha più modo di invocare l'oggetto remoto, i riferimenti diventano inutilizzabili.

Il Remote Object Activation Service crea il server object dinamicamente all'interno di una virtual machine esistente o nuova. Ottiene un nuovo riferimento all'oggetto creato che verrà passato al client che ne ha richiesto l'attivazione.

Distributed Garbage Collection. Meccanismo di garbage collection specifica per RMI. Realizza la ripulitura delle aree di memoria non più utilizzate.

Per ogni oggetto remoto RMI, il **server reference layer** mantiene la lista di riferimenti remoti registrati dai client, ottenuti esplicitamente tramite ***`lookup()`*** o implicitamente a seguito di una invocazione di metodo remoto.

Quando la VM del client si accorge che l'oggetto remoto non è più referenziato localmente, invia una notifica al server RMI e il server aggiorna la lista dei riferimenti.

Quando un oggetto non è più referenziato da alcun client nè da oggetti locali, viene eliminato attraverso un'operazione di **garbage collection**.

Inoltre il **DGC** prevede l'uso di un **timeout** per l'invocazione di un riferimento remoto. Quando il timeout scade il riferimento viene eliminato dalla lista e viene data notifica ai client che avevano quel riferimento.

Se il client è ancora attivo richiederà al server di mantenere attivo nella lista il riferimento remoto. Il meccanismo evita di occupare memoria con riferimenti inutilizzati da client non più esistenti.

La prima fase per l'uso di RMI è la definizione e creazione di oggetti remoti.

L' oggetto remoto implementa l'interfaccia ***Remote*** e ha metodi che sono eseguibili da un applicazione client non residente sulla stessa macchina virtuale.

L'interfaccia remota, invece rende disponibili dei metodi utilizzabili per l'invocazione a distanza.

Esempio: definizione dell'oggetto non remoto **MyServer.**

```
Public class MyServer {
      Public void String concat(String a, String b)
}
Return a + b;
}
}
```

Trasformazione dell'oggetto nella sua versione remota: definizione della interfaccia remota corrispondente.

Per creare una interfaccia remota è necessario estendere **java.rmi.Remote.**

```
Public interface MyServerInterface extends Remote {
      Public String concat(String a, String b) throws Remote Exception;
}
```

Una volta definita l'interfaccia remota si modifica la classe di partenza in modo che implementi l'interfaccia stessa.

```
Public class MyServerImpl Implements MyServerInterface extends
UnicastRemoteObject {
      Public MyServerImpl() throws RemoteException {
```

```
        …
        }
        Public String concat(String a , String b) throws RemoteException {
        Return a+b;
        }
}
```

Adesso dobbiamo implementare l'interfaccia. Va fatto notare che il nome della classe è stato modificato in modo che implementi l'interfaccia.

A questo punto l'oggetto è visibile all'esterno ma non ancora utilizzabile da RMI: si creano gli **stub** e gli **skeleton**.

Per rendere utilizzabile l'oggetto creato con RMI è necessario compilare opportunamente l'interfaccia per la creazione di STUB e SKELETON.

Nella JDK esiste un compilatore apposito: **rmic**

Con un comando del tipo:

rmic MyServerImpl

si ottengono i due file **MyServerImpl_stub.class** e **MyServerImpl_skel.class**

Ora è necessario abilitare il collegamento tra client e server per l'invocazione remota. Funzione realizzata da Server RMI che è l'applicazione di servizio necessaria per avviare il meccanismo di attivazione remota che istanzia un oggetto remoto e lo registra tramite un **bind** all'interno del RMI **registry**.

Sul lato server lo skeleton notifica di possedere un oggetto abilitato all'invocazione remota tramite il metodo **java.rmi.Naming.bind()** che associa all'istanza dell'oggetto remoto un nome logico che identifica l'oggetto in rete.

Si crea un'istanza dell'oggetto remoto:

MyServerImpl server = new MyServerImpl

Si effettua la registrazione con un nome simbolico:

Naming.bind("pluto", server)

Ogni associazione nome logico – oggetto remoto è memorizzato nel RMI registry gestito con l'istruzione rmiregistry.

start rmiregistry

Il **registry** creato si mette in ascolto di processi locali che vogliano registrare nuovi oggetti, o di client che si connettono per fare lookup di oggetti RMI.

Quando viene registrato l'oggetto, il client è in grado di ottenere un reference all'oggetto con una ricerca utilizzando il nome logico, ad esempio

MyServerInterface server;

String url = "//" +serverhost + "/MyServer";

Server = (MyServerInterface) Naming.lookup(url);

La ***`lookup`*** identifica il nome della macchina che ospita l'oggetto remote e il nome con cui l'oggetto è registrato.

Il metodo ***`list()`*** restituisce la lista di tutti gli oggetti referenziati con il **registry** locale.

Le operazioni di ricerca e registrazione accettano come parametro un URL il cui formato è:

rmi://host:port/name

Dove **host** è il nome del server RMI, **port** è la porta su cui sta in ascolto il registry (default 1099) e **name** è il nome logico.

Gli argomenti vengono passati ai metodi remoti attraverso un processo di serializzazione e deserializzazione.

In realtà l'oggetto serializzato non viene fisicamente spostato dal client al server. Vengono inviate nella rete le informazioni per ricreare una copia dell'oggetto.

Client e server devono disporre dello stesso **bytecode** riferito all'oggetto per poterne ricreare l'istanza.

Si ovvia al problema copiando fisicamente i vari file **.class** sia sul client che sul server.

Se sei vuole evitare la complicazione di installare i file .class ad ogni modifica è possibile sfruttare il meccanismo di scaricamento dalla rete tramite server http.

Trattandosi di codice scaricato via rete, è necessario installare un ***`security manager`*** che consenta queste operazioni in modo controllato.

La classe di libreria ***RMISecurityManager*** fornisce un esempio di security manager per RMI.

Come tutti i meccanismi di comunicazione, anche RMI necessita di implementare una forma sicura di comunicazione, introduciamo quindi il Secure Socket Layer.

2.3

SECURE SOCKET LAYER

Prendiamo in considerazione il caso del ***Secure Socket Layer*** (SSL) e spieghiamo meglio in cosa consiste prima di vederne l'utilizzo insieme agli RMI di Java.

Transport Layer Security (**TLS**) e il suo predecessore Secure Sockets Layer (**SSL**) sono dei protocolli basati su crittografia che permettono una comunicazione sicura e una integrità dei dati su reti TCP/IP.

TLS e SSL cifrano la comunicazione dalla sorgente alla destinazione (end-to-end) sul livello di trasporto.

Varie versioni del protocollo vengono ampiamente utilizzate nelle applicazioni tipo browser, client E-mail, istant messagging e Voice over IP.

TLS è un protocollo standard IETF che, nella sua ultima versione, è definito nella RFC 5246, sviluppata sulla base del precedente protocollo SSL da Netscape Corporation.

L'architettura SSL è mostrata nella Figura 2.2.

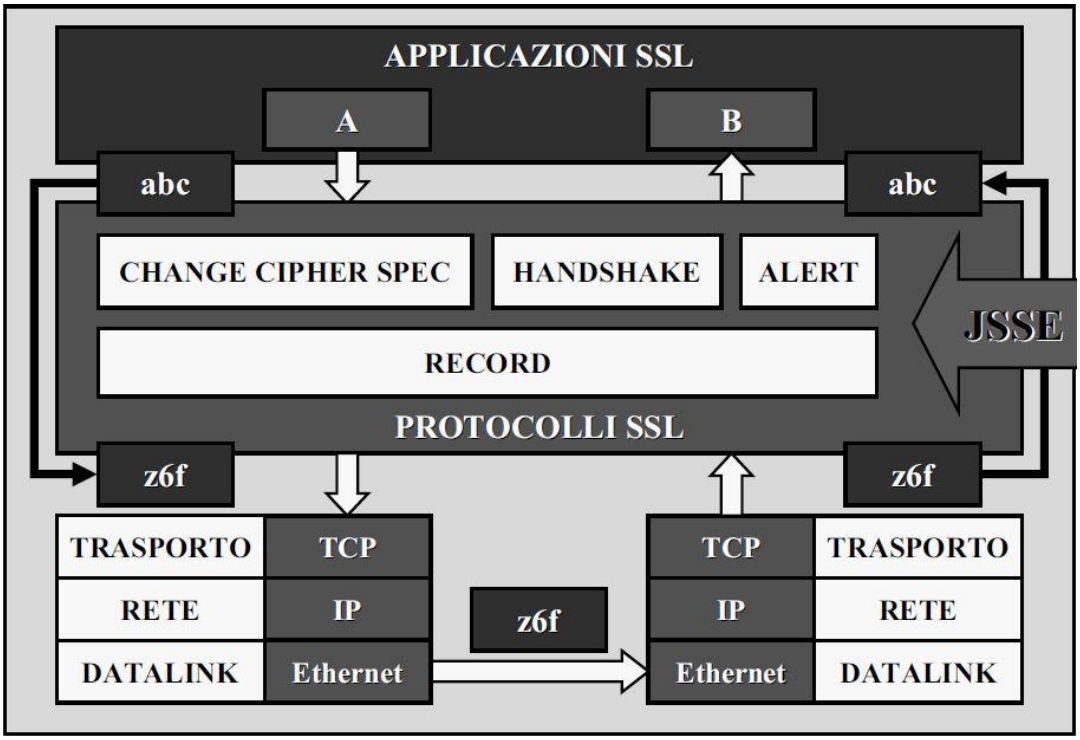


Fig 2.2

SSL e' un protocollo, utilizzato per stabilire comunicazioni sicure tra un Server ed un Client.

Questo sistema non solo garantisce la protezione dei numeri delle carte di credito, dei moduli on-line e dei dati finanziari, ma protegge dalla decodifica e dalla contraffazione un gran numero di informazioni confidenziali, riservate o sensibili.

Quando avviene il collegamento tra un client ed un server (fase di **handshaking**), ha luogo la negoziazione di una "chiave di sessione", che viene stabilita in base alle caratteristiche del client e del server, e che verrà utilizzata per cifrare i dati in transito tra client e server nel corso di una specifica sessione.

La fase di Handshake è mostrata nella Figura 2.3

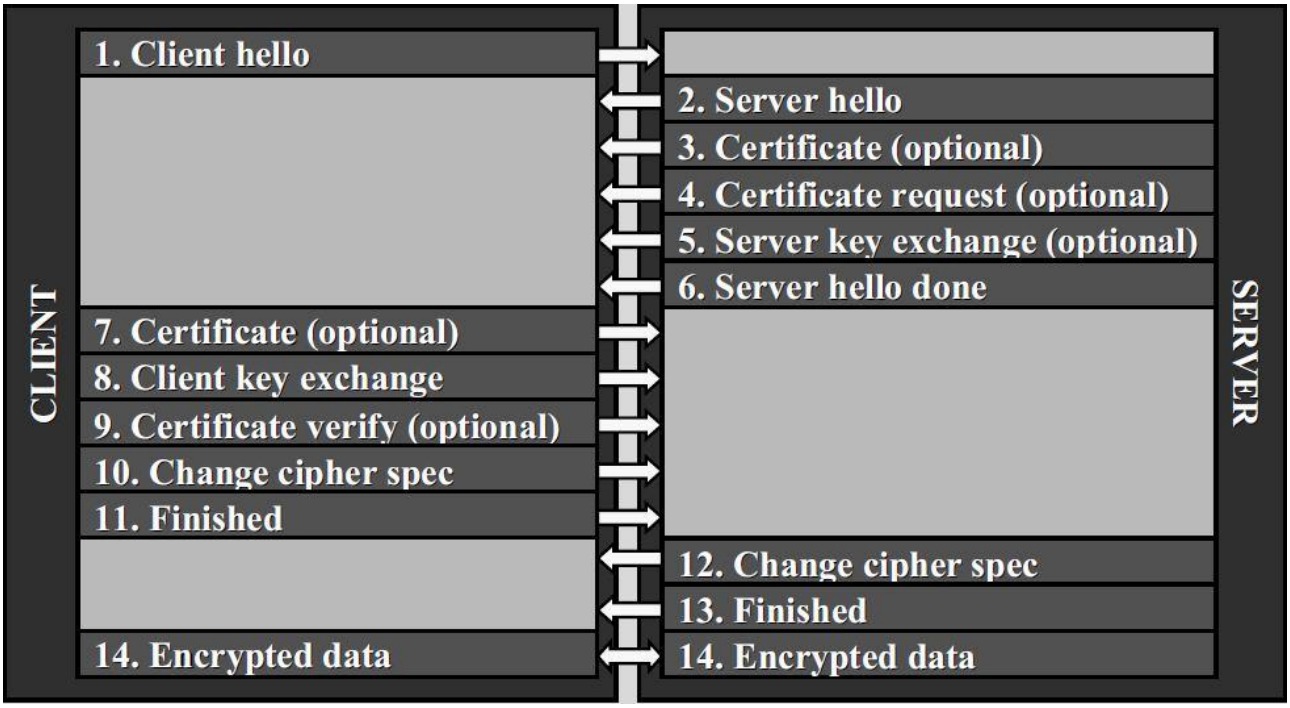


Fig 2.3

Il grado di cifratura si misura in bit: quanto più è lunga la chiave di sessione, tanto più è forte e sicura la cifratura fornita. Una chiave di cifratura a 40 bit è considerata "standard", mentre la chiave di cifratura a 128 bit è conosciuta come "strong encryption".

SSL è una tecnologia che consente di utilizzare socket in modo sufficientemente sicuro, infatti è un protocollo standard per autenticare gli utenti ed allo stesso tempo criptare i dati che poi vengono trasferiti attraverso socket standard.

SSL ha tre interessanti ed utili caratteristiche:

- E' un protocollo pubblico, infatti è stato definito ed implementato da Netscape ma la specifica è pubblica ed è stata oggetto di numerosi e severi controlli.
- E' molto usato. Quasi tutti i linguaggi che usano sockets hanno una libreria **SSL**. Inoltre è facile definire una versione sicura di un protocollo di comunicazione usando SSL invece di Socket standard.
- E' sufficientemente sicuro. SSL è definito come un protocollo di comunicazione sviluppato sopra socket standard. Più precisamente, in una connessione SSL si crea una connessione su socket oridinari. Questa connessione è utilizzata per stabilire il criterio da seguire per lo scambio di informazione in maniera sicura tra le due parti (versione **SSL**, algoritmi di

criptaggio, etc). Dopo questa fase di negoziazione, si usa il socket per trasmettere i dati in forma criptata.

La versione 3.0 del protocollo, rilasciata nel novembre 1996, è un'evoluzione della precedente versione del 1994, SSL 2.0, e rappresenta una soluzione sicura ed efficace per lo scambio di informazioni cifrate.

La versione 3.1 del SSL è invece rappresentata, a tutti gli effetti, dal protocollo TLS 1.0 (Transport Layer Security), che è stato sottoposto all'attenzione del comitato per gli standards IETF (Internet Engineering Task Force) nel 1996.

Il Transport Layer Security Working Group del IETF è il gruppo di lavoro incaricato di rendere conformi agli standards tutti i protocolli Transport Layer come SSL.

In realtà SSL 3.0 e TLS 1.0 vengono comunemente considerati come un identico protocollo, e per questo, anche se l'obiettivo a lungo termine del TLS è quello di sostituirsi a SSL, è assolutamente garantita, per il futuro, la compatibilità all'indietro con SSL 3.0 ed addirittura con SSL 2.0.

Come esempio illustrativo viene adesso mostrato nella Fig 2.4 un diagramma delle iterazioni sulla connessione SSL tra un Client e un Web Server.

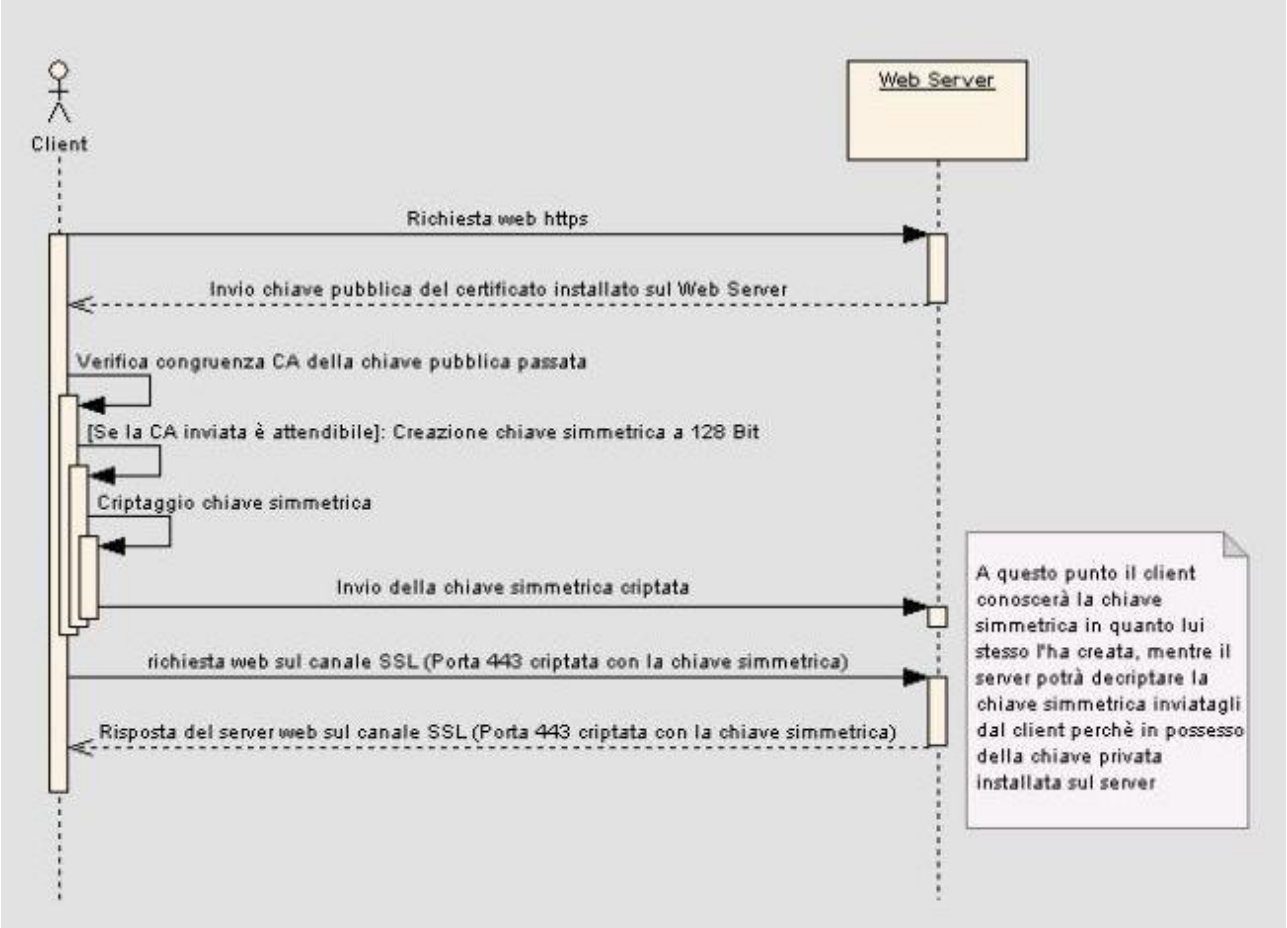


Fig 2.4

All'inizio il Client effetua la richiesta web https al Web Server. Il Client invia anche la lista dei criptosistemi che supporta, smistati per ordine decrescente secondo la lunghezza delle chiavi.

Alla ricezione della richiesta il server invia un certificato al client, contenente la chiave pubblica del server, firmata da un'autorità di cerficazione (CA), nonché il nome del criptosistema in cima alla lista con il quale è compatibile (la lunghezza della chiave di codificazione - 40 bits o 128 bits - sarà quella del criptosistema comune con la chiave di più grandi dimensioni).

Quindi il Client verifica la validità del certificato tramite CA della chiave pubblica ricevuta. Se la CA risulta attendibile, il client crea una chiave simmetrica a 128 Bit.

A questo punto viene criptata la chiave simmetrica, che viene inviata al Web Server.

Il Client conosce la chiave simmetrica in quanto lui stesso l'ha creata, mentre il server può decriptare la chiave simmetrica inviatagli dal client perché in possesso della chiave privata installata sul server.

Le transazioni restanti possono essere effettuate attraverso una chiave di sessione che garantisce l'integrità e la confidenzialità dei dati scambiati.

Da questo punto in poi le richieste web del Client vengono fatte sul canale SSL, sulla porta 443, criptate con la chiave simmetrica.

Anche le risposte del server web viaggiano sul canale SSL, sempre sulla porta 443 criptate con la chiave simmetrica.

In Java l'uso di SSL/TLS si basa sulle classi ***SSLSocket*** e ***SSLServerSocket***, che estendono rispettivamente ***Socket*** e ***ServerSocket***.

Una volta effettuata la creazione del socket, non c'è differenza per l'applicazione rispetto all'uso di socket non crittografici. Le classi e le interfacce necessarie sono nei package **javax.net.*** e **java.net.ssl.***.

Il primo passo è la creazione di una ***SocketFactory***, che è un oggetto che astrae l'operazione di creazione di un socket. Per creare un ***SocketFactory*** in grado di creare Socket SSL occorre usare il metodo ***getDefault()*** della classe ***SSLSocketFactory***.

Una volta ottenuta una factory, si può usare il metodo ***createSocket()*** per creare il Socket vero e proprio.

Una volta creato, il socket si usa come un normale client socket.

Per rendere possibile la creazione del socket SSL, il programma deve conoscere il ***keystore*** e il ***truststore*** e le relative password. E' possibile fornire tali informazioni usando opportune proprietà di sistema (*System.setProperty()*).

Keystore e ***truststore*** sono i file dove vengono salvate le chiavi (pubbliche e private) e i certificati creati per l'autenticazione (spiegati dettagliatamente al paragrafo 3.6).

La creazione di server socket SSL è analoga alla creazione di socket, ed occorre usare le classi ***ServerSocketFactory***, ***SSLServerSocketFactory*** e ***SSLServerSocket***.

Una volta creato, il server socket si usa esattamente com un ServerSocket non crittografato. Il ***keystore*** e il ***truststore*** devono essere specificati con le stesse proprietà di sistema.

Per default i socket creati dalla factory effettuano l'autenticazione del solo server. Se si desidera l'autenticazione anche del client, occorre convertire il socket in un ***SSLServerSocket*** e richiamare il metodo ***setNeedClientAuth()***.

Un esempio di utilizzo dei Socket Client e Server con SSL, per la creazione di una banale applicazione Client/Server che stampa "Hello World", viene mostrata nel codice seguente.

```
import java.io.*;
import java.net.*;
import javax.net.*;
import javax.net.ssl.*;

public class SSLEchoClient {
   public static final int PORT=7777;

   public static Socket createSocket(String host, int port) throws IOException {
      SocketFactory factory=SSLSocketFactory.getDefault();
      Socket sock=factory.createSocket(host, port);
      return sock;
   }

    // continua ...

   public static void main(String args[]) throws IOException {
      System.setProperty("javax.net.ssl.keyStore", "clientkeystore.jks");
      System.setProperty("javax.net.ssl.keyStorePassword", "pippobaudo");
      System.setProperty("javax.net.ssl.trustStore", "clienttruststore.jks");
      System.setProperty("javax.net.ssl.trustStorePassword", "pippobaudo");

      Socket sock=createSocket(args[0], PORT);
      OutputStream os=sock.getOutputStream();
      Writer wr=new OutputStreamWriter(os, "UTF-8");
      PrintWriter prw=new PrintWriter(wr);
      prw.println("Hello, world");
      prw.flush();

      InputStream is=sock.getInputStream();
      Reader rd=new InputStreamReader(is, "UTF-8");
      BufferedReader brd=new BufferedReader(rd);
      String answer=brd.readLine();
      System.out.println(answer);

      sock.close();
   }
}
```

```
import java.io.*;
import java.net.*;
import javax.net.*;
import javax.net.ssl.*;

public class SSLEchoServer implements Runnable {
    public static final int PORT=7777;

    private Socket sock;

    public SSLEchoServer(Socket s) {
        sock=s;
    }

    public static ServerSocket createServerSocket(int port) throws IOException {
        ServerSocketFactory factory=SSLServerSocketFactory.getDefault();
        SSLServerSocket sock=(SSLServerSocket) factory.createServerSocket(port);
        sock.setNeedClientAuth(true);
        return sock;
    }
    // continua ...

    public static void main(String args[]) throws IOException {
        System.setProperty("javax.net.ssl.keyStore", "servkeystore.jks");
        System.setProperty("javax.net.ssl.keyStorePassword", "pippobaudo");
        System.setProperty("javax.net.ssl.trustStore", "servtruststore.jks");
        System.setProperty("javax.net.ssl.trustStorePassword", "pippobaudo");

        ServerSocket serv=createServerSocket(PORT);
        while (true) {
            Socket sock=serv.accept();
            SSLEchoServer server=new SSLEchoServer(sock);
            Thread t=new Thread(server);
            t.start();
        }
    }
    // continua...


    public void run() {
        try {
            BufferedReader brd=new BufferedReader(
                   new InputStreamReader(
                          sock.getInputStream(), "UTF-8"));
            String s=brd.readLine();

            PrintWriter prw=new PrintWriter(
                   new OutputStreamWriter(
                          sock.getOutputStream(), "UTF-8"));
            prw.print(s);
            prw.println(s);
            prw.flush();
        } catch (IOException exc) {
            System.out.println("Eccezione I/O:" + exc);
            exc.printStackTrace();
        } finally {
            try { sock.close(); }
            catch (IOException exc2) { }
        }
    }
}
```

JSSE (Java Secure Socket Extension [23]) implementa la versione Java del protocollo SSL e TLS, con il quale è possibile garantire sicuri trasferimenti di dati sopra il protocollo TCP/IP e sopra altri protocolli come Http, Telnet, NNTP, LDAP, IMAP, FTP, che girano sullo strato TCP/IP.

JSSE comprende molti degli algoritmi e dei concetti che ritroviamo nel JCE [23] (Java Cryptography Extension) ma vengono applicate automaticamente sotto una extension API per stream socket semplici.

L'architettura JSSE è basato sugli stessi principi che ritroviamo nella Java Cryptography Architecture, come: l'implementazione indipendente, gli algoritmi indipendenti e l'architettura "Provider".

Il package **javax.net.ssl** contiene una serie di classi e interfacce per le API JSSE. Il package **javax.net** non riguarda direttamente JSSE, ma è necessario per supportare al livello base le funzionalità dei client socket e dei server socket factory.

Anche il package **javax.security.cert** non riguarda direttamente JSSE, ma è necessario per fornire i certificati di base.

Infine il package **com.sun.net.ssl** fornisce classi per la creazione e la configurazione di socket factory sicure. Le classi sono fornite con l'implementazione "Reference" di JSSE. Essi non fanno parte delle API standard JSSE1.0.2.

Il Diagramma delle classi del JSSE è riportato nella Figura 2.5.

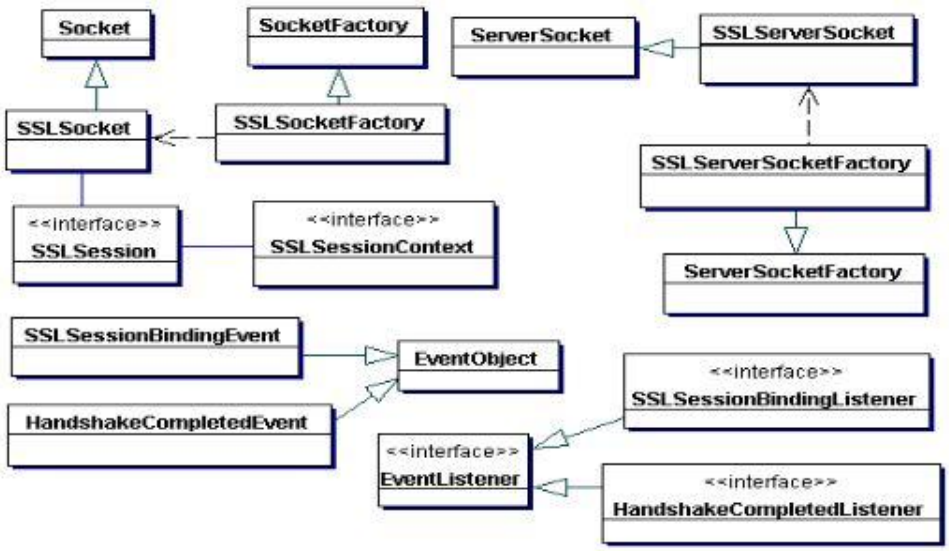


Fig 2.5

2.4

RMI & SSL IN JAVA

In Java, la tecnologia RMI per la comunicazione e la tecnologia SSL per la sicurezza possono essere utilizzate congiuntamente usando gli ***SslRMISocketFactory*** che utilizzano socket SSL/TLS invece dei semplici socket TCP. SslRMISocketFactory utilizza ***SslRMIServerSocketFactory*** e ***SslRMIClientSocketFactory*** per gestire la comunicazione tra il client e il server.

Una istanza di SslRMIServerSocketFactory viene usata da RMI a runtime per ottenere il socket per la chiamata RMI via SSL. Questa classe implementa ***RMIServerSocketFactory*** sopra il protocolli SSL o TLS, e crea un socket SSL usando il metodo ***getDefault***.

Questo comportamento può essere modificato implementando il metodo ***createServerSocket***.

Inoltre, tutte le istanze di questa classe condividono la stessa ***keystore***, e lo stesso ***truststore***, quando l'autenticazione del cliente è richiesta dal server.

Per l'autenticazione di client e server, il Java Development Kit include un tool (***keytool*** da usare da linea di comando), per gestire chiavi e certificati.

Le chiavi pubbliche e private vengono memorizzate nel ***keystore***, invece i certificati ritenuti "fidati" sono memorizzati nel ***truststore*** (una spiegazione più dettagliata viene data nel paragrafo 3.6).

Una istanza di ***SslRMIClientSocketFactory*** viene usata a runtime per ottenere un client socket per la chiamata RMI via SSL.

Questa classe implementa ***RMIClientSocketFactory*** sopra il protocolli SSL o TLS, e crea un SSL socket usando il metodo ***getDefault***. Anche questo comportamento può essere modificato implementando il metodo ***createSocket***.

Tutte le istanze di questa classe sono funzionalmente equivalenti. In particolare, condividono lo stesso ***truststore***, e lo stesso ***keystore*** quando l'autenticazione del client è richiesta dal server.

Se la system properties **javax.rmi.ssl.client.enabledChiperSuite** è specificata, il metodo **createSocket** chiamerà **SSLSocket.setEnableChiperSuites(String[])** prima di restituire il socket. Il valore delle system properties è una stringa che è una lista, separata da punti, di chiper suite SSL/TLS, da abilitare.

Le **Chiper Suite** supportate da Sun JSSE in ordine di preferenza per default sono riportati in Figura 2.6.

CIPHER SUITES	NUOVE IN J2SEv1.4
SSL_RSA_WITH_RC4_128_MD5	
SSL_RSA_WITH_RC4_128_SHA	
TLS_RSA_WITH_AES_128_CBC_SHA	X
TLS_DHE_RSA_WITH_AES_128_CBC_SHA	X
TLS_DHE_DSS_WITH_AES_128_CBC_SHA	X
SSL_RSA_WITH_3DES_EDE_CBC_SHA	
SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA	X
SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA	
SSL_RSA_WITH_DES_CBC_SHA	
SSL_DHE_RSA_WITH_DES_CBC_SHA	X
SSL_DHE_DSS_WITH_DES_CBC_SHA	
SSL_RSA_EXPORT_WITH_RC4_40_MD5	
SSL_DSS_EXPORT_WITH_DES40_CBC_SHA	X
.....	

Fig 2.6

Se la system properties **javax.rmi.ssl.client.enableProtocols** è specificata, il metodo **createSocket** chiamerà **SSLSocket.setEnableProtocol(String[])** prima di restituire il socket. Il valore delle proprietà di sistema è una stringa che è una lista, separata da punti, di protocolli SSL/TLS da abilitare.

Le funzionalità crittografiche implementate in JSSE sono riportate nella Figura 2.7.

ALGORITMO CRITTOGRAFICO	PROCESSO CRITTOGRAFICO	LUNGHEZZA CHIAVE (BITS)
RSA	autenticazione e scambio della chiave	2048 (autenticazione) 2048 (scambio della chiave) 512 (scambio della chiave)
RC4	cifratura	128 128 (40 effettivi)
DES	cifratura	64 (56 effettivi) 64 (40 effettivi)
TripleDES	cifratura	192 (112 effettivi)
Diffie-Hellman	scambio della chiave	1024 512
DSA	autenticazione	1024

Fig 2.7

Mostriamo adesso un esempio di utilizzo dell'***SslRMISocketFactory*** [37].

In questo esempio viene mostrato come usare RMI sopra lo strato di trasporto SSL usando JSSE.

Il server eseguirà ***HelloImpl***, che setta l'***rmiregistry*** interno (invece di utilizzare i comandi rmiregistry), mentre il client eseguirà ***HelloClient*** e comunicherà sopra una connessione sicura.

Per tirare su la configurazione di questo esempio sono necesssary i seguenti passi:

```
% javac *.java      //compilare le classi
% rmic HelloImpl    //generare lo stub
//lanciare il server con una policy corretta
% java -Djava.security.policy=policy HelloImpl
% java HelloClient  //quindi su un'altra shell lanciare il client
```

Per il server, è stato installato un **RMI security manager**, e viene fornito un file di **policy** per garantire i permessi nell'accettare le connessioni da alcuni host. Ovviamente, i permessi a tutti non deveno essere dati in ambienti di produzione.

E’ necessario nel file di ***policy*** dare i corretti privilegi per la rete con:

```
permission java.net.SocketPermission \
    "hostname:1024-", "accept,resolve";
```

Le classi utilizzate per usare SSL/TLS, sul quale si basano attualmente gli RMISocketFactory sono, ***javax.rmi.ssl.SslRMIClientSocketFactory*** e ***javax.rmi.ssl.SslRMIServerSocketFactory***.

Queste classi usano i metodi ***SSLSocketFactory.getDefault()*** e ***SSLServerSocketFactory.getDefault()***, per ottenere i Socket e quindi sarà necessario configurare le system properties o i comandi opportuni per dire dove sono memorizzati il ***keystore*** e il ***truststore***.

Le classi relative all’esempio citato sono le seguenti:

Hello.java è la classe dove viene creata l’interfaccia ***Hello*** che estende la classe ***Remote***.

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Hello extends Remote {
    String sayHello() throws RemoteException;
}
```

HelloImpl.java è la classe per la parte relativa al server RMI, dove nel costruttore vengono istanziati due oggetti, il primo della classe ***SslRMIClientSocketFactory*** e il secondo della classe ***SslRMIServerSocketFactory***.

La classe ha un metodo ***sayHello()*** che restituisce una stringa “Hello World”, e nel ***main*** viene creato l’oggetto ***registry*** passandogli come parametro: la ***porta*** sulla quale gira il registry, e due istanze, una della classe ***SslRMIClientSocketFactory*** e una della classe ***SslRMIServerSocketFactory***.

Viene quindi istanziato un oggetto ***obj*** della classe ***HelloImpl*** e viene registrata una istanza dell’oggetto nel ***registry*** con il nome simbolico “***HelloServer***”.

```
import java.io.*;
import java.net.InetAddress;
import java.rmi.RemoteException;
```

```
import java.rmi.RMISecurityManager;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

public class HelloImpl extends UnicastRemoteObject implements Hello {

    private static final int PORT = 2019;

    public HelloImpl() throws Exception {
      super(PORT,
            new  javax.rmi.ssl.SslRMIClientSocketFactory(),
            new javax.rmi.ssl.SslRMIServerSocketFactory());
    }

    public String sayHello() {
      return "Hello World!";
    }

    public static void main(String args[]) {

      // Create and install a security manager
      if (System.getSecurityManager() == null) {
          System.setSecurityManager(new RMISecurityManager());
      }

      try {
          // Create SSL-based registry
          Registry registry = LocateRegistry.createRegistry(PORT,
              new  javax.rmi.ssl.SslRMIClientSocketFactory(),
            new javax.rmi.ssl.SslRMIServerSocketFactory());
          HelloImpl obj = new HelloImpl();

          // Bind this object instance to the name "HelloServer"
          registry.bind("HelloServer", obj);

          System.out.println("HelloServer bound in registry");
      } catch (Exception e) {
          System.out.println("HelloImpl err: " + e.getMessage());
          e.printStackTrace();
      }
    }
}
```

HelloClient.java è la classe per la parte relativa al client RMI, dove una volta ottenuto, con l'oggetto ***obj***, un riferimento al registry richiamando il metodo ***getRegistry()***, viene fatta la ***lookup*** con il nome del servizio "***HelloServer***".

Il metodo ***getRegistry*** ha come parametri: l'indirizzo del l'***host*** (in questo caso localhost), la ***porta*** sulla quale gira il registry, e una istanza della classe ***SslRMIClientSocketFactory***.

Sull'oggetto ***obj*** viene quindi richiamato il metodo ***sayHello()*** della classe ***HelloImpl***.

```
import java.net.InetAddress;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class HelloClient {

    private static final int PORT = 2019;

    public static void main(String args[]) {
      try {
          // Make reference to SSL-based registry
          Registry registry = LocateRegistry.getRegistry(
            InetAddress.getLocalHost().getHostName(), PORT,
            new  javax.rmi.ssl.SslRMIClientSocketFactory());

          // "obj" is the identifier that we'll use to refer
          // to the remote object that implements the "Hello"
          // interface
          Hello obj = (Hello) registry.lookup("HelloServer");

          String message = "blank";
          message = obj.sayHello();
          System.out.println(message+"\n");
      } catch (Exception e) {
          System.out.println("HelloClient exception: " + e.getMessage());
          e.printStackTrace();
      }
    }
}
```

Per utilizzare le classi menzionate, è necessario specificare il ***keystore*** usando le System properties:

-Djavax.net.ssl.keyStore=testkeys

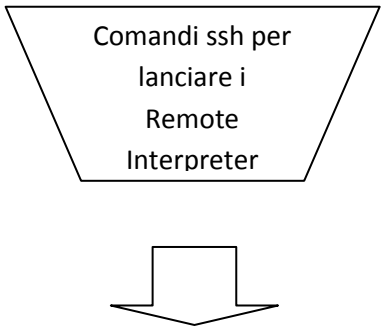
-Djavax.net.ssl.keyStorePassword=passphrase

Verrà adesso esposto nel paragrafo 2.4 il progetto logico relativo alle nuove feature introdotte nella nuova versione di muskel, evidenziando principalmente l'aspetto relativo alla sicurezza.

2.5

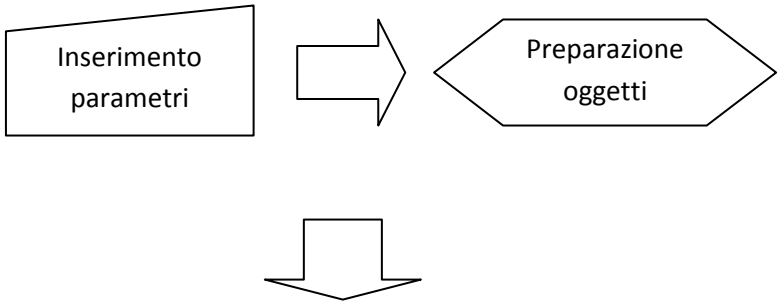
PROGETTO LOGICO

La prima cosa che dovrà essere fatta, per utilizzare questa evoluzione della versione di muskel (2.0), è quella di lanciare manualmente, ad esempio tramite comandi ssh, gli interpreti remoti nelle macchine sulle quali si vorrà eseguire la computazione.



Gli interpreti remoti potranno essere lanciati specificando da linea di comando i parametri per l'esecuzione di interpreti remoti che utilizzano o non utilizzano SSL.

Verrà quindi fornito dal programmatore una configurazione iniziale per capire chi sono gli oggetti protagonisti della computazione: grado di parallelismo, skeleton, tipo di input stream e output stream.



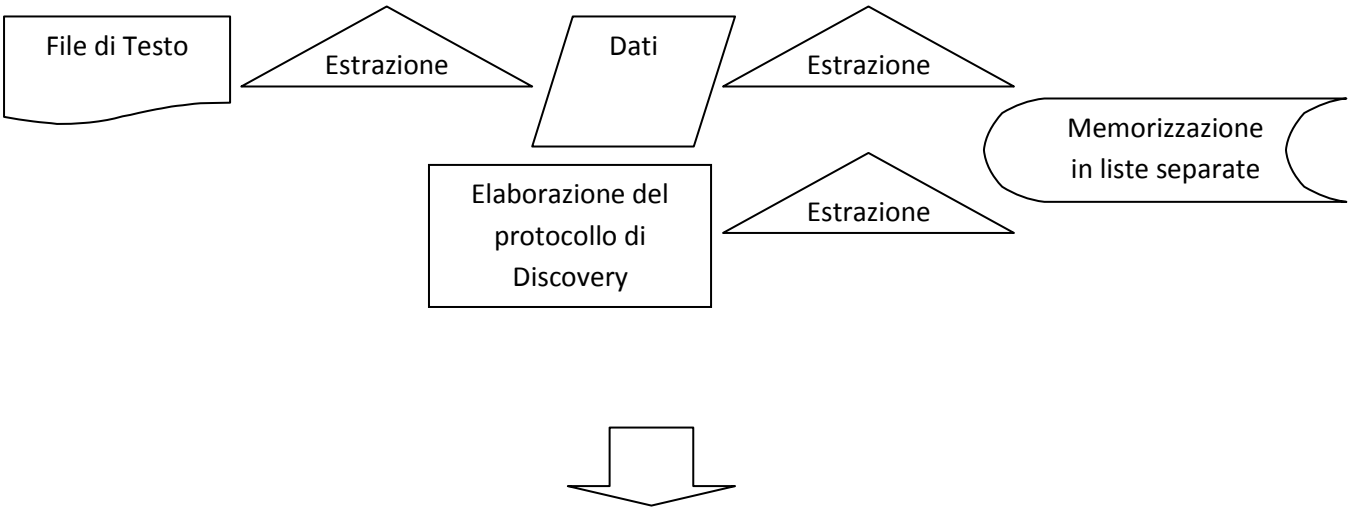
L'idea di base è stata quella di dividere in due classi gli host, inserendoli in liste separate.

Gli host sicuri sui quali si assume che la comunicazione avvenga su un collegamento non pericoloso verranno caricati da un file di testo ***`hosts.conf`***.

Questa è una semplificazione introdotta in questa versione di muskel che delega al programmatore il compito di configurare opportunamente le macchine sicure e non

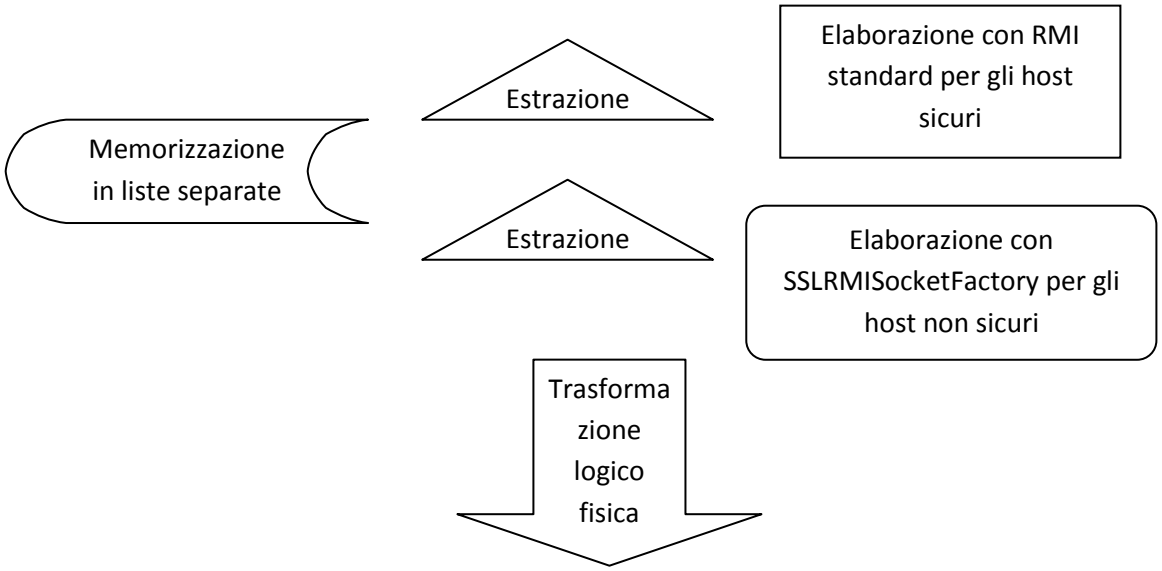
sicure lanciando i comandi corretti per l'esecuzione della forma dell'interprete remoto richiesto.

Gli host non sicuri, saranno trovati nella fase di discovery, utilizzando un protocollo multicast peer-2-peer.



Una volta memorizzati gli indirizzi IP degli interpreti remoti a disposizione, verrà lanciata l'elaborazione cercando di soddisfare il grado di parallelismo richiesto con il contratto di performance fornito dal programmatore.

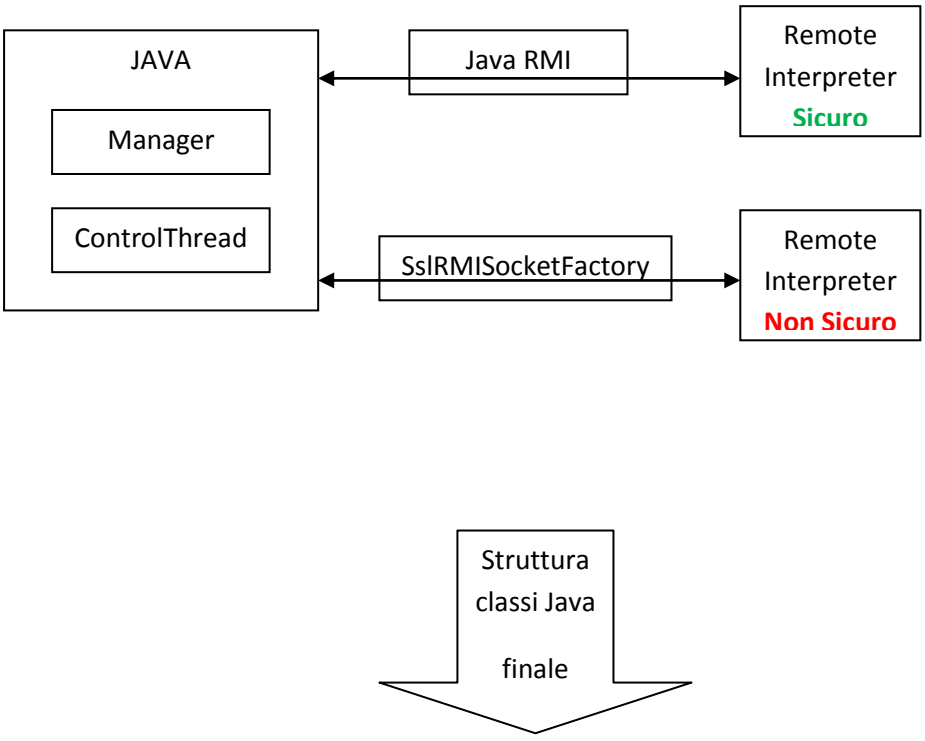
Questo verrà fatto utilizzando le forme di comunicazione più o meno sicuri in base alla pericolosità del collegamento attraverso il quale si comunica con il worker remoto.



La comunicazione e quindi, lo scambio di codice e dati, per eseguire la computazione utilizzeranno due diverse forme per implementare la comunicazione.

Se l'host è presente nella lista ***hostsicuri*** allora, la comunicazione tra il ***controlthread***, e quel ***RemoteInterpreter*** avviene tramite il protocollo RMI standard.

Se invece l'host non è considerato sicuro, e quindi non è presente nel file ***hosts.conf***, la comunicazione tra il ***controlthread*** e il ***RemoteInterpreter*** avviene tramite ***SslRMISocketFactory***.



Nel grafico 2.8 verrano mostrate le classi relative alle modifiche fatte in questa nuova versione di muskel.

Queste sono le classi per l'implementazione delle nuove feature:

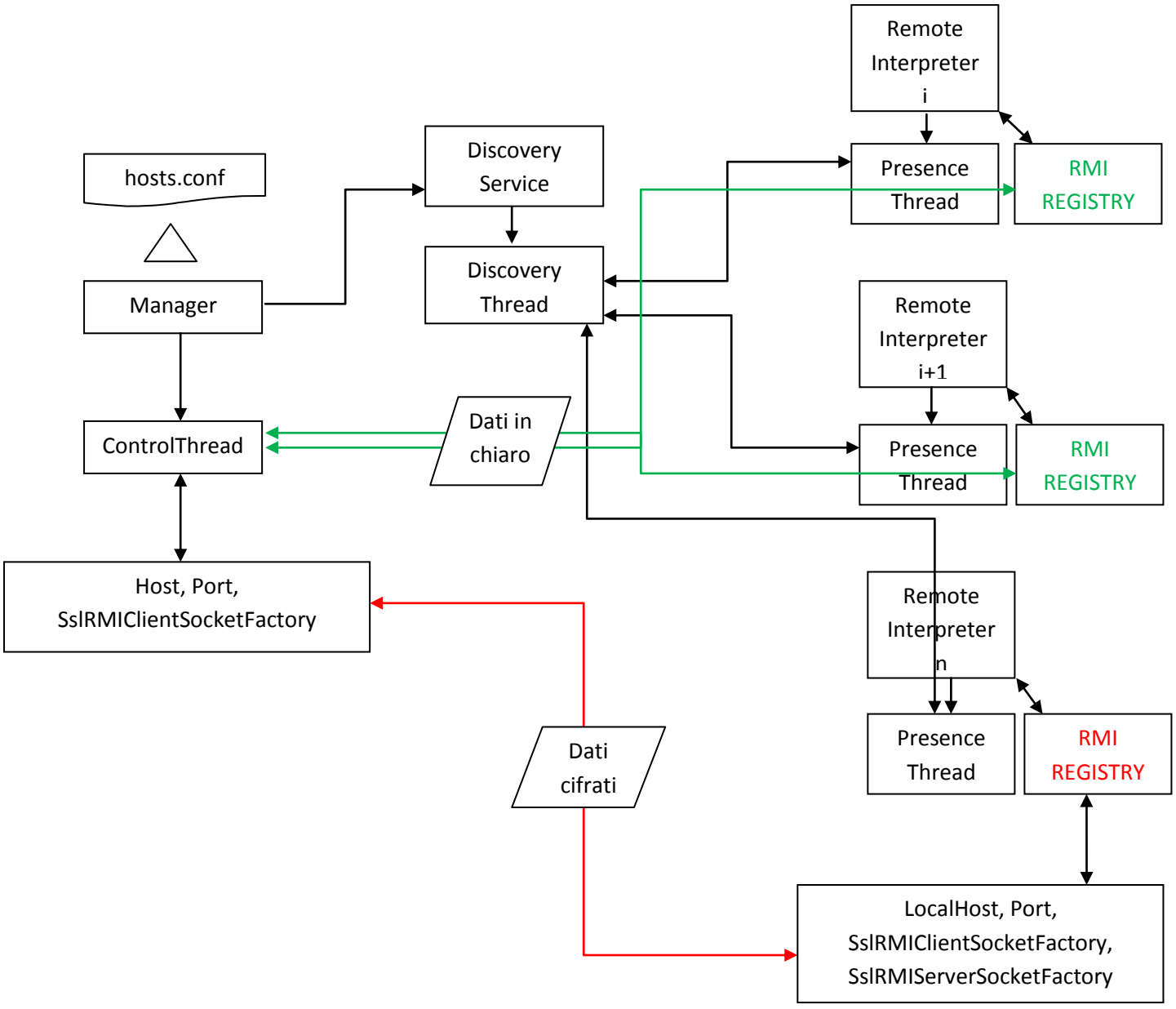


Fig 2.8 Grafico delle classi in muskel

Alcuni dettagli delle nuove feature, introdotte nella nuova versione di Muskel, sono riportati nel Capitolo 3 che segue.

Capitolo 3

IMPLEMENTAZIONE

3.1

INTRODUZIONE

Nell'evoluzione di questa nuova versione di Muskel (2.0), discussa in questa tesi, si darà maggior risalto all'aspetto relativo alla sicurezza di codice e dati trasmessi tra i ***controlthread***, in esecuzione sulla macchina dell'utente, e i ***RemoteInterpreter***, in secuzione sulle macchine remote, quando vengano fatti girare su reti pubbliche e/o non protette.

L'interprete Muskel (in particolare l'oggetto della classe ***Manager*** che sovraintende all'intera computazione) caricherà da un file di testo l'indirizzo IP, delle macchine considerate sicure (questa è una semplificazione implementata in questa versione di Muskel), e sulle quali potrà essere usato semplicemente ***RMI*** come strumento di comunicazione con gli host remoti.

Una volta che il ***DiscoveryThread*** (classe che si occupa di reclutare i worker) avrà restituito il numero di worker che è riuscito a trovare, attraverso lo scambio di messaggi multicast, il ***manager*** confronterà l'indirizzo di tali host con quelli letti dal file e applicherà due diverse politiche.

Per ogni host sicuro, verrà istanziato un ***controlthread*** passando come parametro un variabile booleana ***Sicuro*** impostata a ***true***, che una volta lanciato si occupera di comunicare tramite RMI con l'host remoto.

Per gli host considerati non sicuri, verrà istanziato un ***controlthread*** dove stavolta la variabile ***Sicuro*** è impostata a ***false***, e la comunicazione con l'host remoto avverrà tramite ***SslRMISocketFactory***.

Sarà inoltre introdotta, in questa nuova versione di Muskel, una feature per l'implementazione di una politica ***Best Effort*** nella scelta del parallelismo da utilizzare.

Il programmatore deciderà sempre attraverso il contratto di performance passato al ***manager***, il grado di parallelismo del programma.

Tuttavia, per fare in modo che la computazione prosegua, anche se il numero di macchine richieste nel contratto di performance, non viene rispettato, è stato introdotto un timer.

Il ***DiscoveryService*** aspetterà un tempo fissato durante il quale cercherà di reperire il numero di macchine richiesto con il contratto di performance.

Quando il suddetto tempo sarà trascorso il ***DiscoveryService*** farà proseguire la computazione impostando il valore dei ***discoveredWorker*** al valore attuale.

La computazione potrà quindi essere avviata, visto che il programmatore avrà opportunamente inserito i parametri (grado di parallelismo, skeleton, tipo di input output stream) di configurazione all'inizio del programma.

La computazione avviene distribuendo i ***task*** tra i vari Worker, cercando di bilanciare il carico.

In questa nuova versione di muskel, inoltre, sarà introdotta nel ***manager*** una politica di controllo dello stato dei worker remoti attraverso "Thread Group".

Infatti, tutti i ***controlthread*** istanziati faranno parte di un ***ThreadGroup*** che terrà costantemente sotto controllo lo stato dei ***controlthread***.

Qualora si verificasse qualche problema, (problemi di rete, guasti, ecc), il ***manager*** si accorgerà della modifica dello stato di quel ***controlthread***, che passerà da "RUNNABLE" a "TERMINATED" e attraverso una variabile "**blocked**" terrà conto nel numero di worker bloccati.

Una volta che tutti i ***controlthread*** avranno finito la computazione sull'interprete remoto associato, aspetteranno di terminare andando in stato di "WAITING" e incrementando anche in questo caso la variabile ***blocked***.

La variabile ***blocked*** viene utilizzata per capire quando la computazione è finita e i ***controlthread*** possono terminare.

Un ***controlthread*** può terminare o perché il ***RemoteInterpreter*** associato ha finito di calcolare, o perché il ***RemoteInterpreter*** ha smesso di funzionare (guasti, problemi alla rete, ecc.).

Se il ***RemoteInterpreter*** ha smesso di funzionare dovranno essere reinseriti in coda nel ***MdfiPool*** i task non calcolati, per poterli ridistribuire ai ***RemoteInterpreter*** rimasti.

Verranno adesso presentati i dettagli più significativi delle modifiche apportatate alla versione di muskel (2.0), e cioè quelli che riguardano:

- il caricamento degli host sicuri da file di testo (***hosts.conf***),
- l'implementazione del ***Best Effort*** nella gestione del parallelismo,
- l'implementazione della gestione della ***RemoteException*** nel ***controlthread*** per il fallimento di un worker e relativa comunicazione al ***manager***,
- implementazione della parte Client nelle comunicazioni tra ***controlthread*** e ***RemoteInterpreter*** utilizzando SSL,
- implementazione della parte Server nelle comunicazioni tra ***controlthread*** e ***RemoteInterpreter*** utilizzando SSL,
- creazione e utilizzo delle chiavi e dei certificati per l'autenticazione SSL attraverso il ***keytool***, contenuto nel JDK.

3.1

IMPLEMENTAZIONE DELLA RICERCA DEGLI HOST NEL MANAGER

Per l'implementazione degli host sicuri è stato creato un file di testo "**hosts.conf**" dove vengono memorizzati, prima dell'inizio della computazione, tutti gli indirizzi IP per le macchine da considerare SICURE.

Il file viene letto utilizzando le classi ***FileInputStream***, ***BufferedReader*** e ***InputStreamReader***.

Una volta che il ***manager*** avrà caricato gli host sicuri nella lista ***hostsicuri*** e, avrà controllato la correttezza degli indirizzi IP, chiamerà il metodo ***getNWorkers(nw)*** della classe ***DiscoveryService***.

Tale metodo restituirà un numero di ***RemoteInterpreter*** pari a quelli richiesti dal programmatore nel contratto di performance.

Il codice utilizzato per il caricamento degli host dal file ***hosts.conf*** è il seguente:

```
List<String> hostsicuri = new ArrayList<String>();
try {
	FileInputStream fis = new FileInputStream("./hosts.conf");
	BufferedReader br = new BufferedReader(new InputStreamReader(fis));
	try {
		String line;
		/*"indirizzio IP standard o hostname"*/
		while ((line = br.readLine()) != null)    {
			line = line.trim();//elimino gli spazi

			if(line.isEmpty() ||
			line.startsWith(COMMENT_SEPARATOR))//linea vuota o commenti
			continue;

		// ottiene un'istanza di InetAddress a partire dall'indirizzo IP
			InetAddress inetAddress = InetAddress.getByName(line);
			// risolve il nome
			String hostname = inetAddress.getHostName();
			String hostaddress = inetAddress.getHostAddress();
			// stampa in console l'indirizzo risolto
			System.out.println("host name: " + hostname + " per
			l'indirizzo " + hostaddress);

			//controllo se è un ip corretto
			if(line.equals(inetAddress.getHostAddress())){
				//aggiungo alla lista
				hostsicuri.add(hostaddress);
				}
			}
			//stampo in console gli host sicuri
			System.out.println("Host Sicuri: " + hostsicuri);
	} catch (IOException e) {
		e.printStackTrace();
	}
} catch (FileNotFoundException e) {
	System.out.println("File di config non trovato");
}
```

A questo punto, il programmatore tramite il contratto di performance avrà richiesto ***nw*** worker, e il ***manager*** chiederà al ***DiscoveryService*** di fornire il numero di macchine richiesto.

Le macchine trovate nella ricerca fatta dal ***DiscoveryThread*** verranno controllate per vedere se appartenengono alla classe della macchine sicure o meno, tramite due liste ***hostsicuri2*** e ***hostnonsicuri*** dove gli host verranno aggiunti.

La verifica degli host e la suddivisione in sicuri e non, è rappresentata dal seguente codice:

```
String [] machines = ds.getNWorkers(nw); // search machines
// may be less than nw, risolta con Best Effort
List<String> hostsicuri2 = new ArrayList<String>();
List<String> hostnonsicuri = new ArrayList<String>();
for(int i=0; i<machines.length; i++) {
      if (hostsicuri.contains(machines[i])){
            hostsicuri2.add(machines[i]);
            System.out.println("Host sicuro trovato: " +  machines[i]);
      } else {
            hostnonsicuri.add(machines[i]);
            System.out.println("Host non sicuro trovato " + machines[i]);
      }
}
```

3.2

IMPLEMENTAZIONE "BEST EFFORT"

Per implementare una politica " ***Best Effort*** " nell'implementazione del parallelismo è stato modificato il metodo ***getNworkers(int nw)*** all'interno della classe ***DiscoveryService***.

Prima di questa implementazione, si aspettava di trovare ***nw*** worker prima di proseguire la computazione, cercando di rispettare un certo grado di parallelismo.

Implementando questa politica si è voluto far proseguire la computazione anche quando il ***DiscoveryService*** non restituisce un numero pari ai ***RemoteInterpreter*** richiesti dal programmatore con il contratto di performance.

E' stata introdotta una ***wait*** di 10000 msec, come timeout, dopo il quale se non sono stati trovati gli "nw worker" richiesti per il parallelismo, si opta col proseguire la computazione, impostando ***nw*** al valore dei worker fino a quel momento trovati, ritornando quel valore come risultato.

Sapendo di testare la computazione su host in rete locale collegati tramite Fast Ethernet, è stato impostato il tempo del timeout a 10 secondi, considerando questo tempo adeguatamente sufficiente per permettere ai ***RemoteInterpreter*** di rispondere ai messaggi multicast inviati dal manager attraverso il ***DiscoveryService***.

Per test su reti di tipo diverso, i valori del timeout della ***wait***, dovrebbero venire configurati in maniera diversa.

Il codice per il metodo ***getNWorkers*** che ritorna un numero di worker richiesti dal programmatore, o scaduto il timeout applica **Best Effort**, è il seguente:

```
public synchronized String [] getNWorkers(int nw) {
	while(discoveredWorkers.size() < nw) {
		try {
			System.out.println("Aspetto di trovare " + nw + " worker");

			wait(10000);//aspetto un pò di tempo fissato

			//se ancora non ho trovato il numero necessario di worker
			if (discoveredWorkers.size() < nw )

				nw = discoveredWorkers.size();//vado avanti con quelli
				trovati

			System.out.println("Scaduto il timeout, proseguo con " + nw +
			" worker");
		} catch (InterruptedException e) {
				e.printStackTrace();
		}
	}
	String [] v = new String [nw];
	for(int i=0; i<nw; i++) {
		v[i] = (String) discoveredWorkers.remove(0);
		givenWorkers.add(v[i]);
	}
	System.out.println("getNWorkers returned a vector with "+nw+" workers");
	return v;
	}
```

3.3

IMPLEMENTAZIONE DELLA GESTIONE TRA MANAGER E CONTROLTHREAD DELLA REMOTE EXCEPTION

Per fare in modo che il ***manager*** si accorga che un nodo remoto fallisce, si è fatto in modo che tutti i ***controlthread*** istanziati, vengano messi all'interno di un unico ***ThreadGroup*** e che di questo gruppo venga continuamente controllato lo stato.

Un ***controlthread*** normalmente in esecuzione è in stato di "RUNNABLE", ma quando l'interprete remoto associato finisce la computazione, il controlthread passa in stato di "WAITING" aspettando il momento per poter terminare.

Quando invece, un thread fallisce, finisce in stato di "TERMINATED" e il ***manager*** può gestirlo.

Quando viene catturata la ***RemoteException*** all'interno del ***controlthread***, per il fallimento del ***RemoteInterpreter***, vengono rimessi in coda, nel ***MdfiPool***, i task, non elaborati, assegnati a quel worker.

Il ***manager*** quindi riassegnerà quei task fra gli altri ***RemoteInterpreter*** rimasti, e incrementerà la variabile ***blocked***.

Questa variabile è utilizzata dal ***manager*** per capire quando terminare i ***controlthread***.

Infatti essa viene incrementata sia, quando i ***controlthread*** aspettano perché il ***RemoteInterpreter*** associato, ha finito di eseguire il calcolo assegnato, sia quando i ***RemoteInterpreter*** per qualsiasi motivo generano una ***RemoteException***.

Il ***manager*** utilizza ***InputManager*** come interfaccia per implementare le classi che si occuperanno di gestire l'input stream del programma. Tale interfaccia ha due metodi ***next*** e ***hasNext***.

Il primo viene utilizzato dal ***manager*** per ottenere il task successivo da calcolare, mentre il secondo, viene utilizzato per controllare se ci sono altri ***task*** disponibile all'interno dell'***input stream***.

Le classi che implementano questa interfaccia sono: ***IntegerStreamInputManager***, un semplice input stream, che restituisce un numero di oggetti interi quando il metodo ***next*** viene invocato e ***IntegerVectorStreamInputManager***, che restituisce un Vector di interi (delle dimensioni specificate) quando il metodo ***next*** viene invocato.

Esistono inoltre, due interfacce che estendono la classe ***InputManager***.

La prima ***BlockingInputManager*** fornisce i metodi ***next*** e ***hasNext*** bloccanti. Questa classe permette agli oggetti ***SECompute*** di aggiungere task.

La classe ***SECompute*** estende la classe ***Compute*** introducendo "Side Effect", cioè con la capacità di generare un nuovo task da calcolare.

La sottoclasse ***SECompute*** implementa il solo metodo ***compute*** all'interno del quale imposta il ***task*** generato con il metodo ***setGeneratedTask***.

E' fondamentale che la generazione dei ***task*** sia condizionata dall'input del metodo ***compute***, altrimenti si creeranno dei loop che porteranno alla generazione continua di ***task***.

La classe ***BlockingInputManager*** fornisce i seguenti metodi:

- ***addTask(Object Task)***, per aggiunge un oggetto ***Task*** allo stream.
- ***addProducers(int numSECompute)*** per segnalare allo stream la presenza di potenziali nuovi produttori pari a numSECompute.
- ***addProducer()*** che segnala allo stream la presenza di un potenziale nuovo produttore.
- ***removeProducer()*** che segnala allo stream che un produttore ha smesso di produrre.

La seconda interfaccia che estende ***InputManager*** è ***NonBlockingInputManager,*** che dovrà essere implementata dagli ***InputManager*** che forniscono metodi ***next*** e ***hasNext*** non bloccanti.

La classe che implementa l'interfaccia ***NonBlockingInputManager*** è ***DynamicInputManager***.

Questa classe costituisce un "wrapper" per l'interfaccia ***NonBlockingInputManager*** che essendo statico non permette l'aggiunta di elementi da produrre a runtime.

Il ***DynamicInputManager*** consuma, durante la sua creazione, un oggetto ***NonBlockingInputManager*** (che dovrà contenere almeno un elemento) passato come parametro.

La classe ***DynamicInputManager***, fornisce i seguenti metodi:

- ***hasNext()***, restituisce true se ci sono elementi pronti da consumare e si blocca se non ci sono elementi pronti ma sono presenti potenziali produttori. Qualora i produttori non aggiungano nessuno elemento durante l'attesa, il metodo restituisce false.

- ***Next()***, restituisce il task successivo e se la coda è vuota aspetta.
- ***addTask(Object Task)***, per aggiunge un oggetto ***Task*** allo stream,
- ***addProducers(int n)*** per segnalare allo stream la presenza di potenziali nuovi produttori pari a n.
- ***addProducer()*** che segnala allo stream la presenza di un potenziale nuovo produttore
- ***removeProducer()*** che segnala allo stream che un produttore ha smesso di produrre.

La classe ***DynamicInputManager*** permette inoltre ai produttori di inserire nello stream nuovi elementi. I produttori devono segnalare allo stream sia la volontà di iniziare a produrre che quella di smettere.

Una volta che viene reinserita l'istruzione all'interno dell'***MdfiPool*** a causa del fallimento di un nodo remoto, se lo stream di input supporta i produttori, e quindi fa parte di una delle classi citate, e se l'istruzione stessa è un produttore, allora viene segnalato allo stream la perdita di un produttore richiamando il metodo ***removeProducer*** sull'oggetto ***inputstream***. Viene quindi fatto terminare il ***controlthread***.

Il codice per il trattamento della ***RemoteException*** dovuta al fallimento di un ***RemoteInterpreter***, con reinserimento del task in coda nell'***MdfiPool*** è il seguente:

```
 try { // compute the instruction remotely
       res = worker.compute(instr);
} catch (RemoteException e) {

       // first: put back the not computed mdf instruction
       pool.insert(instr);
       System.out.println("PUT BACK THE NOT COMPUTED MDF INSTRUCTION");

       // eventually terminate the thread
       System.out.println("remote worker terminated due RemoteException on mdfi
       compute request");
       //se lo stream supporta i produttori e se l'instr e' un produttore
       if(blockingInStream && instr.isSeMDFi()){
             //segnalo allo stream la perdita di un produttore
             ((BlockingInputManager) ism).removeProducer();
             System.out.println("Rimosso Produttore");
       }
e.printStackTrace();
return; // this terminates the thread actually
```

}

L'oggetto ***pool*** della classe ***MdfiPool*** viene creato nel ***manager*** e viene condiviso con i ***controlthread***.

Sull'oggetto pool è possibile richiamare i metodi:

- ***insert (Mdfi istr)*** per inserire una istruzione MDFi nel ***pool***
- ***insert (Mdf graph)*** per inserire un grafo MDF nel ***pool***
- ***extract (IstructionTag instrtag)*** per estrarre l'istruzione Mdfi corrispondente ad un dato ***instructioTag***
- ***extract (InstructionTag instrtag, GraphId graph)*** per estrarre l'istruzione Mdfi corrispondente ad un dato ***InstructionTag*** e ad un dato ***GraphId***.

Il ***manager*** quando crea un ***controlthread*** per ogni ***RemoteInterpreter***, trovato con il ***DiscoveryService***, gli passa tra i parametri di input anche un riferimento all'oggetto ***pool***.

Il ***controlthread*** potrà quindi utilizzarlo, nel caso citato, per reinserire i ***task*** non eseguiti dal worker fallito, richiamando il metodo ***insert*** su tale oggetto.

A questo punto il ***controlthread*** termina ed è compito del ***manager*** gestirlo.

All'interno del metodo ***compute***, il ***manager*** controlla lo stato dei ***controlthread*** precedentemente istanziati ed eseguiti.

Tramite una variabile ***blocked***, viene tenuto sotto controllo lo stato dei ***controlthread***..

La variabile viene incrementata sia, quando un ***controlthread*** passa in stato di "WAITING" perché il worker associato ha finito la computazione e aspetta di terminare, sia quando un ***controlthread*** passa in stato di "TERMINATED" perché il nodo remoto associato è fallito.

Quando la variabile **blocked** raggiunge il valore del numero di macchine istanziate all'inizio per il parallelismo, o per l'implementazione del Best Effort, tutti i ***controlthread*** vengono fatti terminare.

Il codice per il controllo dello stato dei ***controlthread*** e per la gestione della variabile ***blocked*** è il seguente.

```
while(((blocked != machines.length))) {

	try {
		Thread.sleep(100);
	} catch (InterruptedException e) {
		e.printStackTrace();
	}  // just for releasing the CPU

	//System.out.println("ACTIVE COUNT "+thGr.activeCount());
	blocked = 0; // added to avoid accumulation ...

	for(int i=0;i<machines.length; i++) {
	//    System.out.println(">>> "+controlThreads[i]+"
	"+controlThreads[i].getState()+")");
		if(((controlThreads[i].getState() == Thread.State.WAITING ) ||
		(controlThreads[i].getState() == Thread.State.TERMINATED ))) {
			blocked++;
		}
	}
	//System.out.println("La variabile blocked vale: " + blocked);

	if(blocked == machines.length) {
		// terminate threads
		//TODO after gathering statistics ...
	for(int i=0; i<machines.length; i++)
		controlThreads[i].interrupt();
	}
}
```

3.4

IMPLEMENTAZIONE SERVER RMI CON SSL

Nell'implementazione della parte Server per la comunicazione con host sicuri e non, è stato modificato il codice della classe ***RemoteInterpreter***. E' stata introdotta un variabile booleana ***Sicuro***, che di default viene impostata a ***false***.

E' stato aggiunto un costruttore per l'implementazione della comunicazione con SSL, dove al momento della creazione dell'istanza del ***RemoteInterpreter***, sono richiesti come parametri: un intero per la ***porta*** da utilizzare, e due oggetti, uno della classe ***SslRMIServerSocketFactory,*** e uno della classe ***SslRMIClientSocketFactory***.

Nel metodo principale ***main***, viene valutata la variabile ***Sicuro***, e se questa è impostata a ***true***, l'esecuzione prosegue utilizzando il protocollo RMI standard, altrimenti se è impostata a ***false***, l'esecuzione prosegue istanziando prima un

oggetto ***SslRMIServerSocketFactory***, per la parte relativa al Server, poi un oggetto ***SslRMIClientSocketFactory***, per la parte relativa al Client.

Quindi viene istanziato un oggetto della classe ***RemoteInterpreterInterface*** richiamando il costruttore precedentemente aggiunto e passandogli opportunamente gli input richiesti, precedentemente istanziati.

A questo punto viene creato un oggetto della classe ***Registry*** basato su SSL passandogli come parametri, una porta e due istanze per gli oggetti ***SslRMIClientSocketFactory*** e ***SslRMIServerSocketFactory***.

Quindi viene registrato l'oggetto nel registry RMI con il ***rebind***.

La signature per il costruttore aggiunto a quelli già presenti è la seguente:

```
protected RemoteInterpreter(int Port, SslRMIServerSocketFactory ServerSocket,
            SslRMIClientSocketFactory ClientSocket ) throws RemoteException {
```

Il controllo sulla variabile ***Sicuro*** e l'implementazione della comunicazione per la parte Server, con e senza SSL, è mostrata nel seguente codice:

```
if (Sicuro){

      RemoteInterpreterInterface worker = new RemoteInterpreter();
      System.out.println("Worker created ..");
      Registry reg = null;
      try {
            reg = LocateRegistry.createRegistry(port); // or I'm the first one
            // running on this
            // server
      } catch (RemoteException e) {
            e.printStackTrace();
            System.exit(-1);//port already in use, exit.
      }

      System.out.println("Register created on port " + port);
      // now register the object
      try {
            String workerName = "muSkelMDFWorker";
            reg.rebind(workerName, (Remote) worker);
            System.out.println(worker);
      } catch (AccessException e1) {
            e1.printStackTrace();
      } catch (RemoteException e1) {
            e1.printStackTrace();
      }

}else{
```

```
        SslRMIServerSocketFactory ssf = new SslRMIServerSocketFactory();
        SslRMIClientSocketFactory csf = new SslRMIClientSocketFactory();
        RemoteInterpreterInterface worker = new RemoteInterpreter(port,ssf,csf);
        System.out.println("Worker with SSL created ..");
        Registry reg = null;
        try {
              // Create SSL-based registry
              reg = LocateRegistry.createRegistry(port,
                    new  javax.rmi.ssl.SslRMIClientSocketFactory(),
                    new javax.rmi.ssl.SslRMIServerSocketFactory());
                    // or I'm the first one
                    // running on this
                    // server
        } catch (RemoteException e) {
              e.printStackTrace();
              System.exit(-1);//port already in use, exit.
        }
        System.out.println("Register created on port " + port);
        // now register the object
        try {
              String workerName = "muSkelMDFWorker";
              reg.rebind(workerName, (Remote) worker);
              System.out.println(worker);
        } catch (AccessException e1) {
              e1.printStackTrace();
        } catch (RemoteException e1) {
              e1.printStackTrace();
        }catch (Exception e1){
              System.out.println("RemoteInterpreter err: " + e1.getMessage());
              e1.printStackTrace();
        }
}
```

I ***RemoteInterpreter*** devono essere eseguiti manualmente, ad esempio tramite comandi ssh, oppure tramite appositi script forniti con l'ambiente Muskel.

Quando viene lanciato il ***RemoteInterpreter*** è possibile passargli come argomento il parametro ***sicuro,*** da linea di comando per caricare un ***RemoteInterpreter*** che non utilizzi SSL, mentre di default viene impostato il worker con SSL.

Per l'esecuzione del RemoteInterpreter con SSL devono essere fornite da linea di comando le properties:

-Djavax.net.ssl.keyStore=./keystore.jks (per settare il percorso del ***keystore***)

-Djavax.net.ssl.keyStorePassword=muskel (per la password del ***keystore***)

Una descrizione di cosa è il ***keystore*** e come viene generato è discussa al paragrafo 3.6.

3.5

IMPLEMENTAZIONE CLIENT RMI CON SSL

Nell'implementazione della parte Client per la comunicazione con host sicuri e non, è stata modificata la classe ***ControlThread***..

Quando il ***Manager*** crea le istanze di un ***RemoteInterpreter*** che gestisce uno degli interpreti remoti, passa al ***ControlThread*** una variabile booleana ***Sicuro***, che dice a quest'ultimo come implementare la comunicazione con l'host su cui gira l'interprete remoto.

Se il valore della variabile ***Sicuro*** è impostata a ***true*** la comunicazione avviene secondo il protocollo RMI standard.

Altrimenti viene creato un riferimento all'oggetto ***Registry*** basato su SSL, tramite il metodo ***getRegistry*** della classe ***LocateRegistry***, passando come parametri: l'indirizzo dell'host remoto, la ***porta*** sulla quale gira il ***registry*** e una istanza della classe ***SslRMIClientSocketFactory***.

Viene quindi creato l'oggetto ***worker***, precedentemente istanziato, che servirà per riferire l'oggetto remoto che implementa ***RemoteInterpreterInterface***.

Se la variabile ***Sicuro*** è impostata a ***true,*** la comunicazione verrà stabilita utilizzando il protocollo RMI standard, e vorrà dire che il nodo remoto sul quale gira il ***RemoteInterpreter*** è presente nella lista degli IP sicuri contenuta nel file ***hosts.conf***, caricata dal ***manager*** all'inizio della computazione.

Una volta ottenuto l'***host*** e la ***porta*** sulla quale connettersi, il ***controlthread*** creerà un oggetto ***String*** con la sintassi per la comunicazione con rmi (*rmi://host:port/muSkelMDFWorker*).

Tale oggetto ***String*** sarà passato al metodo ***lookup*** su un oggetto della classe ***java.rmi.Naming***, che restituirà un riferimento all'oggetto remoto associato con quel nome simbolico (*muSkelMDFWorker*).

Se invece, ***Sicuro*** è impostato a ***false*** la comunicazione avviene utilizzando ***SslRMISocketFactory***.

Verrà creato un oggetto ***reg*** della classe ***java.rmi.registry.Registry***, che attraverso il metodo ***getRegistry*** restituirà un riferimento locale all'oggetto remoto ***Registry*** specificando l'indirizzo dell'***host*** remoto e la ***porta.***

Per comunicare con il ***registry*** remoto verrà usato l'oggetto ***SslRMIClientSocketFactory*** fornito, creando le connessioni SSL Socket con il ***registry*** sull' ***host*** remoto e sulla ***porta*** specificata.

In questo caso la ***lookup*** verrà richiamata sull'oggetto ***reg*** creato, passondagli soltanto il nome simbolico del servizio (*muSkelMDFWorker*).

Il codice per la parte client nella comunicazione con il server, utilizzando RMI, con e senza SSL, è mostrato di seguito.

```
// set up the remote node connection
RemoteInterpreterInterface worker = null;

System.out.println("Control thread started");

if(Sicuro){ //Host Sicuro non uso SSL

	try {
		ciccio = InetAddress.getByName(host);
		System.out.println("Got address " + ciccio + " for " + host);
		String stringa = "rmi://" + host + ":" + RemoteInterpreter.Port
				+ "/muSkelMDFWorker";
		if (debug)
			System.out.println("Looking up worker on host " + stringa);

		worker = (RemoteInterpreterInterface) Naming.lookup(stringa);
			System.out.println("worker is " + worker);
			System.out.println("Remote worker looked'up");
		if (debug)
			System.out.println("worker is " + worker);
	} catch (MalformedURLException e) {
		e.printStackTrace();
	} catch (RemoteException e) {
		e.printStackTrace();
	} catch (NotBoundException e) {
		e.printStackTrace();
	} catch (UnknownHostException e) {
		e.printStackTrace();
	}

}else{ //Host non sicuro uso SSL

	try {
		// Make reference to SSL-based registry
		Registry reg = LocateRegistry.getRegistry(
			host, Port, new javax.rmi.ssl.SslRMIClientSocketFactory());

		ciccio = InetAddress.getByName(host);
		System.out.println("Got address " + ciccio + " for " + host);
```

```
            System.out.println("Looking up worker on host " + host);
            if (debug)
                  System.out.println("Looking up worker on host " + host);

            // "worker" is the identifier that we'll use to refer
            // to the remote object that implements the "RemoteInterpreter"
            // interface
            worker = (RemoteInterpreterInterface) reg.lookup("muSkelMDFWorker");
            System.out.println("worker is " + worker);
            System.out.println("Remote worker looked'up");
            if (debug)
                  System.out.println("worker is " + worker);
      } catch (RemoteException e) {
            e.printStackTrace();
            return;
      } catch (NotBoundException e) {
            e.printStackTrace();
      } catch (UnknownHostException e) {
            e.printStackTrace();
      }
}
```

Per l'esecuzione del codice del Client con SSL devono essere fornite da linea di comando le properties:

-Djavax.net.ssl.trustStore=./trustore.jks (per settare il percorso del ***truststore***)

-Djavax.net.ssl.trustStorePassword=muskel ***(***per la password del ***truststore)***.

Verrà adesso descritto cosa sono e come vengono generati il ***keystore*** e il ***trustore*** attraverso il ***keytool***.

3.6

GENERAZIONE TRUSTSTORE E KEYSTORE CON KEYTOOL

Keytool [35] è una utility, contenuta all'interno del Sun JDK [36], per la gestione delle chiavi e dei certificati.

Permette all'utente di amministrare la propria coppia di chiavi pubbliche e private e i certificati associati per l'auto-autenticazione (dove l'utente autentica se stesso con altri utenti o servizi) o per l'integrità dei dati e l'autenticazione dei servizi, usando le firme digitali.

E’ inoltre permesso all’utente, di salvare in cache le chiavi pubbliche (in forma di certificati) dei propri partner di comunicazione. Il tutto è firmato da una terza parte fidata.

Un **certificato digitale** è un documento elettronico che attesta, con una firma digitale, l'associazione tra una chiave pubblica e l'identità di un soggetto (una persona, una società, un computer, etc).

Lo scopo del certificato digitale è quello di garantire che una chiave pubblica sia associata alla vera identità del soggetto che la rivendica come propria.

In un sistema a crittografia asimmetrica ciò può essere molto importante: infatti, ogni messaggio crittografato con una data chiave pubblica può essere decrittato solo da chi possiede la relativa chiave privata; per cui, se siamo sicuri che la chiave pubblica appartiene a "Mario Rossi" allora siamo anche sicuri che solo "Mario Rossi" potrà leggere i messaggi crittati con quella chiave pubblica.

Inoltre vale anche il viceversa: se possiamo decriptare un messaggio con quella chiave pubblica allora siamo sicuri che quel messaggio è stato criptato da "Mario Rossi" (anche se ciò non implica che quel messaggio è stato inviato da "Mario Rossi").

Nello schema tipico di un'infrastruttura a chiave pubblica (PKI), la firma apposta sarà quella di una autorità di certificazione (CA). Nella rete di fiducia ("web of trust"), invece, la firma è quella o dello stesso utente (un'autocertificazione) oppure di altri utenti ("endorsements").

In entrambi i casi, la firma certifica che la chiave pubblica appartiene al soggetto descritto dalle informazioni presenti sul certificato (nome, cognome, indirizzo abitazione, indirizzo IP, etc.)

I certificati sono utili per la crittografia a chiave pubblica quando usata su larga scala.

Infatti, scambiare la chiave pubblica in modo sicuro tra gli utenti diventa impraticabile, se non impossibile, quando il numero di utenti comincia a crescere. I certificati digitali sono una soluzione per superare questa difficoltà.

In principio se Mario Rossi voleva inviare/ricevere un messaggio segreto, doveva solo divulgare la sua chiave pubblica. Ogni persona che la possedeva poteva inviargli e/o ricevere da lui messaggi sicuri; tuttavia qualsiasi individuo poteva divulgare una

differente chiave pubblica (di cui conosceva la relativa chiave privata) e dichiarare che era la chiave pubblica di Mario Rossi.

Per ovviare a questo problema, Mario Rossi inserisce la sua chiave pubblica in un certificato firmato da una terza parte fidata ("trusted third party"): tutti quelli che riconoscono questa terza parte devono semplicemente controllarne la firma per decidere se la chiave pubblica appartiene veramente a Mario Rossi.

In una PKI (Public Key Infrastructure), la terza parte fidata sarà un'autorità di certificazione. Nel web of trust, invece, la terza parte può essere un utente qualsiasi e sarà compito di chi vuole comunicare con Mario Rossi decidere se questa terza parte è abbastanza fidata.

Un certificato solitamente ha un intervallo temporale di validità, al di fuori del quale deve essere considerato non valido.

Un certificato può essere revocato se si scopre che la relativa chiave privata è stata compromessa, oppure se la relazione specificata nello stesso (cioè la relazione tra un soggetto ed una chiave pubblica) è incorretta o è cambiata; questo potrebbe succedere se, per esempio, una persona cambia lavoro oppure indirizzo.

Ciò significa che un utente oltre a controllare che il certificato sia fidato (verificare che sia firmato da una CA riconosciuta) e non sia scaduto dovrebbe controllare anche che non sia stato revocato.

Questo può essere fatto attraverso la lista dei certificati revocati (CRL). Una funzione chiave della PKI (Public Key Infrastructure) è proprio quella di tenere aggiornata la CRL.

Un altro modo per verificare la validità di un certificato è quello di interrogare la CA attraverso un protocollo specifico come, per esempio, Online Certificate Status Protocol (OCSP).

Un certificato tipicamente include:

* una chiave pubblica;

* dei dati identificativi, che possono riferirsi ad una persona, un computer o un'organizzazione;

* un periodo di validità;

* l'URL della lista dei certificati revocati (CRL);

Il tutto è firmato da una terza parte fidata.

Keytool, contenuto nel JDK, memorizza le chiavi e i certificati all'interno di un cosiddetto ***keystore***. L'implementazione standard del ***keystore*** è quella all'interno di un file, e le chiavi sono protette da una password.

Ci sono due tipi differenti ***entries*** in un ***keystore***:

- ***Key entries*** – relative ad ogni informazione sensibile sulle chiavi crittografiche, che sono memorizzate in un formato protetto per prevenire accessi non autorizzati. Tipicamente, una chiave memorizzata in questo tipo di ***entries*** è una chiave segreta, o chiave privata accompagnata da un certificato per la corrispondente chiave pubblica. Il ***keytool*** gestisce soltanto il secondo tipo di ***entry***, che sono chiavi private con il loro certificato associato.

- ***Trusted certificate entries*** – detiene un singolo certificato a chiave pubblica di proprietà di qualcun altro. Viene chiamato "***trusted certificate***" perché il proprietario del ***keystore***, garantisce che la chiave pubblica nel certificato appartiene effettivamente all'entità identificata dal "***subject***" (proprietario) nel certificato. Chi emette il certificato, garantisce per esso, firmando il certificato.

Per generare una coppia di chiavi in un ***keystore*** il comando è:

keytool –genkey [opz] –alias nome –validity giorni –keystore keystore.jks

Il tool richiede alcune informazioni sull'identità della persona che genera le chiavi, che saranno memorizzate all'interno delle chiavi stesse.

Il ***keystore*** è protetto da password.

E' possibile visualizzare il contenuto di un keystore mediante il comando:

keytool –list –v –keystore keystore.jks

In genere il processo per generare un certificato richiede tre passi.

- Creazione di una "***Certificate Request***" a partire dalla chiave pubblica nel keystore.

- Invio della "***Certificate Request***" alla CA, che produce il certificato.

- Importazione del certificato nel ***truststore***.

Non avendo una CA (***Certification Authority***) genereremo un **self-signed certificate**. Non c'è garanzia sull'identità data dalla CA. E' adeguato se i due end-point si fidano reciprocamente e possono scambiarsi i certificati in maniera sicura.

I passi diventano: generazione del certificato a partire dalla chiave pubblica nel ***keystore*** e importazione del certificato nel ***truststore***.

Per generare il certificato:

keytool -export -alias nome -keystore keystore.jks -rfc -file muskel.cert

Per importare il certificato:

keytool -import-alias nome -keystore trustore.jks -file muskel.cert

Il ***keystore*** creato e utilizzato per questa tesi nelle esecuzione dell'ambiente parallelo ***muskel,*** con l'introduzione delle nuove feature compresa quella per l'implementazione della sicurezza con SSL, è il seguente:

```
Immettere la password del keystore:

Tipo keystore: JKS
Provider keystore: SUN

Il keystore contiene 1 entry

Nome alias: muskel
Data di creazione: 1-mag-2010
Tipo di voce: PrivateKeyEntry
Lunghezza catena certificati: 1
Certificato[1]:
Proprietario: CN=Daniele Grotti, OU=muskel, O=muskel, L=Pisa, ST=Pisa, C=IT
Autoritó emittente: CN=Daniele Grotti, OU=muskel, O=muskel, L=Pisa, ST=Pisa, C=I
T
Numero di serie: 4bdc4224
Valido da: Sat May 01 17:00:52 CEST 2010 a: Sun May 01 17:00:52 CEST 2011
Impronte digitali certificato:
         MD5:  AD:5C:8B:55:24:64:B9:C8:69:4C:92:63:89:4F:8D:4D
         SHA1: C7:81:65:57:42:CE:18:48:FC:64:0F:0D:59:AF:F4:56:DF:45:3B:FA
         Nome algoritmo firma: SHA1withDSA
         Versione: 3


*******************************************
*******************************************
```

Capitolo 4

4.1

VALUTAZIONI

Verranno adesso discussi i risultati ottentuti nei test effettuati presso il Laboratorio H del Dipartimento di Informatica all'Università di Pisa, facendo anche delle considerazioni a livello hardware sul costo della rete utilizzata.

Gli esperimenti sono stati effettuati su un Cluster di macchine Pentium III con sistema operativo Linux (kernel 2.6.26) collegate con Fast Ethernet ed equipaggiate con Java VM versione 1.6.0_14.

Le applicazioni utilizzate per i test sono applicazioni "sintetiche", che non effettuano grossi calcoli ma cercano di stressare tutte le parti della nuova versione di muskel presentata in questa tesi.

Gli esperimenti sono stati eseguiti utilizzando la classe ***SampleCode*** come codice di esempio per creare una applicazione parallela sfruttando le nuove feature.

Tale classe, una volta acquisito, da riga di comando, il grado di parallelismo richiesto, salvato nella variabile ***pardegree*** (di default pardegree vale 1), crea quattro oggetti che estendono la classe ***Compute***, per comporre la computazione parallela di un ***Pipeline*** a due stadi.

Il primo stadio del ***Pipeline*** è composto da un ***Farm***, che prende come input un oggetto della classe **Square**, e il secondo stadio del pipeline è un oggetto della classe ***Inc***.

La classe ***Square*** calcola il quadrato di un numero mentre la classe ***Inc*** prende un intero e lo incrementa.

A questo punto vengono specificati l'input e l'output stream.

L'input stream è un oggetto della classe ***IntegerStreamInputManager*** che prende come parametro, un intero che rappresenta il numero di iterazioni da compiere (numero di ***task***), mentre l'output stream è un oggetto della classe **ConsoleOutputManager** che visualizza i risultati sulla "console".

Viene quindi creato un oggetto della classe ***Manager*** passandogli il pipeline creato, insieme all'input e all'output stream, e viene settato il contratto per le performance al valore contenuto nella variabile ***pardegree***.

Sul ***manager*** viene quindi chiamato il metodo ***compute*** per far partire la computazione.

I test sono stati effettuati configurando una diversa ***grana*** del calcolo (rapporto tra il tempo speso nella computazione di una istruzione MDF e il tempo speso nella consegna dell' input token e nel recupero dell'output token), da un valore minimo di 10 ad un valore massimo di 300 ***task*** da calcolare.

Un dump dell'esecuzione del programma su 3 macchine è mostrato nella Figura 4.1

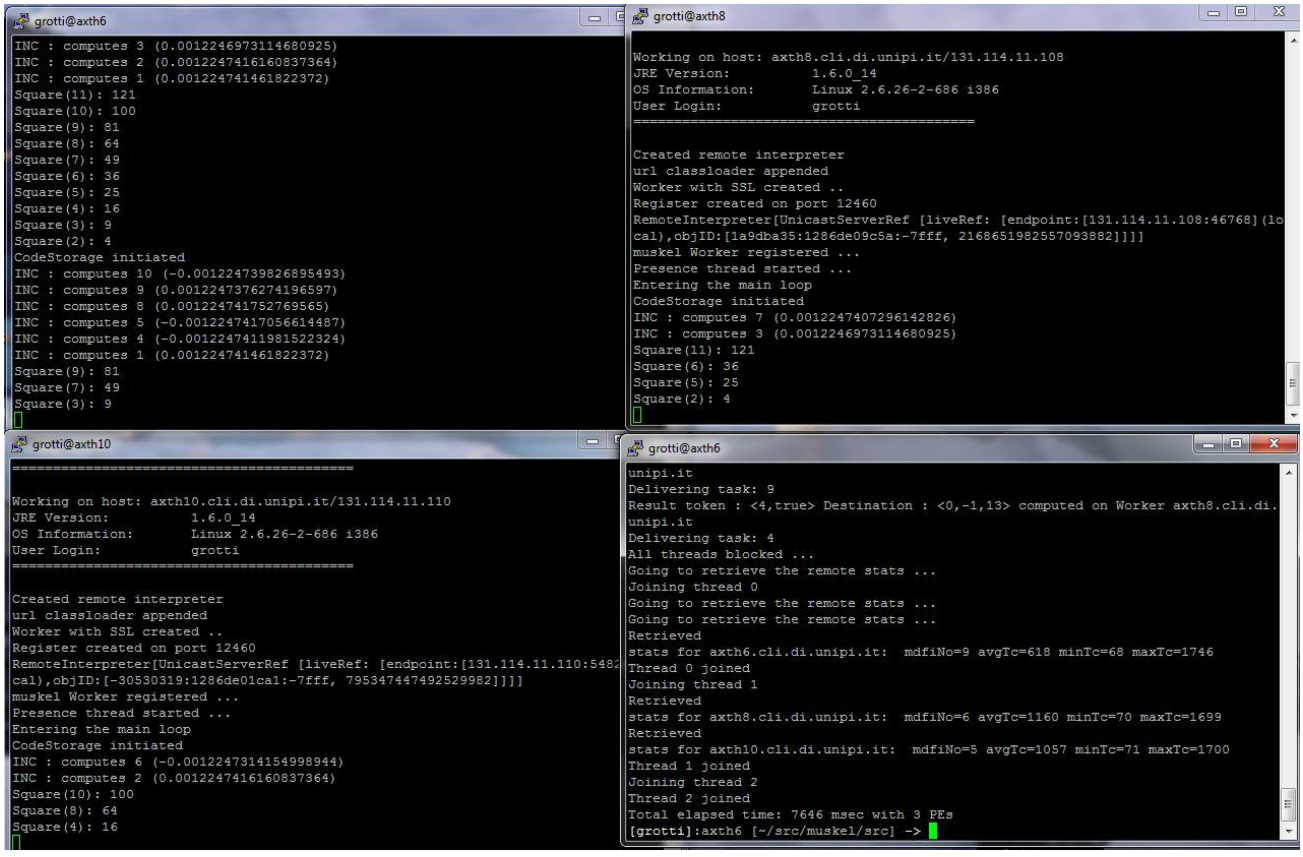


Figura 4.1

I test sono stati eseguiti su un numero di macchine (PEs) compreso tra 1 e 16, sulle quali girava una istanza di ***RemoteInterpreter***, configurati di volta in volta o come Worker standard o come Worker SSL.

Le prove sono state effettuate impostando percentuali diverse per l'implementazione dell'SSL nella comunicazione con i vari worker.

Prima si è testata la computazione utilizzando su tutte le macchine la comunicazione senza SSL, e poi è stata introdotta una percentuale progressiva di SSL fino ad arrivare all'utilizzo completo di SSL su tutte le macchine a disposizione.

I grafici della **scalabilità** e del **tempo di completamento** del programma, con una diversa grana del calcolo e su un diverso numero di macchine disponibile, sono riportati nei grafici seguenti.

Nel grafico 4.2 viene mostrato il tempo di completamento in millisecondi con 10 Task per la computazione su 8 macchine. E' stato eseguito prima il calcolo senza utilizzare SSL su nessuna delle 8 macchine e poi una percentuale sempre maggiore fino ad arrivare a utilizzarlo su tutte e 8.

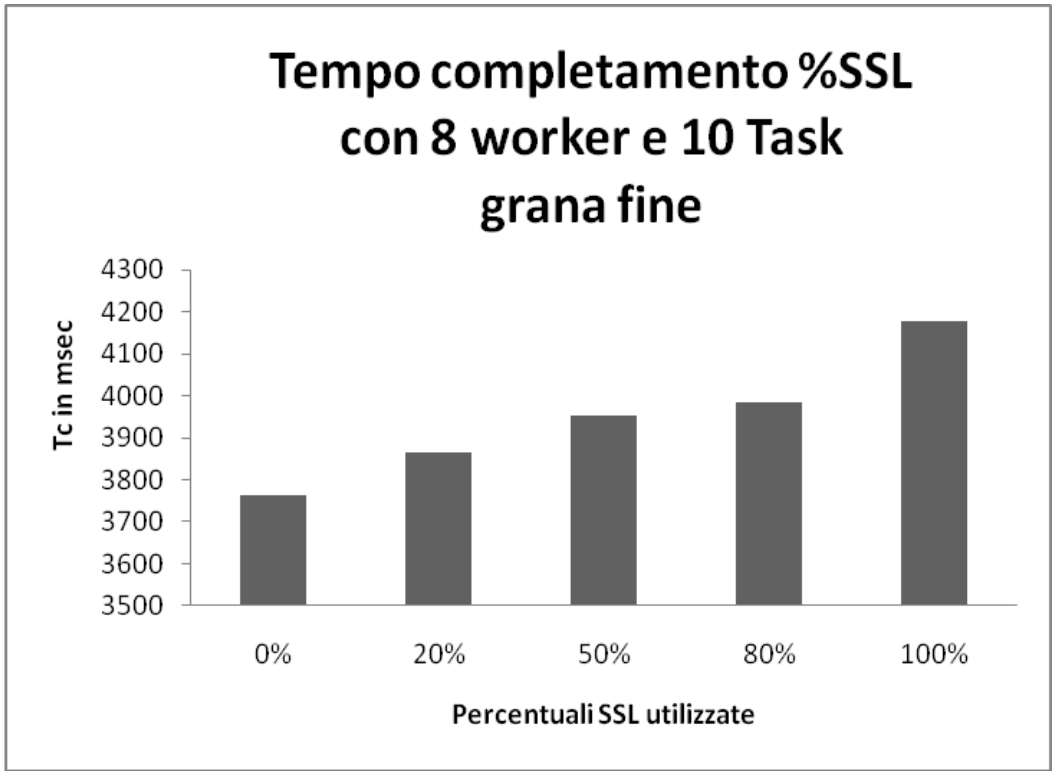


Figura 4.2

Nella Figura 4.3 vengono mostrati i tempi di setup nel caso sopra citato con 8 worker e diverse percentuali di SSL.

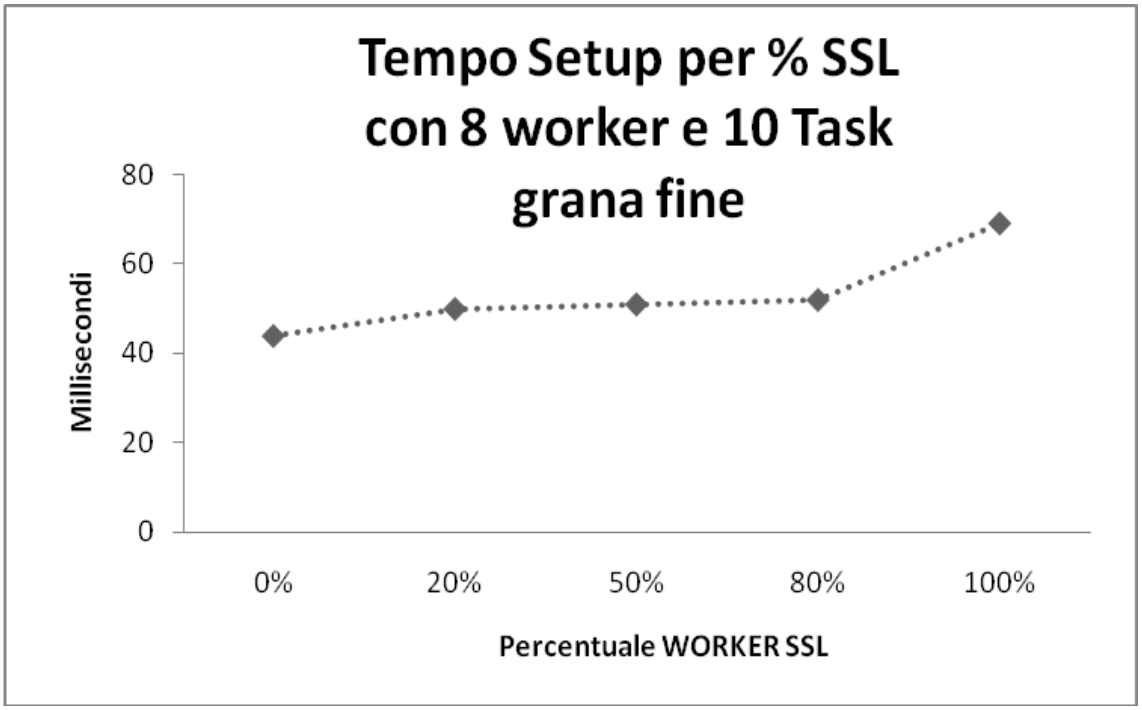


Figura 4.3

Si può vedere come all'aumentare della percentuale di SSL utilizzata per l'implementazione della comunicazione, aumenti anche il tempo di setup comprensivo di inizalizzazione dei controlthread e assegnazione dei task.

In particolare possiamo notare che, per un numero piccolo di 10 Task, il tempo di setup pagato nel caso di utilizzo di SSL su tutti i worker è quasi il doppio di quello pagato senza utilizzare SSL su nessuna macchina, fra le 8 disponibili.

In questo caso il tempo di setup è costituito principalmente dal costo per l'inizializzazione dei controlthread per più del 70% e per il restante 30% è tempo speso nell'assegnamento dei task.

Nella Figura 4.4 (e 4.5) viene mostrato lo stesso esperimento ma con una grana più grossa di 100 Task.

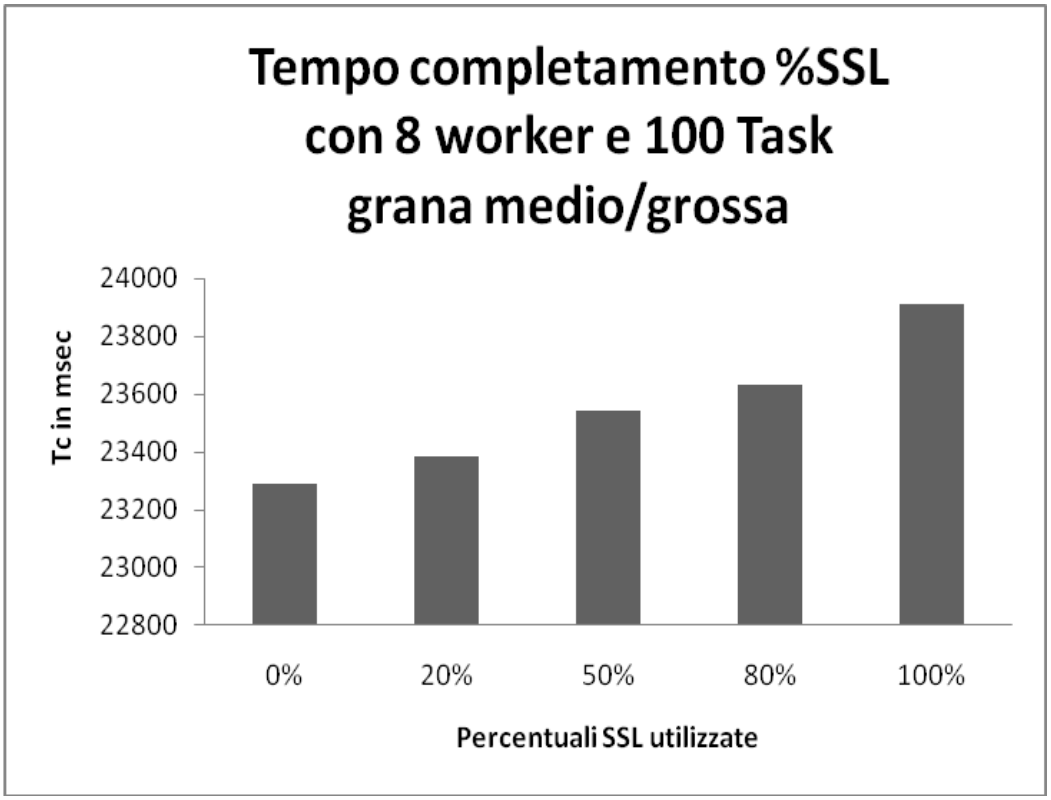


Figura 4.4

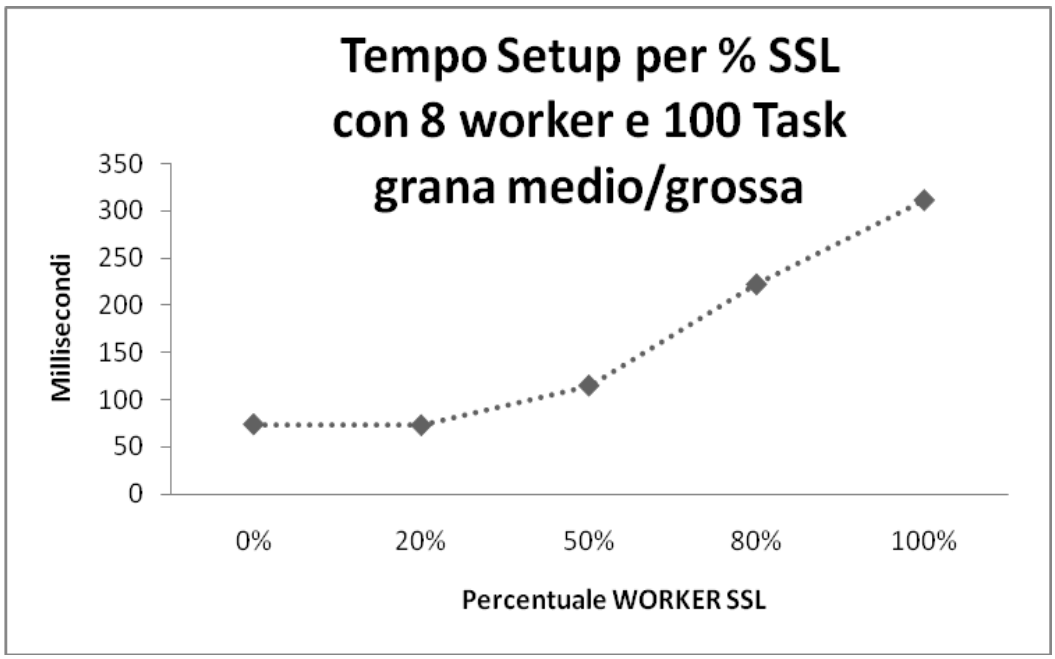


Figura 4.5

La differenza tra i tempi di setup con l'utilizzo o meno di SSL, come immaginavamo, cresce notevolmente rispetto al caso con solo 10 task, inoltre in questo caso il tempo di setup è quasi suddiviso equamente.

Infatti per il 55% è dovuto all'inizializzazione dei controlthread e per il 45% all'assegnamento dei task e come ci aspettavamo, dato il maggior numero di task, viene pagato un costo maggiore per la cifratura e decifratura di codice e dati.

Nella Figura 4.6 (e 4.7) vengono mostrati la scalabilità e il tempo di completamento con 16 macchine (PEs) per una computazione con 100 Task con diverse percentuali di utilizzo di SSL per la comunicazione tra le macchine disponibili.

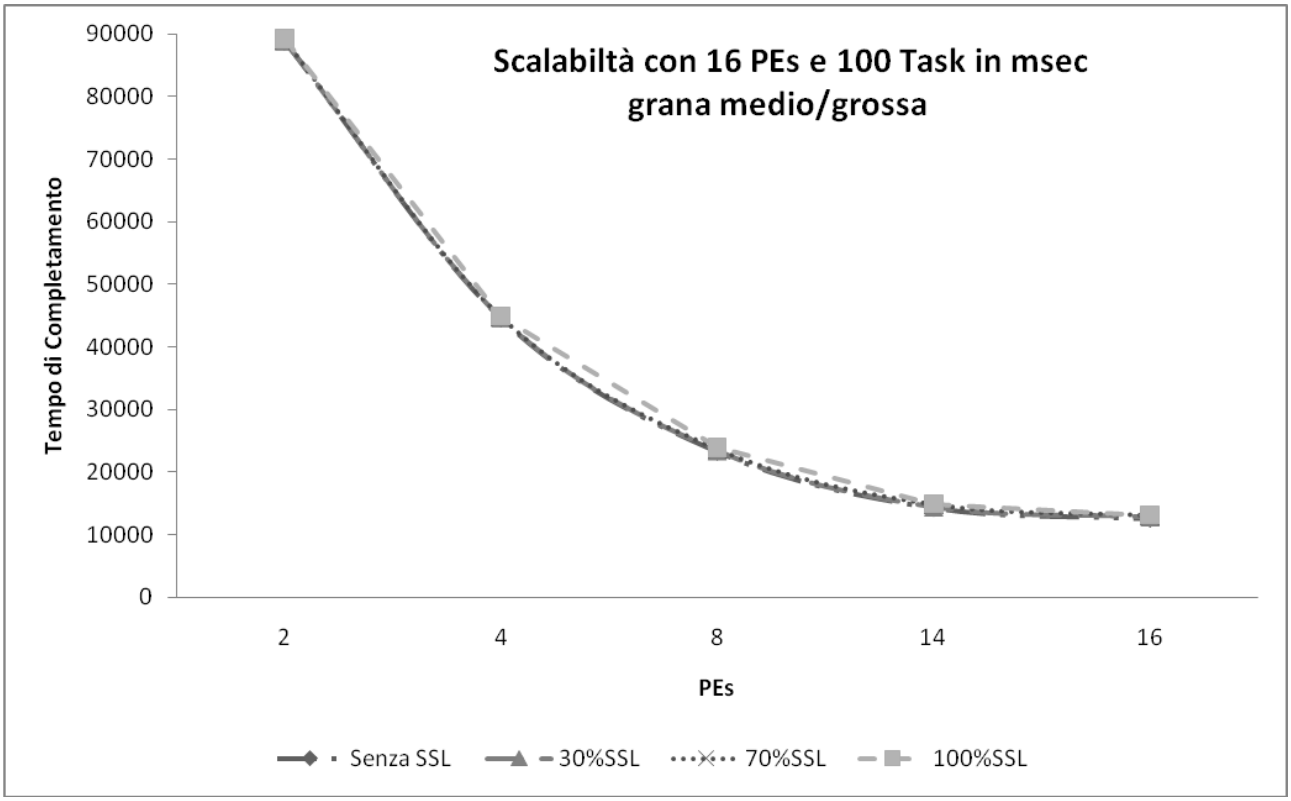


Figura 4.6

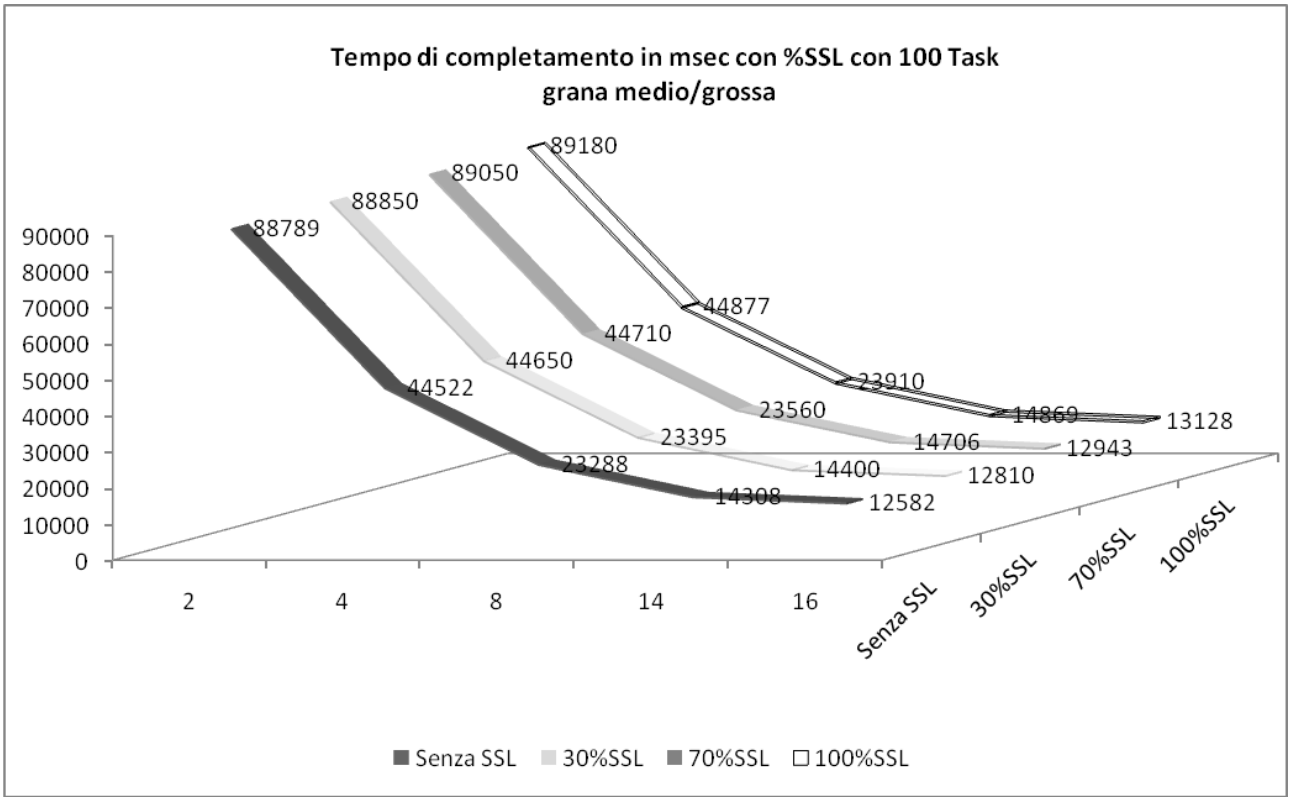


Figura 4.7

Come si può vedere dai grafici, con un utilizzo sia totale che parziale di interpreti remoti che utilizzano SSL per la comunicazione, si riesce a raggiungere una ottima **scalabilità**.

Come ci aspettavamo, quando vengono aggiunti worker che utilizzano una percentuale maggiore di SSL, cioè maggiori interpreti remoti che utilizzano SSL per la comunicazione sicura con il controlthread, il tempo di completamento aumenta.

Quando i worker sono un numero relativamente ristretto, ed è presente un numero medio/piccolo di task, il tempo di inserimento dei task prevale su tempo di inizializzazione dei controlthread nel calcolo del tempo di setup.

Invece quandi worker iniziano ad essere un numero relativamente grande il costo di inizializzazione dei controlthread inizia a prevalere nel costo di setup, rispetto al costo per l'inserimento dei task, sempre per lo stesso numero di task.

Il maggiore tempo di inizializzazione degli **SslRMISocket** (soprattutto per la cifratura e decifratura), rispetto agli RMI standard, in questo caso, pesa maggiormente sul tempo di setup complessivo.

Dai test risulta inoltre che aumentando la grana del calcolo, la scalabilità viene sempre mantenuta.

Quando la grana aumenta, le percentuali sul costo di setup tra, l'inizializzazione e l'inserimento dei task, variano a seconda del numero di worker scelti per la computazione e del numero di worker che utilizzano SSL per la comunicazione, tra quelli disponibili.

La scelta della grana più o meno grossa incide sui valori dove muskel "smette di scalare". Nella Figura 4.6 vediamo come muskel smetta di scalare rapidamente una volta superati i 14 PEs, con 100 task.

Il grafico per l'efficienza relativa con riferimento al tempo di completamento, ad esempio con 100 Task per un numero di worker compreso tra 2 e 16, per diverse percentuali di SSL, è mostrato di seguito nella Figura 4.8.

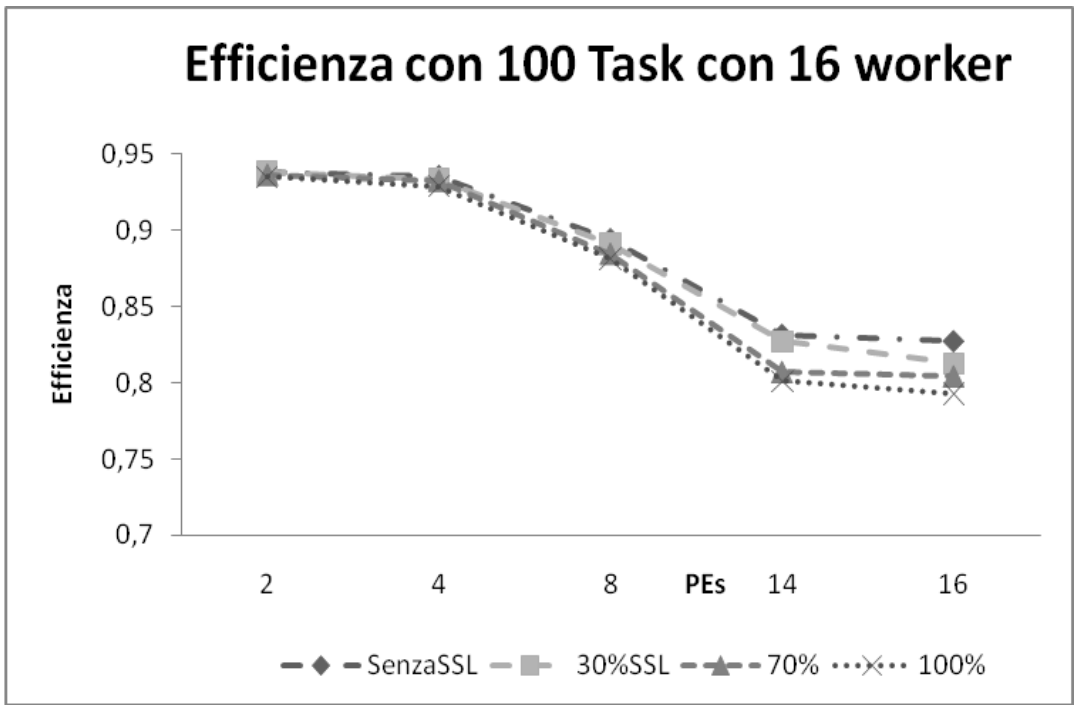


Figura 4.8

Come ci aspettavamo l'efficienza diminuisce quando aumentano il numero di worker disponibili, soprattutto quando viene utilizzato in misura maggiore SSL per le comunicazioni, ma comunque l'efficienza continua ad essere buona.

Anche se in passato l'implementazione degli RMI si è dimostrata poco efficiente [22], sembra che l'implementazione attualmente presente nel JDK 1.6 permetta di raggiungere una buona efficienza e prestazioni ottime nell'esecuzione di programmi

basati su skeleton anche con l'utilizzo di SSL per la comunicazione tra i controlthread e gli interpreti remoti.

Alcune considerazioni vanno fatte anche in merito all'hardware utilizzato per collegare le macchine come viene discusso nel paragrafo 4.2 che segue.

4.2

IMPATTO DELLA RETE

Cerchiamo di misurare la banda delle comunicazioni della versione di muskel, discussa in questa tesi, con e senza sicurezza, per capire un po' meglio quello che succede anche a livello hardware.

La minore banda nel caso dell'utilizzo di SSL, mostrata nei grafici del capitolo 4, è principalmente dovuta all'overhead pagato dal processore nella cifratura e decifratura dei dati durante l'invio e la ricezione.

Questo è parzialmente dovuto allo scambio inziale della chiave nella fase di **handshake**, che viene eseguita una volta per tutte.

Naturalmente le valutazioni avrebbero dovuto tenere conto del tipo di rete usata per gli esperimenti, che in questo caso è stata Fast Ethernet.

L'interfaccia di rete NIC utilizzata non supporta nessun tipo di "*on board data processing*".

In particolare, le schede di rete utilizzate permettono soltanto un supporto hardware per l'acceso via DMA.

Quindi il costo per i messaggi sicuri, che vuol dire costi di cifratura e decifratura dei pacchetti, viene pagato dalla CPU insieme al tempo speso dalla scheda di rete per inviare il messaggio.

Inoltre il costo dei messaggi scambiati all'inizio della fase di negoziazione per la connessione SSL, per stabilire la chiave simmetrica di sessione, viene comunque pagata una volta per tutte al momento dell'inizio della computazione.

Anche le connessioni tra gli ***interpreti remoti*** e i ***controlthread*** vengo stabilite una sola volta e per tutto il tempo in cui, il ***manager*** cercherà di eseguire

codice ***muskel*** parallelo attraverso la chiamata del metodo ***compute***, questo costo non verrà più pagato.

Inoltre questi messaggi addizionali aggiungono un overhead trascurabile nel caso che vengano eseguiti *input stream* molto lunghi.

Nel caso in cui vengano utilizzate altre tipologie di rete, le cose dovrebbero cambiare.

Il particolare, se viene utilizzato dell'hardware di rete moderno ad alte prestazioni, come ad esempio il QsNell, può essere sfruttato il processore montato sulla scheda di rete per eseguire carico utile, ad esempio, eseguendo le operazioni di cifratura.

Capitolo 5

5.1

CONCLUSIONI

In questo lavoro, sono state introdotte una serie di modifiche all'ambiente Java Muskel (versione 2.0) per la programmazione parallela, una libreria che permette di usare gli skeleton per creare applicazioni parallele sfruttando i macro data flow (MDF).

In particolare, sono stati fatti degli esperimenti per valutare la possibilità di introdurre la sicurezza di codice e dati per garantire autenticazione, cifratura e integrita nelle comunicazioni tra i ***controlthread*** e gli interpreti remoti, che potrebbero comunicare attraverso reti pubbliche e quindi non sicure..

E' stata implementata una forma di comunicazione che sfrutta gli ***SslRMISocketFactory*** e che incide sulle prestazioni in base alla percentuale scelta di worker che utilizzano SSL, tra quelli disponibili, e considerando la grana del calcolo scelta per la computazione.

E' stato dimostrato come la scalabilità venga garantita anche con l'utilizzo di SSL, sia in tutte le macchine che su un sottoinsieme, tra quelle disponibili.

E’ stato dimostrato inoltre come i costi di setup per connettere i Socket SSL inficino poco sulle prestazioni, soprattutto in base alla percentuale di worker SSL utilizzata e alla grana scelta.

Si è notato, infine che anche nelle computazioni con grana medio/grossa si mantiene una ottima scalabilità e una buona efficienza.

La sicurezza è un obiettivo fondamentale in piccoli e grandi sistemi distribuiti, come i multi cluster e le architetture GRID, e questa implementazione potrebbe servire anche nell’implementazione di altre tecnologie per sistemi a skeleton.

5.2

FUTURE WORK

Gli sviluppi che posso essere fatti alla versione di muskel presentata in questa tesi, riguardano:

- il metodo di ricerca dei nodi sicuri.
- una configurazione automatica nell’implementazione del Best Effort in base alle caratteristiche della rete.
- una gestione ottimizzata nel manager dei controlthread che scoprono attraverso la RemoteException che il RemoteInterpreter associato è caduto per un qualunque motivo.

Tali obiettivi sono stati trattati, con qualche semplificazione, perché non rientrano negli scopi principali di questa tesi che puntava principalmente a realizzare un ambiente a skeleton che utilizzasse autenticazione, cifratura e integrità di codice e dati anche su reti pubbliche e/o non protette, considerate insicure.

BIBLIOGRAFIA

[1] Sun, The Java home page, 2002, http://java.sun.com

[2] JavaGrande. The JAvaGrande home page, 2002, http://www.javagrande.org.

[3] D.C. Hyde, Java and differenti flavors of parallel programming models, in: R. Buyya (Ed), High Performance Cluster Computing, Prentice-Hall, Englewood Cliffs, N, 1999, pp. 274-290.

[4] L.M, Silva, Web-based parallel computing with Java, in: R. Buyya (Ed.), High Performance Cluster Computing, Prentice-Hall, Englewood CCliffs, NJ, 1999, pp. 310-326.

[5] G.Antoniu, L. Bougè, P. Hatcher, M MacBeth, K. McGuigan, R. Namyst, Compiling multithreaded Java bytecode for distributed execution, in: A. Bode, T. Ludwig, W. Karl, R. Wismuller (Eds.), EuroPar'2000 Parallel Processing, LNCS, No. 1900, Springer, Berlin, 2000, pp. 1039-1052.

[6] Y. Aridor, M Factor, A. Teperman, cJVM: a single system image of a JVM on a cluster, in: Proceedings of tehe International Conference on Parallel Processing, Fukushima, Japan, 1999, http://www.haifa.il.ibm.com/projects/systems/cjvm/index.html.

[7] P.Teti, Lithium: a Java skeleton environment, Master's Thesis, Department of Computer Science, University of Pisa, October 2001 (in Italian).

[8] M.Danelutto & P.Teti: Lithium: A Structured Parallel Programming Environment in Java Department of Computer Science, University of Pisa, October 2002.

[9] M.Cole, Algorithmic Skeletons: Structured Management of Parallel Computations, Researc Monographs in Parallel and Distributed Computing, Pitman, London, 1989.

[10] P.Au, j. Darlington, M. Ghanen, Y. Guo, H. To, J. Yang, Cordinating heterogeneous parallel computation, in: L. Bouge, P. Fraigniaud, A. Mignotte, Y. Robert (Eds.), Europar'96, Springer, Berlin, 1996, pp 601-614.

[11] B.Bacci, M. Danelutto, S. Pelagatti, M. Vanneschi, SkIE: a heterogeneous environment for HPC applications, Parallel Comput, 25 (1999) 1827-1852.

[12] J.Serot, D. Ginhac, R. Chapuis, J Derutin, Fast prototyping of parallel-vision applications using fuctional skeletons, Mach Vision Appl, 12 (2001) 217-290, Springer, Berlin.

[13] M.Danelutto, R.D. Cosmo, X. Leroy, S. Pelagatti, Parallel functional programming with skeletons: the OCAMLP3L experiment, in: Proceeding of the ACM Sigplan Workshop on ML, 1998, pp 31-39.

[14] M.Danelutto, M. Stigliani, SKElib: parallel programming with skeleton in C, in: A. Bode, T. Ludwing, W. Karl, R. Wismuller (Eds.), EuroPar'2000 Parallel Processing, LNCS, No. 1900, Springer, Berlin, 2000, pp. 1175-1184.

[15] H.Kuchen, A skeleton library, Technical Report 6/02-I, Angewandte Mathematik und Informatik, University of Munster, 2002.

[16] S.Pelagatti, Structured Development of Parallel Programs, Taylor & Francis, London 1998.

[17] M.Danelutto, Task farm computations in Java, in: Buback, Afsarmanesh, Williams, Hertzberger (Eds.), High Performance Computing and Networking, LNCS, No 1823, Springer, Berlin, 2000, pp. 385-394.

[18] M. Aldenucci, M. Danelutto, Stream parallel skeleton optimizations, in: Proceedings of the IASTED International Conference Parallel and Distributed Computing and Sustems, IASTED/ACTA Press, Boston, 1999, pp. 955-962.

[19] M.Aldenucci, Automatic program transformation: the meta tool for skeleton-based languages, in: S. Gorlatch, C. Lenguager (EDs.), Costructive Methods for Parallel Programming, Advances in Computation: Theory and Practice, NOVA Science Publisher, New York, 2002, pp. 59-78, (draft available at ftp://ftp.unipi.it/pub/Papers/aldinuc/meta_book_dft.ps.gz).
[20] M. Danelutto, dynamic run time support for skeletons, in: E.H. D'Hollander, G.R Joubert, F.J Peters, H.J. Sips (Eds.), Proceedings of the International Conference ParCo99, Vol. PArallell Computing Fundamentals and Applications, Imperial College Press, 1999, pp. 460-467.
[21] M.Danelutto, Efficient support for skeletons on workstation cluster, Parallel Process. Lett. 11 (1) (2001) 41-56.
[22] C. Nester, R. Philippsen, B. Haumacher. A more efficient RMI for Java, in: Proceedings of the ACM 1999 JavaGrande Conference, 1999, pp. 152-157.
[23] Sun Java JSSE page http://java.sun.com/javase/technologies/security/
[24] SETI@home home page http://setiathome.berkeley.edu/
[25] Sun Java RMI page http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp
[26] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P3L: A Structured High level programming language and its structured support. Conc. Practice and Experience,7(3):225–255, 1995.
[27] M. Vanneschi. The Programming Model of ASSIST, an Environment for Parallel and Distributed Portable Applications.Parallel Computing, 12, December 2002.
[28] M. Aldinucci, A. Petrocelli, E. Pistoletti, M. Torquati, M. Vanneschi, L. Veraldi, and C. Zoccolo. Dynamic reconfiguration of grid-aware applications in ASSIST. In 11th Intl Euro-Par: Parallel and Distributed Computing, volume 3648 of LNCS, pages 771–781, Lisboa, Portugal, August 2005. Springer Verlag.
[29] M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. Parallel Computing,30(3):389–406, 2004.
[30] A. Benoit, M. Cole, S. Gilmore, and J. Hillston. Flexible Skeletal Programming with eSkel. In 11th Intl Euro-Par: Parallel and Distributed Computing, volume 3648 of LNCS,Lisboa, Portugal, Aug. 2005. Springer Verlag.
[31] J. S´erot and D. Ginhac. Skeletons for parallel image processing: an overview of the SKiPPER project. Parallel Computing,28(12):1785–1808, Dec 2002.
[32] H. Kuchen. A Skeleton Library. In Euro-Par 2002, Parallel Processing, number 2400 in LNCS, pages 620–629.Springer Verlag, August 2002.
[33] M. Danelutto. QoS in parallel programming through application managers. In Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-based processing. IEEE, 2005. Lugano.
[34] M. Aldinucci, M. Danelutto, and P. Teti. An advanced environment supporting structured parallel programming in Java. Future Generation Computer Systems, 19(5):611–626,2003. Elsevier Science.
[35] SUN JDK KEYTOOL http://java.sun.com/j2se/1.3/docs/tooldocs/win32/keytool.html
[36] Sun JDK page http://java.sun.com/javase/downloads/index.jsp
[37] Sun Java RMI http://java.sun.com/j2se/1.5.0/docs/guide/rmi/index.html

APPENDICE

Codice

Il codice di muskel è suddiviso in 4 package principali: **examples**, **map**, **muskel** e **skeleton**. Il cuore di muskel oltre al package principale, si suddivide ulteriormente nei package **mdf**, **stream** e **util**.

ComputeTest.java

```
import examples.Square;
import muskel.mdf.Task;
import muskel.stream.IntegerStreamInputManager;

/**
 * Classe per testare il tempo di completamento medio di una Compute
 * */
public class ComputeTest {

	public static void main(String[] args) {

		int howmuch = 8;
		IntegerStreamInputManager ism = new IntegerStreamInputManager(howmuch);

		long tot = 0;

		Square w = new Square();//Compute da testare
		int x,y;

		Task[] t = new Task[1];
		Task[] result;

		while(ism.hasNext())
		{
			x = (Integer) ism.next();
			t[0] = new Task(new Integer(x));
			long inizio = System.currentTimeMillis();

			result = w.compute(t);

			long fine = System.currentTimeMillis();
			y = (Integer) result[0].getValue();
			tot += fine - inizio;

			System.out.println("f("+x+"): " + y);
		}

		System.out.println("Durata media per "+howmuch+" input: " + tot/howmuch);

	}

}
```

Package examples

SampleCode.java

```
package examples;
/* This is the muskel: a skeleton/RMI based, parallel programming library
Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
```

```
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

import skeleton.Compute;
import skeleton.Farm;
import skeleton.Pipeline;
import muskel.*;
import muskel.stream.ConsoleOutputManager;
import muskel.stream.InputManager;
import muskel.stream.IntegerStreamInputManager;
import muskel.stream.OutputManager;

/**
 * Codice di esempio per creare una applicazione parallela
 * usando muskel.
 *
 * @author marcod
 *
 */

public class SampleCode {

	public static void main(String[] args) {

		int parDegree;
		try { parDegree = Integer.parseInt(args[0]);
		} catch(ArrayIndexOutOfBoundsException e) { parDegree = 1; }

		Compute inc1 = new Inc();
		Compute sq1 = new Square();
		Compute f1 = new Farm(sq1);
		Compute main = new Pipeline(inc1,f1);

		InputManager sampleISM =
			(InputManager) new IntegerStreamInputManager(100);
		OutputManager sampleOSM =
			(OutputManager) new ConsoleOutputManager();

		Manager manager = new Manager(main, sampleISM, sampleOSM);
		manager.setContract(new ParDegree(parDegree));

		manager.compute();
	}

}
```

Comp.java

```
package examples;
/* This is the muskel: a skeleton/RMI based, parallel programming library
Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

import muskel.mdf.Task;
```

```java
import skeleton.Compute;


/**
 * Classe Compute di esempio, che compone (sommma, in effetti)
 * i due input tokens Integer in un output token
 * @author marcod
 *
 */
public class Comp extends Compute {

	private static final long serialVersionUID = 1L;

	@Override
	public Task[] compute(Task[] task) {

		Integer iI = (Integer) task[0].getValue();
		Integer jI = (Integer) task[1].getValue();

		Task[] toReturn = {new Task(iI+jI)};
		System.out.println(toReturn);
		return toReturn;
	}
}
```

Id.java

```java
package examples;

import muskel.mdf.Task;
import skeleton.Compute;

public class Id extends Compute {

	@Override
	public Task[] compute(Task[] task) {
		return task;
	}
}
```

Inc.java

```java
package examples;
/* This is the muskel: a skeleton/RMI based, parallel programming library
Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

import muskel.mdf.Task;
import skeleton.Compute;

/**
 * Prende un Integer e lo incrementa. Usato a scopo di test.
 * E' incluso un delay loop per dimostrare i timings.
 * @author Danelutto Marco
 *
 */
public class Inc extends Compute {

	private static final long serialVersionUID = 8291468701762043559L;
```

```
	@Override
	public Task[] compute(Task[] task) {

		Integer xI = (Integer) task[0].getValue();

		int x = xI.intValue();
		double y = (double) x;
		for(int i=0;i<2000000;i++)
			y = Math.sin(y);

		System.out.println("INC : computes "+x+" ("+y+")");

		Task[] toReturn = {new Task(new Integer(++x))};

		return toReturn;
	}
}
```

IntegerInc.java

```
package examples;

import muskel.mdf.Task;
import skeleton.Compute;

public class IntegerInc extends Compute {

	@Override
	public Task[] compute(Task[] task) {
		Integer i = (Integer)task[0].getValue();
		i++;
		Task []  ret =  {new Task(i)};
		return ret;
	}

}
```

Map8.java

```
package examples;

import map.MapCollector;
import map.MapEmitter;
import muskel.*;
import muskel.mdf.*;
import skeleton.*;

/**
 * Codice muskel di una Map con 8 workers, generato da IMuskelGUI.
 * */
public class Map8 extends ParCompute {

	public Map8(Manager manager) {

		super(null);

		/* please use your favourite id generator */
		IdGenerator idGen = new IntegerIdGenerator();

		InstructionId i1_Id = new InstructionId(idGen.generateId());
		InstructionId i2_Id = new InstructionId(idGen.generateId());
		InstructionId i3_Id = new InstructionId(idGen.generateId());
		InstructionId i4_Id = new InstructionId(idGen.generateId());
		InstructionId i5_Id = new InstructionId(idGen.generateId());
		InstructionId i6_Id = new InstructionId(idGen.generateId());
		InstructionId i7_Id = new InstructionId(idGen.generateId());
		InstructionId i8_Id = new InstructionId(idGen.generateId());
		InstructionId i9_Id = new InstructionId(idGen.generateId());
		InstructionId i10_Id = new InstructionId(idGen.generateId());

		/*MDFi n.1 setup*/
		Destination[] dests_1 = new Destination[8];
```

```
                Destination d0_1 = new Destination(new InputPositionId(new IntegerIdentifier(0)),
i2_Id, Mdfi.NoGraphId);
                dests_1[0] = d0_1;

                Destination d1_1 = new Destination(new InputPositionId(new IntegerIdentifier(0)),
i3_Id, Mdfi.NoGraphId);
                dests_1[1] = d1_1;

                Destination d2_1 = new Destination(new InputPositionId(new IntegerIdentifier(0)),
i4_Id, Mdfi.NoGraphId);
                dests_1[2] = d2_1;

                Destination d3_1 = new Destination(new InputPositionId(new IntegerIdentifier(0)),
i5_Id, Mdfi.NoGraphId);
                dests_1[3] = d3_1;

                Destination d4_1 = new Destination(new InputPositionId(new IntegerIdentifier(0)),
i6_Id, Mdfi.NoGraphId);
                dests_1[4] = d4_1;

                Destination d5_1 = new Destination(new InputPositionId(new IntegerIdentifier(0)),
i7_Id, Mdfi.NoGraphId);
                dests_1[5] = d5_1;

                Destination d6_1 = new Destination(new InputPositionId(new IntegerIdentifier(0)),
i8_Id, Mdfi.NoGraphId);
                dests_1[6] = d6_1;

                Destination d7_1 = new Destination(new InputPositionId(new IntegerIdentifier(0)),
i9_Id, Mdfi.NoGraphId);
                dests_1[7] = d7_1;

                Mdfi i1 = new Mdfi(manager, i1_Id, new MapEmitter(), 1, 8, dests_1);

                /*MDFi n.2 setup*/
                Destination[] dests_2 = new Destination[1];

                Destination d0_2 = new Destination(new InputPositionId(new IntegerIdentifier(0)),
i10_Id, Mdfi.NoGraphId);
                dests_2[0] = d0_2;

                Mdfi i2 = new Mdfi(manager, i2_Id, new Square(), 1, 1, dests_2);

                /*MDFi n.3 setup*/
                Destination[] dests_3 = new Destination[1];

                Destination d0_3 = new Destination(new InputPositionId(new IntegerIdentifier(1)),
i10_Id, Mdfi.NoGraphId);
                dests_3[0] = d0_3;

                Mdfi i3 = new Mdfi(manager, i3_Id, new Square(), 1, 1, dests_3);

                /*MDFi n.4 setup*/
                Destination[] dests_4 = new Destination[1];

                Destination d0_4 = new Destination(new InputPositionId(new IntegerIdentifier(2)),
i10_Id, Mdfi.NoGraphId);
                dests_4[0] = d0_4;

                Mdfi i4 = new Mdfi(manager, i4_Id, new Square(), 1, 1, dests_4);

                /*MDFi n.5 setup*/
                Destination[] dests_5 = new Destination[1];

                Destination d0_5 = new Destination(new InputPositionId(new IntegerIdentifier(3)),
i10_Id, Mdfi.NoGraphId);
                dests_5[0] = d0_5;

                Mdfi i5 = new Mdfi(manager, i5_Id, new Square(), 1, 1, dests_5);

                /*MDFi n.6 setup*/
                Destination[] dests_6 = new Destination[1];
```

```
                Destination d0_6 = new Destination(new InputPositionId(new IntegerIdentifier(4)),
i10_Id, Mdfi.NoGraphId);
                dests_6[0] = d0_6;

                Mdfi i6 = new Mdfi(manager, i6_Id, new Square(), 1, 1, dests_6);

                /*MDFi n.7 setup*/
                Destination[] dests_7 = new Destination[1];

                Destination d0_7 = new Destination(new InputPositionId(new IntegerIdentifier(5)),
i10_Id, Mdfi.NoGraphId);
                dests_7[0] = d0_7;

                Mdfi i7 = new Mdfi(manager, i7_Id, new Square(), 1, 1, dests_7);

                /*MDFi n.8 setup*/
                Destination[] dests_8 = new Destination[1];

                Destination d0_8 = new Destination(new InputPositionId(new IntegerIdentifier(6)),
i10_Id, Mdfi.NoGraphId);
                dests_8[0] = d0_8;

                Mdfi i8 = new Mdfi(manager, i8_Id, new Square(), 1, 1, dests_8);

                /*MDFi n.9 setup*/
                Destination[] dests_9 = new Destination[1];

                Destination d0_9 = new Destination(new InputPositionId(new IntegerIdentifier(7)),
i10_Id, Mdfi.NoGraphId);
                dests_9[0] = d0_9;

                Mdfi i9 = new Mdfi(manager, i9_Id, new Square(), 1, 1, dests_9);

                /*MDFi n.10 setup*/
                Destination d0_10 = new Destination(new InputPositionId(new IntegerIdentifier(0)),
Mdfi.NoInstrId, Mdfi.NoGraphId);
                Destination[] dests_10 = {d0_10};

                Mdfi i10 = new Mdfi(manager, i10_Id, new MapCollector(), 8, 1, dests_10);
                /*MDF graph setup*/
                program = new MdfGraph();
                program.addInstruction(i1);
                program.addInstruction(i2);
                program.addInstruction(i3);
                program.addInstruction(i4);
                program.addInstruction(i5);
                program.addInstruction(i6);
                program.addInstruction(i7);
                program.addInstruction(i8);
                program.addInstruction(i9);
                program.addInstruction(i10);
                program.setInputInstruction(i1);
                program.setOutputInstruction(i10);

        }
}
```

Map8Example.java

```
package examples;
/* This is the muskel: a skeleton/RMI based, parallel programming library
Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
```

```
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

import skeleton.Compute;

import muskel.*;
import muskel.stream.ConsoleOutputManager;
import muskel.stream.InputManager;
import muskel.stream.IntegerVectorStreamInputManager;
import muskel.stream.OutputManager;

/**
 * Codice di esempio per creare una app applicazione parallela
 * usando muskel.
 *      Lo skeleton Map8 utilizzato Ã¨ un Map a "grado fisso di parallelismo"
 *      con 8 Square come workers.
 *      Map8 Ã¨ un ParCompute (cioe' uno skeleton definito dall'utente).
 * @author albanese
 */
public class Map8Example {

        public static void main(String[] args) {
                int parDegree;
                try { parDegree = Integer.parseInt(args[0]);
                } catch(ArrayIndexOutOfBoundsException e) {
                        //il grado di parallellismo richiesto a muskel
                        parDegree = 1;
                }

                Manager manager = new Manager();

                Compute map8 = new Map8(manager);

                //quanti task calcolare
                int nTasks = 5;
                //il grado di parallelismo della map
                int mapdegree = 8;

                InputManager sampleISM =
                        (InputManager) new IntegerVectorStreamInputManager(mapdegree, nTasks);
                OutputManager sampleOSM =
                        (OutputManager) new ConsoleOutputManager();

                manager.setInputManager(sampleISM);
                manager.setOutputManager(sampleOSM);
                manager.setContract(new ParDegree(parDegree));
                manager.setProgram(map8);

                manager.compute();
        }

}
```

MapExample.java

```
package examples;
/* This is the muskel: a skeleton/RMI based, parallel programming library
Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

import skeleton.*;
```

```
import map.MapFactory;
import muskel.*;
import muskel.stream.ConsoleOutputManager;
import muskel.stream.InputManager;
import muskel.stream.IntegerVectorStreamInputManager;
import muskel.stream.OutputManager;

/**
 * Esempio di utilizzo di MapFactory per la creazione skeleton map
 * con grado di parallelismo e tipo di worker specificati
 *
 * @author albanese
 */

public class MapExample {

	public static void main(String[] args) {

		if(args.length != 2) {
			printUsage();
			return;
		}

		//il grado di parallellismo richiesto a muskel
		int parDegree = 1;

		//il grado di parallelismo della map
		int mapDegree = 5;

		try {
			mapDegree = Integer.parseInt(args[0]);
			parDegree = Integer.parseInt(args[1]);

			if(mapDegree < 1 || parDegree < 1) {
				System.out.println("mapDegree and parDegree must be >= 1");
				return;
			}

		} catch(NumberFormatException e) {
			printUsage();
		}

		System.out.println("Requested a "+mapDegree+"-degree Map on "+parDegree+" Remote
Interpreters.");

		//il class del worker
		Class<Square> workerClass = Square.class;

		//quanti task calcolare
		int nTasks = 1;

		Manager manager = new Manager();

		Compute map = MapFactory.newMap(manager, workerClass, mapDegree);

		if(map == null) {
			System.out.println("Error occured during map creation, sorry.");
			return;
		}

		InputManager sampleISM =
			(InputManager) new IntegerVectorStreamInputManager(mapDegree, nTasks);
		OutputManager sampleOSM =
			(OutputManager) new ConsoleOutputManager();

		manager.setInputManager(sampleISM);
		manager.setOutputManager(sampleOSM);
		manager.setContract(new ParDegree(parDegree));
		manager.setProgram(map);

		manager.compute();
	}

	private static void printUsage() {
```

```
                System.out.println("usage: java examples.MapExample mapDegree parDegree");
        }

}
```

ProvaMD.java

```
package examples;

import muskel.mdf.Task;
import skeleton.Compute;
import skeleton.Farm;
import skeleton.Pipeline;

public class ProvaMD {



/**
         * @param args
         */
        public static void main(String[] args) {

                Compute firstStage = new IntegerInc();
                Compute secondStage = new Farm(new SquareInteger());

                Compute main = new Pipeline(firstStage,secondStage);

        }

}
```

SampleCode.java

```
package examples;
/* This is the muskel: a skeleton/RMI based, parallel programming library
Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

import skeleton.Compute;
import skeleton.Farm;
import skeleton.Pipeline;
import muskel.*;
import muskel.stream.ConsoleOutputManager;
import muskel.stream.InputManager;
import muskel.stream.IntegerStreamInputManager;
import muskel.stream.OutputManager;

/**
 * Codice di esempio per creare una applicazione parallela
 * usando muskel.
 *
 * @author marcod
 *
 */

public class SampleCode {

        public static void main(String[] args) {
```

```
                int parDegree;
                try { parDegree = Integer.parseInt(args[0]);
                } catch(ArrayIndexOutOfBoundsException e) { parDegree = 1; }

                Compute inc1 = new Inc();
                Compute sq1 = new Square();
                Compute f1 = new Farm(sq1);
                Compute main = new Pipeline(inc1,f1);

                InputManager sampleISM =
                        (InputManager) new IntegerStreamInputManager(100);
                OutputManager sampleOSM =
                        (OutputManager) new ConsoleOutputManager();

                Manager manager = new Manager(main, sampleISM, sampleOSM);
                manager.setContract(new ParDegree(parDegree));

                manager.compute();
        }

}
```

SampleMdf2.java

```
package examples;
/* This is the muskel: a skeleton/RMI based, parallel programming library
Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

import skeleton.Compute;
import skeleton.ParCompute;
import muskel.*;
import muskel.mdf.*;
import muskel.stream.ConsoleOutputManager;
import muskel.stream.InputManager;
import muskel.stream.IntegerStreamInputManager;
import muskel.stream.OutputManager;

/**
 * Codice di esempio per la costruzione di una applicazione usando muskel.
 *
 * L'esempio crea un grafo MDF che non puÃ² essere il risultato
 * di annidamenti arbitrari di pipe e farm.
 *
 * @author marcod
 *
 */
public class SampleMdf2 {

        public static void main(String[] args) {

                int parDegree;
                try { parDegree = Integer.parseInt(args[0]);
                } catch(ArrayIndexOutOfBoundsException e) { parDegree = 1; }

                Manager manager = new Manager();

                Compute inc1 = new Splitter();

                IntegerIdGenerator intIdGen = new IntegerIdGenerator();

                InstructionId i1_Id = new InstructionId(intIdGen.generateId());
```

```
                InstructionId i2_Id = new InstructionId(intIdGen.generateId());
                InstructionId i3_Id = new InstructionId(intIdGen.generateId());
                InstructionId i4_Id = new InstructionId(intIdGen.generateId());

                Destination d1 = new Destination(
                                new InputPositionId(new IntegerIdentifier(0)), i2_Id, Mdfi.NoGraphId);

                Destination d2 = new Destination(
                                new InputPositionId(new IntegerIdentifier(0)), i3_Id, Mdfi.NoGraphId);

                Destination[] dests = {d1, d2};
                Mdfi i1 = new Mdfi(manager,i1_Id,inc1,1,2,dests);

                Compute sq1 = new Square();

                Destination d3 = new Destination(
                                new InputPositionId(new IntegerIdentifier(0)), i4_Id, Mdfi.NoGraphId);

                Destination[] dests1 = {d3};

                Mdfi i2 = new Mdfi(manager, i2_Id, sq1, 1, 1, dests1);

                Compute sq2 = new Square();

                Destination d4 = new Destination(
                                new InputPositionId(new IntegerIdentifier(1)), i4_Id, Mdfi.NoGraphId);

                Destination[] dests2 = {d4};
                Mdfi i3 = new Mdfi(manager, i3_Id, sq2, 1, 1, dests2);

                Compute add = new Comp();

                Destination d5 = new Destination(
                                new InputPositionId(new IntegerIdentifier(0)), Mdfi.NoInstrId,
Mdfi.NoGraphId);

                Destination[] dests3 = {d5};

                Mdfi i4 = new Mdfi(manager, i4_Id, add, 2, 1, dests3);

                MdfGraph graph = new MdfGraph();

                graph.setInputInstruction(i1);
                graph.addInstruction(i1);
                graph.addInstruction(i2);
                graph.addInstruction(i3);
                graph.addInstruction(i4);
                graph.setOutputInstruction(i4);

                ParCompute main = new ParCompute(graph);

                InputManager sampleISM =
                        (InputManager) new IntegerStreamInputManager(10);
                OutputManager sampleOSM =
                        (OutputManager) new ConsoleOutputManager();

                manager.setInputManager(sampleISM);
                manager.setOutputManager(sampleOSM);
                manager.setContract(new ParDegree(parDegree));
                manager.setProgram(main);

                manager.compute();
        }

}
```

SampleMdfg.java

```
package examples;
/* This is the muskel: a skeleton/RMI based, parallel programming library
Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
```

```
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

import skeleton.Compute;
import skeleton.ParCompute;
import muskel.*;
import muskel.mdf.Destination;
import muskel.mdf.InputPositionId;
import muskel.mdf.InstructionId;
import muskel.mdf.IntegerIdGenerator;
import muskel.mdf.IntegerIdentifier;
import muskel.mdf.MdfGraph;
import muskel.mdf.Mdfi;
import muskel.stream.ConsoleOutputManager;
import muskel.stream.InputManager;
import muskel.stream.IntegerStreamInputManager;
import muskel.stream.OutputManager;

/**
 * This shows how to build a parallel application using muskel.
 * This example runs with version May06 of muskel (or later);
 *
 * @author marcod
 *
 */
public class SampleMdfg {

        public static void main(String[] args) {


                int parDegree;
                try { parDegree = Integer.parseInt(args[0]);
                } catch(ArrayIndexOutOfBoundsException e) { parDegree = 1; }

                Manager manager = new Manager();

                IntegerIdGenerator intIdGen = new IntegerIdGenerator();

                InstructionId i1_Id = new InstructionId(intIdGen.generateId());
                InstructionId i2_Id = new InstructionId(intIdGen.generateId());

                Compute inc1 = new Inc();

                Destination d = new Destination(
                                new InputPositionId(new IntegerIdentifier(0)), i2_Id, Mdfi.NoGraphId);

                Destination[] dests = {d};

                Mdfi i1 = new Mdfi(manager, i1_Id, inc1, 1, 1, dests);

                Compute sq1 = new Square();

                Destination d1 = new Destination(
                                new InputPositionId(new IntegerIdentifier(0)), Mdfi.NoInstrId,
Mdfi.NoGraphId);

                Destination[] dests1 = {d1};

                Mdfi i2 = new Mdfi(manager, i2_Id, sq1, 1, 1, dests1);

                MdfGraph graph = new MdfGraph();

                graph.setInputInstruction(i1);
                graph.addInstruction(i1);
                graph.addInstruction(i2);
                graph.setOutputInstruction(i2);
```

```
                ParCompute main = new ParCompute(graph);

                InputManager sampleISM =
                        (InputManager) new IntegerStreamInputManager(10);
                OutputManager sampleOSM =
                        (OutputManager) new ConsoleOutputManager();

                manager.setInputManager(sampleISM);
                manager.setOutputManager(sampleOSM);
                manager.setContract(new ParDegree(parDegree));
                manager.setProgram(main);

                manager.compute();
        }

}
```

SESampleCode.java

```
package examples;
/* This is the muskel: a skeleton/RMI based, parallel programming library
Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

import skeleton.Compute;
import skeleton.Pipeline;
import muskel.*;
import muskel.stream.ConsoleOutputManager;
import muskel.stream.InputManager;
import muskel.stream.IntegerStreamInputManager;
import muskel.stream.NonBlockingInputManager;
import muskel.stream.OutputManager;
import muskel.stream.DynamicInputManager;

/**
 * Utilizzo tipo di SECompute in congiunzione con un DynamicInputManager che
 * supporta aggiunta di Task a runtime.
 *
 * @author albanese
 *
 */
public class SESampleCode {

        public static void main(String[] args) {
                int parDegree;
                try { parDegree = Integer.parseInt(args[0]);
                } catch(ArrayIndexOutOfBoundsException e) { parDegree = 1; }

                Compute sq1 = new SESquare();
                Compute sq2 = new SESquare();

                Compute main = new Pipeline(sq1,sq2);

                int numTask = 10;

                NonBlockingInputManager sampleISM = new IntegerStreamInputManager(numTask);

                InputManager sampleSeISM = new DynamicInputManager(sampleISM);

                OutputManager sampleOSM =
```

```
                        (OutputManager) new ConsoleOutputManager();

                Manager manager = new Manager(main, sampleSeISM, sampleOSM);
                manager.setContract(new ParDegree(parDegree));

                manager.compute();
        }

}
```

SESquare.java

```
package examples;
/* This is the muskel: a skeleton/RMI based, parallel programming library
Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

import muskel.mdf.Task;
import skeleton.SECompute;

/**
 * Questo e' una funzione Compute che calcola il quadrato di un numero.
 * Il delay loop Ã¨ stato aggiunto per dimostrare i timings.
 *
 * Genera come effetto collaterale un task se l'input Ã¨ pari.
 *
 * @author marcod
 * @author albanese
 *
 */
 public class SESquare extends SECompute {

        private static final long serialVersionUID = 2295172192703176107L;

        /**
          * Implements a simple worker calculating the square of an Integer
          */
         public Task[] compute(Task[] task) {

                int numIter = 2000000;//2M

                int x = (Integer) task[0].getValue();
                double y = (double) x;

                //delay loop
                for(int i=0; i<numIter; i++)
                        y = Math.sin(y);

                //se x e' pari genero un nuovo task dispari
                if (x % 2 == 0) {
                        int newTask = x-1;
                        System.out.println("New task generated: "+(newTask) +" from task "+x);
                        setGeneratedTask(new Task[]{new Task(new Integer(newTask))});
                }

                int square = x*x;

                Task[] toReturn = {new Task(new Integer(square))};

                System.out.println("SESquare("+x+"): "+(square));
```

```
                return toReturn;
        }
}
```

Splitter.java

```
package examples;
/* This is the muskel: a skeleton/RMI based, parallel programming library
Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

import muskel.mdf.Task;
import skeleton.Compute;

/**
 * Codice di una funzione Compute che prende un Integer
 * e ne produce due come output tokens, uno con il successore
 * e l'altro con il predecessore del valore dell' input token.
 * @author marcod
 *
 */
public class Splitter extends Compute {

        private static final long serialVersionUID = 1L;

        @Override
        public Task[] compute(Task[] task) {

                int i = (Integer) task[0].getValue();

                Task[] toReturn = {new Task(new Integer(i-1)),
                                                        new Task(new Integer(i+1))};

                return toReturn;
        }
}
```

Square.java

```
package examples;
/* This is the muskel: a skeleton/RMI based, parallel programming library
Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

import muskel.mdf.Task;
```

```java
import skeleton.Compute;

/**
 * This is a compute function computing the square of a number.
 * A delay loop is included for demonstrating timings.
 *
 * @author marcod
 *
 */
 public class Square extends Compute {

	private static final long serialVersionUID = 4244043631856051889L;

	/**
	  * Implements a simple worker calculating the square of an Integer
	  */
	 public Task[] compute(Task[] task) {

		int numIter = 50000000;//50M

		int x = (Integer) task[0].getValue();
		double y = (double) x;

		//delay loop
		for(int i=0; i<numIter; i++)
			y = Math.sin(y);

		int square = x*x;

		System.out.println("Square("+x+"): "+(square));
		Task[] toReturn = {new Task(new Integer(square))};

		return toReturn;
	 }
}
```

SquareInteger.java

```java
package examples;

import muskel.mdf.Task;
import skeleton.Compute;

public class SquareInteger extends Compute {

	@Override
	public Task[] compute(Task[] task) {
		Integer i = (Integer)task[0].getValue();
		i *= i;
		Task []  ret =  {new Task(i)};
		return ret;
	}

}
```

Package map

MapCollector.java

```java
package map;

import java.util.Vector;
import muskel.mdf.Task;
import skeleton.Compute;

/**
 * Semplice collector per implementare una Map.
 * */
public class MapCollector extends Compute {

	private static final long serialVersionUID = -3656835617041306069L;
```

```
        /**
         * Restituisce un Task[] avente come unico elemento un Vector
         * contenente gli elementi di <code>task</code>.
         * @param task I task da raccogliere in un unico Vector
         * */
        @SuppressWarnings("unchecked")
        @Override
        public Task[] compute(Task[] task) {

                Vector v = new Vector(task.length);

                //collect the results
                for (Task t: task)
                        v.add(t.getValue());

                //create the output
                Task[] toReturn = {new Task(v)};

                return toReturn;
        }
}
```

MapEmitter.java

```
package map;

import java.util.Vector;
import muskel.mdf.Task;
import skeleton.Compute;

/**
 * Semplice emitter per implementare una Map.
 * */
public class MapEmitter extends Compute {

        private static final long serialVersionUID = 7824884734899083241L;

        /**
         * Restituisce un Task[] avente come elementi, i corrispettivi
         * elementi del vettore contenuto in <code>task[0]</code>
         * (il primo elemento dell'array Task passato come parametro)
         * <code>task[0]</code> deve essere un Vector di lunghezza
         * <code>n</code> (specificato alla creazione).
         * @param task array avente come primo (e unico) elemento un Vector
         * @throws IllegalArgumentException se task[0] non e' un Vector
         * */
        @SuppressWarnings("unchecked")
        @Override
        public Task[] compute(Task[] task) {

                Object o = task[0].getValue();

                Vector t;

                if (o instanceof Vector)
                        t = (Vector) o;
                else throw new IllegalArgumentException();

                Task[] toReturn = new Task[t.size()];

                for(int i = 0; i < t.size(); i++)
                        toReturn[i] = new Task(t.elementAt(i));

                return toReturn;
        }

}
```

MapFactory.java

```
package map;
```

```java
import java.io.CharArrayWriter;
import java.io.PrintWriter;
import java.lang.reflect.InvocationTargetException;
import java.util.Arrays;

import javax.tools.*;
import javax.tools.JavaCompiler.CompilationTask;

import muskel.Manager;
import muskel.util.JavaSourceFromString;

import skeleton.Compute;
import skeleton.ParCompute;

/**
 * Classe che permette la creazione di Map skeleton
 * con worker e grado di parallelismo specificati.
 * @author albanese
 * */
public class MapFactory {

	/**
	 * Restituisce un Map skeleton con worker e grado di
	 * parallelismo specificati
	 * @return Un ParCompute che implementa lo skeleton Map,
	 *                    oppure null se non e' stato possibile crearlo
	 * */
	@SuppressWarnings("unchecked")
	public static synchronized ParCompute newMap(Manager manager, Class<? extends Compute>
worker, int degree)
	{
		String className = "Map"+degree;

		ParCompute map = null;

		//Map e' un ParCompute
		Class<? extends ParCompute> mapClass = null;

		boolean classFileAvailable = false;

		classFileAvailable = generateCompileMap(worker, degree, className);

		if (classFileAvailable) {//il class e' stato compilato
			try {
				//riprovo a caricare la classe
				mapClass = (Class<? extends ParCompute>) Class.forName(className);
			} catch (ClassNotFoundException e) {
				//e.printStackTrace();
				return null;//loading fallito restituisco null
			}
		}
		else {//la compilazione e' fallita restituisco null
			return null;
		}

		try {
			//creo un nuovo oggetto ParCompute usando il primo
			//e unico costruttore di Map che ha manager come parametro
			map = (ParCompute) mapClass.getConstructors()[0].newInstance(manager);
		} catch (IllegalArgumentException e) {
			e.printStackTrace();
		} catch (SecurityException e) {
			e.printStackTrace();
		} catch (InstantiationException e) {
			e.printStackTrace();
		} catch (IllegalAccessException e) {
			e.printStackTrace();
		} catch (InvocationTargetException e) {
			e.printStackTrace();
		}

		return (map != null) ? map : null;
	}
```

```
        /**
         * Genera i sorgenti di una Map con worker, grado di parallelismo
         * e il nome della classe, specificati.
         * Successivamente compila i sorgenti creando nella directory
         * corrente il file class in modo da poter creare
         * dinamicamente sue istanze.
         * @return true se la compilazione non fa generato errori, false altrimenti
         * */
        private static synchronized boolean generateCompileMap(Class<? extends Compute> worker, int
degree, String className) {
                //ottengo il compilatore di sistema
                JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();

                //ottengo l'implementazione del file manager standard. Non aggiungo nessun Diagnostic
Listener
                StandardJavaFileManager fileManager = compiler.getStandardFileManager(null, null,
null);

                //Creo il sorgente per il Map
                CreateMapSrc src = new CreateMapSrc(worker, degree);

                JavaSourceFromString javaFileObject = null;

                //creo un JavaFileObject con la stringa creata da CreateMapSrc
                javaFileObject = new JavaSourceFromString(className, src.getJavaSrc());

                //creo la lista di JavaFileObject da compilare
                Iterable<? extends JavaFileObject> fileObjects = Arrays.asList(javaFileObject);

                //creo il compilationTask senza nessun listener per gli errori
                CompilationTask task = compiler.getTask(null, fileManager, null, null, null,
fileObjects);

                //avvio la compilazione
                boolean result = task.call();

                return result;
        }

}

/**
 * Classe CreateMapSrc, crea i sorgenti dell'implementazione con grafi MDF
 * di un Map Skeleton con grado di parallelismo e worker specificati.
 *
 */
class CreateMapSrc {
        /** il nome della classe worker*/
        private String worker;
        /** per indentare */
        private static final String INDENT = "\t";
        /** per indentare */
        private static final String INDENT2 = "\t\t";
        /** il nome della classe da creare*/
        private String className;//Formato: Mapdegree

        private int degree;
        private CharArrayWriter buffer;

        /** PrintBuffer per usufruire della formattazione */
        private PrintWriter pBuffer;

        /**
         * Costruttore di un' istanza di ParseJava
         * @param worker
         *            nome della classe worker (Diverso da <code>null</code>)
         */
        public CreateMapSrc(Class<? extends Compute> worker, int degree) {

                if (worker == null || degree < 1)
                        return;

                this.worker = worker.getName();
                this.degree = degree;
```

```
        className = "Map"+degree;

        // creo gli stream
        buffer = new CharArrayWriter();
        pBuffer = new PrintWriter(buffer);

        printImports();
        pBuffer.println();

        printClassDefinition();
        pBuffer.println();

        printConstructor();
        pBuffer.println();

        printConstructorStaticContent();

        printIdGeneration();

        pBuffer.println(INDENT2 + "/*Emitter MDFi setup*/");
        printEmitterDestinations();
        printEmitterMdf();

        pBuffer.println();

        for (int i = 0; i<degree; i++){
                pBuffer.println(INDENT2 + "/*Worker MDFi n."+i+" setup*/");

                printWorkerDestination(i);
                printWorkerMdf(i);

                pBuffer.println();
        }

        pBuffer.println(INDENT2 + "/*Collector MDFi setup*/");
        printCollectorDestination();
        printCollectorMdf();

        pBuffer.println();

        printGraphSection();

        pBuffer.println();

        printClosingBrackets();

        pBuffer.close();
}

private void printImports() {
        pBuffer.println("import muskel.*;");
        pBuffer.println("import muskel.mdf.*;");
        pBuffer.println("import skeleton.*;");
        pBuffer.println("import examples.*;");
        pBuffer.println("import map.*;");
        pBuffer.println("import static muskel.mdf.Mdfi.*;");
}

private void printClassDefinition() {
        pBuffer.println("public class "+ className +" extends ParCompute {");
}

private void printConstructor() {
        pBuffer.println(INDENT + "public " + className + "(Manager manager) {");
}

private void printConstructorStaticContent() {

        pBuffer.println(INDENT2 + "super(null);");
        pBuffer.println();
        pBuffer.println(INDENT2 + "/* please use your favourite id generator */");
        pBuffer.println(INDENT2 + "IdGenerator idGen = new IntegerIdGenerator();");
        pBuffer.println();
}
```

```
	private void printIdGeneration() {

		pBuffer.println(INDENT2 + "InstructionId emitter_Id = new
InstructionId(idGen.generateId());");

		for(int i = 0; i < degree; i++) {

			pBuffer.println(INDENT2 + "InstructionId w"+i+"_Id = new
InstructionId(idGen.generateId());");
		}

		pBuffer.println(INDENT2 + "InstructionId collector_Id = new
InstructionId(idGen.generateId());");

		pBuffer.println();

	}

	private void printEmitterDestinations() {

		pBuffer.println(INDENT2 + "Destination[] emitter_dests = new Destination["+ degree +
"];");
		pBuffer.println();

		String destName;

		for (int i = 0; i < degree; i++) {

			destName = "d"+ i +"_emitter";

			pBuffer.print(INDENT2 + "Destination "+destName+" = new Destination(");
			pBuffer.print("new InputPositionId(new IntegerIdentifier(0)),");
			pBuffer.println(" w"+ i +"_Id, NoGraphId);");
			pBuffer.println(INDENT2 + "emitter_dests["+i+"] = "+ destName +";");
			pBuffer.println();
		}

	}

	private void printEmitterMdf() {

		pBuffer.print(INDENT2 +"Mdfi emitter = new Mdfi(manager, emitter_Id, new
MapEmitter(), ");
		pBuffer.println("1, "+ degree +", emitter_dests);");

	}

	private void printWorkerDestination(int i) {

		String destsName = "dests_w" + i;

		String destName = "d_w"+i;

		pBuffer.print(INDENT2 + "Destination "+destName+" = new Destination(");
		pBuffer.print("new InputPositionId(new IntegerIdentifier("+i+")),");
		pBuffer.println(" collector_Id, NoGraphId);");

		pBuffer.println(INDENT2 + "Destination[] "+destsName+" = {"+destName+"};");
		pBuffer.println();

	}

	private void printWorkerMdf(int i) {

		pBuffer.print(INDENT2 +"Mdfi w"+ i +" = new Mdfi(manager, w"+ i +"_Id, new
"+worker+"(), ");
		pBuffer.println("1, 1, dests_w"+ i +");");

	}

	private void printCollectorDestination() {

		pBuffer.print(INDENT2 + "Destination d0_collector = new Destination(");
		pBuffer.print("new InputPositionId(new IntegerIdentifier(0)), NoInstrId,
NoGraphId);");
```

```
                pBuffer.println();

                pBuffer.println(INDENT2 + "Destination[] collector_dests = {d0_collector};");
                pBuffer.println();

        }

        private void printCollectorMdf() {

                pBuffer.print(INDENT2 +"Mdfi collector = new Mdfi(manager, collector_Id, new
MapCollector(), ");
                pBuffer.println(degree +", 1, collector_dests);");

        }

        private void printGraphSection() {

                pBuffer.println(INDENT2 + "/*MDF graph setup*/");
                pBuffer.println(INDENT2 + "program = new MdfGraph();");
                pBuffer.println();

                pBuffer.println(INDENT2 + "program.setInputInstruction(emitter);");
                pBuffer.println(INDENT2 + "program.addInstruction(emitter);");

                for (int i = 0; i< degree; i++)
                        pBuffer.println(INDENT2 + "program.addInstruction(w"+ i +");");

                pBuffer.println(INDENT2 + "program.addInstruction(collector);");
                pBuffer.println(INDENT2 + "program.setOutputInstruction(collector);");
        }

        private void printClosingBrackets()  {
                pBuffer.println(INDENT+"}\n\n}\n");
        }

        /**
         * Restituisce il sorgente java generato
         *
         * @return il sorgente java generato
         */
        public String getJavaSrc() {
                return (buffer != null) ? buffer.toString() : "";
        }
```

Package muskel

CodeStorage.java

```
/* This is the muskel: a skeleton/RMI based, parallel programming library
Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

package muskel;

import java.io.Serializable;
import java.util.Collection;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Map;
```

```
import skeleton.Compute;

import muskel.mdf.IdGenerator;
import muskel.mdf.IntegerIdGenerator;
import muskel.mdf.OperationId;

/**
 * This is used to store the Compute in such a way
 * that can be later accessed by a OperationId.
 * Only The latter are passed with MDFi,
 * rather than serialized code (which can be huge).
 * The code is serialized just once, at the very
 * beginning of the computation
 *
 * @author marcod
 * @author albanese
 *
 */
public class CodeStorage implements Serializable {

	//private final boolean debug = Manager.debug;

	private static final long serialVersionUID = 8537789175206285538L;

	HashMap<OperationId, Compute> storage = null;

	/**
	 * Contiene il class di ogni Compute presente.
	 * Necessario per distinguerli ed evitare duplicati
	 * */
	HashSet< Class<Compute> > storedClasses = null;

	IdGenerator idGen = null;

	public CodeStorage() {

		storage = new HashMap<OperationId, Compute>();

		storedClasses = new HashSet<Class<Compute>>();

		idGen = new IntegerIdGenerator();
	}

	/**
	 * Aggiunge un Compute al CodeStorage, se non giÃ  presente,
	 * e ne restituisce l'identificatore.
	 * Qualora il Compute fosse gia' presente,
	 * ne restituisce l'identificatore corrente.
	 * */
	@SuppressWarnings("unchecked")
	public OperationId store(Compute op) {

		OperationId id = null;

		Class<Compute> opClass = (Class<Compute>)  op.getClass();

		if(!storedClasses.contains(opClass)) {// nuovo opcode, lo aggiungo

			id = new OperationId(idGen.generateId());

			storedClasses.add(opClass);
			storage.put(id, op);
		}
		else { // opcode esistente cerco l'id corrispondente

			id = getOperationId(op);//ritorna null se op non e' presente, ma ne e' gia'
stata controllata la presenza
		}

		return id;
	}

	public Compute getCompute(OperationId id) throws OpCodeNotPresentException {

		Compute toReturn = storage.get(id);
```

```
		if(toReturn == null)
			throw new OpCodeNotPresentException();
		else return toReturn;
	}

	public String toString() {
		String out = "len:"+storage.size()+":\n\t";

		Collection<Map.Entry<OperationId, Compute>> entrySet = storage.entrySet();

		for (Map.Entry<OperationId, Compute> entry : entrySet) {

			OperationId id = entry.getKey();

			out = out + ":" + id + ":" + storage.get(id) + ":\n\t";
		}

		return out;
	}


	/**
	 * Restituisce l'identificatore corrispondente
	 * al'opcode <code>comp</code> se l'associazione esiste,
	 * <code>null</code> altrimenti.
	 * @return L'identificatore corrispondente al'opcode
	 * <code>comp</code> se l'associazione esiste,
	 * <code>null</code> altrimenti.
	 * */
	@SuppressWarnings("unchecked")
	private OperationId getOperationId(Compute comp)
	{
		OperationId id = null;

		Class<Compute> compClass = (Class<Compute>) comp.getClass();

		Collection<Map.Entry<OperationId, Compute>> entrySet = storage.entrySet();

		boolean done = false;

		/* scorro le entry per cercare un Compute con Class
		 * uguale a quello del parametro e ne prendo il corrispettivo
		 * OperationId (la key della entry)
		*/
		for (Iterator<Map.Entry<OperationId, Compute>> iterator =  entrySet.iterator();
				!done && iterator.hasNext();)
		{

			Map.Entry<OperationId, Compute> entry =
                                                                (Map.Entry<OperationId, Compute>)
iterator.next();

			Compute opcode = entry.getValue();

			Class<Compute> entryClass = (Class<Compute>) opcode.getClass();

			if(entryClass.equals(compClass)){ //stesso Class

				id = entry.getKey();
				done = true;//id trovato, ho finito
			}

		}

		return id;
	}

}
```

Compiler.java

```
import java.util.Collection;
```

```
import muskel.mdf.Destination;
import muskel.mdf.IdGenerator;
import muskel.mdf.InputPositionId;
import muskel.mdf.InstructionId;
import muskel.mdf.IntegerIdGenerator;
import muskel.mdf.IntegerIdentifier;
import muskel.mdf.MdfGraph;
import muskel.mdf.Mdfi;
import muskel.mdf.OperationId;

import skeleton.Compute;
import skeleton.Farm;
import skeleton.ParCompute;
import skeleton.Pipeline;
import skeleton.SECompute;
/**
 *
 * This is to compile skeletons to MDF graphs
 * @author marcod
 *
 */
public class Compiler {

    private static IdGenerator idGen = new IntegerIdGenerator();

        public static MdfGraph compile(Compute prgm, CodeStorage cs) {
                //System.out.println("Compiling => "+(new PrettyPrint(prgm)).toString());
                if( prgm instanceof Farm) {
                        // this is the trivial case, just compile the workers
                        Compute worker =  ((Farm) prgm).getWorker();
                        MdfGraph w = compile(worker,cs);
                        //System.out.println("Compiled ==> \n"+w.toString());
                        //System.out.println("<==");
                        return w;
                }
                if(prgm instanceof Pipeline) {
                        // this is the composite case
                        // get the two graphs
                        Compute firstStage = ((Pipeline) prgm).getFirstStage();
                        Compute secondStage = ((Pipeline) prgm).getSecondStage();
                        MdfGraph firstGraph = compile(firstStage,cs);
                        MdfGraph secondGraph = compile(secondStage,cs);

                        // link the two
                        InstructionId secondGrFirstInstrId = secondGraph.getInputInstrId();
                        InstructionId firstGrFirstInstrId = firstGraph.getInputInstrId();
                        // ATTENTION: this works only on 1 in 1 out instructions !!! due to the 0
first parameter

                        // this links the two
                        Destination newdest = new Destination(new InputPositionId(new
IntegerIdentifier(0)), secondGrFirstInstrId, Mdfi.NoGraphId);

                        //System.out.println("First stage graph is :: "+firstGraph.toString());
                        //System.out.println("Dest is ::"+newdest);
                        //System.out.println("Setting dest in graph "+firstGraph.toString());

                        InstructionId last = firstGraph.setOutputInstrId(newdest); // should be void
... returns the instrID changed (the last)...
                        //System.out.println("Dest id was "+last+" graph is now ::
"+firstGraph.toString());

                        // now return the new graph: merging the first one and the relocated second
one
                        MdfGraph newgraph = new MdfGraph(Mdfi.NoGraphId);

                        //la prima istruzione del primo grafo
                        Mdfi firstGrFirstInstr = firstGraph.getGraph().get(firstGrFirstInstrId);
                        //l'ultima istruzione del secondo grafo
                        Mdfi secondGrLastInstr =
secondGraph.getGraph().get(secondGraph.getOutputInstrId());

                        //imposto la prima istruzione del primo grafo come istruzione di input del
nuovo grafo,
                        newgraph.setInputInstruction(firstGrFirstInstr);
```

```
                        Collection<Mdfi> iterableFirstGraph = firstGraph.getGraphMdfi();
                        Collection<Mdfi> iterableSecondGraph = secondGraph.getGraphMdfi();

                        //aggiungo al nuovo grafo le istruzioni del primo
                        for (Mdfi mdfi : iterableFirstGraph)
                                        newgraph.addInstruction(mdfi);

                        //aggiungo al nuovo grafo le istruzioni del secondo
                        for (Mdfi mdfi : iterableSecondGraph)
                               newgraph.addInstruction(mdfi);

                        //imposto l'ultima istruzione del secondo grafo come istruzione di output del
nuovo grafo,
                        newgraph.setOutputInstruction(secondGrLastInstr);

                        //System.out.println("Compiled ==> \n"+newgraph.toString());
                        //System.out.println("<==");
                        return newgraph;
                }
                if(prgm instanceof ParCompute) {
                    // this is the case the graph is already provided by the user in the body of the
ParCompute object
                    // actually this is the simplest of the cases handled here:

                    MdfGraph g =  (((ParCompute) prgm).getMdfGraph()).mdfClone();
                    //System.out.println("got ParCompute graph "+g);
                    return g;
                }

                // no other case than a seq code Compute: this is an Object to Object ... must be a
Object[] to Object[]
                MdfGraph seqgraph = new MdfGraph();

                OperationId opcode = cs.store(prgm);

                Destination[] dest1 = {new Destination(new InputPositionId(new IntegerIdentifier(0)),
Mdfi.NoInstrId, Mdfi.NoGraphId)};

                //genero un nuovo id per la nuova mdfi
                InstructionId seqmdfiId = new InstructionId(idGen.generateId());

                boolean seCompute = (prgm instanceof SECompute);

                Mdfi seqmdfi = new Mdfi(seqmdfiId, Mdfi.NoGraphId, opcode, 1, 1, dest1, seCompute);

                seqgraph.setInputInstruction(seqmdfi);
                seqgraph.addInstruction(seqmdfi);
                seqgraph.setOutputInstruction(seqmdfi);

                //System.out.println("Compiled ==> \n"+seqgraph.toString());
                //System.out.println("<==");
                return seqgraph;
        }
}
```

Contract.java

```
/* This is the muskel: a skeleton/RMI based, parallel programming library
Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */
```

```java
/*
 * Created on Oct 21, 2004
 *
 * TODO To change the template for this generated file go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
package muskel;

/**
 * This interface is used to model the performance
 * contract asked to the muskel runtime.
 * This is an interface as multiple contracts
 * can be issued to the muskel run time.
 * <br>
 *
 * @author marcod
 */
public interface Contract {
	// no methods defined there. This is actually a marker interface
}
```

ControlThread.java

```java
/* This is the muskel: a skeleton/RMI based, parallel programming library
Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

package muskel;

import java.net.InetAddress;
import java.net.MalformedURLException;
import java.net.UnknownHostException;
import java.rmi.Naming;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

import muskel.mdf.*;
import muskel.stream.*;

/**
 * Used to check the compiler/storage/pool
 *
 * @author marcod
 *
 */
public class ControlThread implements Runnable {

	public final boolean debug = Manager.debug;
	public boolean Sicuro = true;
	public static final int Port = 12460;
	private CodeStorage cs = null;

	private MdfiPool pool = null;

	private OutputManager osm = null;
```

```
        private InputManager ism = null;

        private String host = "";

        private boolean blockingInStream = false;

        private InetAddress ciccio ;

        public ControlThread(boolean Sicuro, CodeStorage ccs, MdfiPool ppool, InputManager iism,
OutputManager oosm,
                        String hhost) {

                this.Sicuro = Sicuro;
                cs = ccs;
                pool = ppool;
                osm = oosm;
                host = hhost;
                ism = iism;

                /*prendo nota se lo stream e' bloccante e permette
                 * la produzione dei task da parte dei Compute
                 */
                blockingInStream = (ism instanceof BlockingInputManager);

        }

        public void run() {


                // set up the remote node connection
                RemoteInterpreterInterface worker = null;

                System.out.println("Control thread started");

                if(Sicuro){ //Host Sicuro non uso SSL

                        try {
                                ciccio = InetAddress.getByName(host);

                                System.out.println("Got address " + ciccio + " for " + host);
                                String stringa = "rmi://" + host + ":" + RemoteInterpreter.Port
                                        + "/muSkelMDFWorker";
                                if (debug)
                                        System.out.println("Looking up worker on host " + stringa);

                                worker = (RemoteInterpreterInterface) Naming.lookup(stringa);
                                System.out.println("worker is " + worker);
                                System.out.println("Remote worker looked'up");
                                if (debug)
                                System.out.println("worker is " + worker);
                        } catch (MalformedURLException e) {
                                e.printStackTrace();
                        } catch (RemoteException e) {
                                e.printStackTrace();
                        } catch (NotBoundException e) {
                                e.printStackTrace();
                        } catch (UnknownHostException e) {
                                e.printStackTrace();
                        }

                }else{ //Host non sicuro uso SSL

                        try {
                                // Make reference to SSL-based registry
                            Registry reg = LocateRegistry.getRegistry(
                                        host, Port,    new javax.rmi.ssl.SslRMIClientSocketFactory());

                                ciccio = InetAddress.getByName(host);

                                System.out.println("Got address " + ciccio + " for " + host);
                                System.out.println("Looking up worker on host " + host);
                                if (debug)
                                        System.out.println("Looking up worker on host " + host);
```

```
                        // "worker" is the identifier that we'll use to refer
                    // to the remote object that implements the "RemoteInterpreter"
                    // interface
                        worker = (RemoteInterpreterInterface) reg.lookup("muSkelMDFWorker");
                        System.out.println("worker is " + worker);
                        System.out.println("Remote worker looked'up");
                        if (debug)
                        System.out.println("worker is " + worker);
                } catch (RemoteException e) {
                        e.printStackTrace();
                        return;
                } catch (NotBoundException e) {
                        e.printStackTrace();
                } catch (UnknownHostException e) {
                        e.printStackTrace();
                }
        }

        if (worker == null) {
                //TODO non deve uscire, ma cercare un altro worker (da implementare)
                System.out.println("Cannot find worker");
                return;
        }



        System.out.println("RemoteInterpeter found");
        System.out.println("Staging program to remote intepreter... ");
        try {
                worker.setProgram(cs);
        } catch (RemoteException e) {
                System.out.println("setProgram failed on remote node (Exception is " + e
                                + " )");
                e.printStackTrace();
                return;
        }

        System.out.println("code staged");

        System.out.println("Control thread " + this
                        + " getting fireable instruction ... ");

        Mdfi instr = pool.getFireable(); // get an instruction to compute; in
                                                                                // case it blocks

        if (instr == null)
                return; // terminate

        //System.out.println("Control thread " + this + " fetched fireable "
                //      + instr.toString());

        System.out.println("Entering main loop");

        do {
                ComputeResult res = null;
                try { // compute the instruction remotely
                        res = worker.compute(instr);
                } catch (RemoteException e) {
                        // TODO this is where we can trigger the manager

                        // first: put back the not computed mdf instruction
                        pool.insert(instr);
                        System.out.println("PUT BACK THE NOT COMPUTED MDF INSTRUCTION");

                        // then inform the manager TODO
                        // eventually terminate the thread
                        System.out.println("remote worker terminated due RemoteException on
mdfi compute request");
                        //se lo stream supporta i produttori e se l'instr e' un produttore
                        if(blockingInStream && instr.isSeMDFi()){
                                //segnalo allo stream la perdita di un produttore
                                ((BlockingInputManager) ism).removeProducer();
                                System.out.println("Rimosso Produttore");
                        }
                        e.printStackTrace();
```

```
                        return; // this terminates the thread actually
                }

                /* se l'input stream supporta l'aggiunta di task e la compute
                 * calcolata ha side effects aggiungo i task calcolati
                 * allo stream ed gli segnalo che un produttore ha terminato
                 * */

                if(blockingInStream && instr.isSeMDFi()) {

                        BlockingInputManager bism = (BlockingInputManager) ism;
                        SEComputeResult seRes = (SEComputeResult) res;

                        Task[] genTasks = seRes.getGeneratedTasks();

                        if (genTasks != null) {
                                for (Task task: genTasks){
                                        bism.addTask(task.getValue());
                                        System.out.println("TASK VALUE = " + task.getValue());
                                }
                        }
                        bism.removeProducer();
                }

                //gestisco i risultati, caso comune sia a Compute che a SECompute
                OutputToken[] outTokens = res.getOutTokens();

                Destination dest;
                for (OutputToken outTok: outTokens) {// deliver the results
                        System.out.println("Result token : " + outTok + " computed on Worker "
+ host);
                        dest = outTok.getDestination();

                        if (dest.getInstructionId().equals(Mdfi.NoInstrId)) {//controllo se e'
l'ultimo token del grafo
                                try {
                                        osm.deliver(outTok.getTask());//lo spedisco all'output
stream
                                } catch (TokenNotPresentException e) {}//il token non ha un
task

                        } else {
                                //inserisco il token nel pool
                                //System.out.println("Going to store token " +outTok);
                                pool.putToken(outTok);
                                //      System.out.println("Control thread " + this + " stored
token "
                                //              + outTok.toString());
                        }
                }

                // now get another instruction to compute
                instr = pool.getFireable(); // get an instruction to compute; in
                                                                        // case it blocks
                //System.out.println("Control thread " + this + " fetched fireable " + instr);
                if (instr == null) { // this is the case the thread gets an
                                                                // INTERRUPT from the manager,
that verified
                                                                // all threads are blocked on
                        // the empty mdfi pool getFireable ...
                        // therefore statistics are gathered from the associated worker,
                        // they are printed and then the thread terminates
                        System.out.println("Going to retrieve the remote stats ...");
                        try {
                                Stats s = worker.getStats();
                                System.out.println("Retrieved");
                                //System.out.println("Stats: " + s);
                                System.out.println(s);
                        } catch (RemoteException e) {
                                e.printStackTrace();
                        }
                        return; // terminate
                }
        } while (true); // TODO termination
```

```
        }
}
```

DiscoveryService.java

```
/* This is the muskel: a skeleton/RMI based, parallel programming library
Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

/*
 * Created on Mar 26, 2004
 *
 * To change the template for this generated file go to
 * Window - Preferences - Java - Code Generation - Code and Comments
 */
package muskel;

import java.util.Vector;

/**
 * This class is used to maintain a list of available
 * (discovered, actually) processing resources,
 * that is processing elements in the portion of network accessible
 * via multicast messages that currently run a muskel RemoteWorker
 * RMI object. <br>
 * The class uses a DiscoveryThread thread to actually
 * perform remote processing elements discovery.
 * <br>
 * The list of workers maintained by this class
 * is a simple Vector. New workers requested by the ControlThread
 * are provided from the start position. <br>
 * New workers discovered are added at the end.
 * <br>
 * A list of already given worker (to the ControlThread) is also maintained.
 * <br>
 * In case a number nw of processing elements is requested,
 * which is not currently available, the calling thread
 * is actually blocked (wait()) and it is eventually
 * unblocked by the DiscoveryThread communicating
 * that a new resource is available.
 *
 * @author Danelutto Marco
 *
 *
 */
public class DiscoveryService {

        /** this is the vector hosting the hosts discovered but not yes used */
        Vector<String> discoveredWorkers = null;
        /** this is the vector hosting the hosts discovered and already used */
        Vector<String> givenWorkers = null;
```

```
	/**
	 * standard constructor
	 * initializes the vectors and starts the discovery thread
	 */
	public DiscoveryService() {
		if(Manager.debug)
			System.out.println("--> Discovery service creation");

		discoveredWorkers = new Vector<String>();
		givenWorkers = new Vector<String>();

		DiscoveryThread dt = new DiscoveryThread(this);
		dt.setDaemon(true);
		dt.start();

		return;
	}

	/**
	 * used from the StandardEval thread to get a fresh
	 * host to run a worker.
	 * @return the name of the fresh host discovered
	 */
	public synchronized String getNewWorker() {
		while(discoveredWorkers.size()==0) {
			try {
				wait();
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
		String worker = (String) discoveredWorkers.remove(0);
		givenWorkers.add(worker);
		System.out.println("getNewWorker returns "+worker);
		return worker;
	}

	/**
	 * used from within the Standard compute method to get a set of workers
	 * @param nw the number of workers required
	 * @return the vector with the worker
	 */
	public synchronized String [] getNWorkers(int nw) {
		while(discoveredWorkers.size() < nw) {
			try {
				System.out.println("Aspetto di trovare " + nw + " worker");

				wait(10000);//aspetto un pò di tempo fissato

				//se ancora non ho trovato il numero necessario di worker
				if (discoveredWorkers.size() < nw )

					nw = discoveredWorkers.size();//vado avanti con quelli trovati

				System.out.println("Scaduto il timeout, proseguo con " + nw + "
worker");
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
		String [] v = new String [nw];
		for(int i=0; i<nw; i++) {
			v[i] = (String) discoveredWorkers.remove(0);
			givenWorkers.add(v[i]);
		}
		System.out.println("getNWorkers returned a vector with "+nw+" workers");
		return v;
	}

	/**
	 * Used by the DiscoveryThread to place the name of
	 * a fresh host discovered in the
	 * discovered hosts vector.
	 * Checks for presence of that host in both
	 * the fresh and the given vector before adding
```

```
         * as the Discovery thread does not hold the state
         * of already discovered hosts.
         *
         * @param w the name of the host to be added
         */
        public synchronized void addWorker(String w) {
                if(!(discoveredWorkers.contains(w) || givenWorkers.contains(w))) {
                        discoveredWorkers.add(w);
                        System.out.println("addWorker added "+w);
                        notifyAll();
                }
                return;
        }
}
```

DiscoveryService.java

```
/* This is the muskel: a skeleton/RMI based, parallel programming library
Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

/*
 * Created on Mar 26, 2004
 *
 * To change the template for this generated file go to
 * Window - Preferences - Java - Code Generation - Code and Comments
 */
package muskel;

import java.io.IOException;
import java.net.DatagramPacket;
import java.net.InetAddress;
import java.net.MulticastSocket;
import java.net.SocketTimeoutException;

/**
 * The discovery Thread always looks for new workers.
 * Every time a worker is discovered, its name is issued
 * to the Discovery Service that provides storing
 * it properly in the data structure holding all the
 * available processing resources.
 * <br>
 * <p>
 *
 * @author Danelutto Marco
 *
 */
public class DiscoveryThread extends Thread {

        /** the service used to keep track of the workers found */
        DiscoveryService service = null;

        /**
         * constructor
         * @param s the discovery service to be used to
         *                   store the (possibly new) workers found
         */
        public DiscoveryThread(DiscoveryService s) {
```

```
                service = s;
                return;
        }

        /**
         * this is to find workers with the discovery mechanism instead.
         * Assumes that the fastest
         * responders will be the best ones and therefore put them in front of the
         * list.
         * <br>
         * The TTL of the multicast socket used to send
         * discovery messages is currently set to a very large value,
         * in order to be able to reach the larger number of nodes possible. <br>
         * Take into account that usually firewalls of autonomous systems
         * will filter out the discovery messages, being directed
         * to an ephemeral port outside the ports that are usually
         * "open" in most frameworks.
         */
        public void run() {
                try {

//                      final int BUFLEN = 1024;
                        final int BUFLEN = 512;

//                      final int DELAY = 500;   // starts a discovery step every half a second
                        final int DELAY = 5000;       // 5 secs

                        MulticastSocket ms = new MulticastSocket();
                        // set a large TTL, possibly means that a larger set of nodes than those on
the local network
                        // are reached.
                        final int ttl = 150;
                        ms.setTimeToLive(ttl);

                        byte[] discoveryBuffer = PresenceThread.DISCOVERYMESSAGE.getBytes();

                        InetAddress gia = InetAddress.getByName(PresenceThread.multicastGroup);

                        //pacchetto con il messaggio di discovery
                        DatagramPacket discoveryDp =
                                new DatagramPacket(discoveryBuffer, discoveryBuffer.length, gia,
PresenceThread.multicastPort);

                        //pacchetto per la ricezione della risposta
                        DatagramPacket dp = new DatagramPacket(new byte[BUFLEN], BUFLEN);

                        while(true) {
                                //log.debug("New discovery cycle");

                                //send the discovery packet
                                ms.send(discoveryDp);

                                boolean done = true;

                                do {
                                        ms.setSoTimeout(DELAY);
                                        try {

                                                ms.receive(dp);

                                                ms.setSoTimeout(0);
                                                String fullHost = new String(dp.getData(), 0,
dp.getLength());
                                                // patch windows way of giving names to machines with
                                                // DHCP
                                                String[] hostNames = fullHost.split("/");
                                                String host = hostNames[1];

                                                service.addWorker(host);

                                        //      System.out.println("Message received is:
>>"+fullHost+"<<");

                                        } catch (SocketTimeoutException e) {
                                                done = false;
```

```
                                        }
                                } while (done);
                        }
                } catch (IOException e) {
                        e.printStackTrace();
                }
                return;
        }

}
```

IllegalMakeOpCodeException.java

```
/* This is the muskel: a skeleton/RMI based, parallel programming library
Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

package muskel;

public class IllegalMakeOpCodeException extends Exception {

        private static final long serialVersionUID = 1958030774748889157L;

        public IllegalMakeOpCodeException() {
                super();
                        }

        public IllegalMakeOpCodeException(String message) {
                super(message);
        }

        public IllegalMakeOpCodeException(String message, Throwable cause) {
                super(message, cause);
        }

        public IllegalMakeOpCodeException(Throwable cause) {
                super(cause);

        }

}
```

InstructionTag.java

```
/* This is the muskel: a skeleton/RMI based, parallel programming library
Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
```

```
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

package muskel;

import java.io.Serializable;
import muskel.mdf.GraphId;
import muskel.mdf.InstructionId;
import muskel.mdf.Mdfi;

/**
 * This is used to hash the instructions in the pool
 *
 * @author marcod
 *
*/
public class InstructionTag implements Serializable {

	private static final long serialVersionUID = 1L;

	InstructionId it = Mdfi.NoInstrId;
	GraphId gt = Mdfi.NoGraphId;

	public InstructionTag(InstructionId i, GraphId g) {
		it = i;
		gt = g;
	}

	public InstructionId getInstrId() {
		return it;
	}

	public GraphId getGraphId() {
		return gt;
	}

	public String toString() {
		return (it+","+gt);
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (!(obj instanceof InstructionTag))
			return false;
		final InstructionTag other = (InstructionTag) obj;
		if (gt == null) {
			if (other.gt != null)
				return false;
		} else if (!gt.equals(other.gt))
			return false;
		if (it == null) {
			if (other.it != null)
				return false;
		} else if (!it.equals(other.it))
			return false;
		return true;
	}

	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result + ((gt == null) ? 0 : gt.hashCode());
		result = prime * result + ((it == null) ? 0 : it.hashCode());
		return result;
	}
}
```

Manager.java

```
/* This is the muskel: a skeleton/RMI based, parallel programming library
Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

package muskel;

import java.net.InetAddress;
import java.net.UnknownHostException;
import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintStream;
import java.util.ArrayList;
import java.util.List;
import muskel.mdf.*;
import muskel.stream.BlockingInputManager;
import muskel.stream.InputManager;
import muskel.stream.OutputManager;
import skeleton.Compute;


/**
 * takes care of a program execution request, with contract (#PEs).
 * A manager object should be instantiated to run muskel programs.
 * The manager is to be given the program,
 * the input and output stream and a performance contract.
 * The manager can be asked to "compute" the program
 * by invoking the homonymous method.
 *
 * @author marcod
 *
 */
public class Manager {

        public static final boolean debug = false;
        boolean localDebug = false;
        public static String COMMENT_SEPARATOR = "#";
        public static ThreadGroup thGr = new ThreadGroup("controlThreads");
        public boolean Sicuro;
        Thread [] controlThreads ;
        Compute program = null;
        CodeStorage cs = null;
        Contract pc = null;
        DiscoveryService ds = null;
        MdfiPool pool = null;
        InputManager ism = null;
        OutputManager osm = null;


        /**
         * the evaluator schedule policy
         */
        /**
         * the log file
         */
        PrintStream logFile = null;

        InstructionId firstInstructionId = Mdfi.NoInstrId;

        private boolean blockingInStream = false;
```

```
IdGenerator idGen;

 // this will host the compiled graph (plus user provided mdfis)
MdfGraph theGraph = null;

/** This constructs the manager.
 * @param ppgm the program to be computed. It must implement the Compute interface
 * @param iism the input manager, a class implementing
 *             the InputManager Interface that requires the methods boolean
 *             hasNext() (telling whether or not the input stream has further items)
 *             and an Object next() method that returns the next item of the input stream
 * @param oosm the output manager. Its void deliver(Object res)
 *             method is called each time a new result is available
 */
public Manager(Compute ppgm, InputManager iism, OutputManager oosm) {

        program = ppgm;
        cs = new CodeStorage();

        idGen = new IntegerIdGenerator();

        /*Se il Compute e' disponibile lo compilo.
         * Altrimenti lo faro' quando sarÃ  reso disponibile tramite la setProgram.
         * (e' il caso dei ParCompute)
         */
        if(ppgm != null) { // Il Compute da compilare e' disponibile
                theGraph = Compiler.compile(program,cs);
                //System.out.println("Compiled graph is "+theGraph);
        }

        // set up discovery
        ds = new DiscoveryService();

        pool = new MdfiPool();
        ism = iism;
        osm = oosm;

        /*prendo nota se lo stream e' bloccante
         *e permette la produzione dei task da parte dei Compute
        */
        blockingInStream = (ism instanceof BlockingInputManager);
}

/** this constructs the manager without giving all the parameters.
 * It is used when user defined MDF graphs are
 * supplied in the program via the ParCompute class.
 * Later on the missing parameters are set up with the proper accessor methods.
 */
public Manager() {
        this(null, null, null);
}

/** Accessor method to set program.
 *      Also compile the <code>pgm</code> mdfgraph
 * @param pgm the program to be computed by the manager
 */
public void setProgram(Compute pgm) {
        program = pgm;

        theGraph = Compiler.compile(program, cs);
}

/** accessor method to set the input manager
 * @param im the input manager
 */
public void setInputManager(InputManager im) {
        ism = im;
}

/** accessor method to set the output manager
 * @param om the output manager
 */
public void setOutputManager(OutputManager om) {
        osm = om;
```

```
}
/** the method is used to set up the performance contract with the manager.
 * The user asks a given parallelism degree by stating it in the contract.
 * AT the moment, the request is satisfied all or nothing.
 * IF less than the requested number of remote intepreters are
 * found, then the manager blocks on the compute method.
 * @param c the performance contract to be satisfied by the manager
 */
public void setContract(Contract c) {
	pc = c;
}


/**
 * this method returns a pointer to the CodeStorage.
 * It is used to implement user defined macro data flow instructions.
 * User defined Compute code is used as the MDFi opcode.
 * The Compute code has to be stored into the CodeStorage,
 * in such a way it can be retrieved (with an integer pointer)
 * when the MDFi has to be executed.
 * @return the current CodeStorage pointer
 */
public CodeStorage getCodeStorage() {
	return cs;
}


/** Sets a new Compute code in the CodeStorage,
 * returning the OperationId in the CodeStorage.
 * If the CodeStorage doesn't exist, returns Mdfi.NoOpCode
 * @param opc The Compute program to be stored in the CodeStorage
 * @return The OperationId in the CodeStorage.
 *                     Mdfi.NoOpCode if the CodeStorage not exists.
 */
public OperationId storeOpCode(Compute opc) {

	OperationId opcode;

	try {
		opcode = cs.store(opc);

	} catch (NullPointerException e) {
		System.out.println("storeOpCode with an empty CodeStorage");

		opcode = Mdfi.NoOpCode;
	}

	return opcode;
}


/**
 * Gives the programmer the ability to retrieve
 * the compiled graph of a skeleton program.
 * This can be used to merge with ad hoc data flow graphs,
 * provided by the user, that express parallelism exploitation
 * patterns not provided within the native skeletons.
 *
 * @param pgm The program to be mdf compiled
 * @return The compiled graph of pgm.
 */
public MdfGraph newGraph(Compute pgm) {
	MdfGraph graph = Compiler.compile(pgm,cs);
	return graph;
}


/** the method starts the parallel computation of
 * the program stored in the manager.
 * A number of remote processing elements is recruited,
 * according to the performance contract passed to the manager
 * then an according number of control threads are started.
 * Eventually, they cooperate to compute the program on the
 * distributed network of MDF interpreters
 */
```

```
	public void compute() {
		if(program==null) {
			System.out.println("computing a null program");
			return;
		}
		if(ism== null) {
			System.out.println("computing program without input manager");
			return;
		}
		if(osm== null) {
			System.out.println("computing program without output manager");
			return;
		}

		/*
		 *  set up a proper pool of ControlThreads accessing
		 *  he MdfiPool and delivering fireable
		 *  MDFis to remote interpreters in a loop
		*/
		long t0 = System.currentTimeMillis();
		System.out.println("Manager: start");

		int nw = ((ParDegree) pc).getParDegree();  // look for the contract
		List<String> hostsicuri = new ArrayList<String>();
		try {
			FileInputStream fis = new FileInputStream("./hosts.conf");
			BufferedReader br = new BufferedReader(new InputStreamReader(fis));


			try {
				String line;
				/*"indirizzio IP standard o hostname"*/

			while ((line = br.readLine()) != null)	{

				line = line.trim();//elimino gli spazi

				if(line.isEmpty() || line.startsWith(COMMENT_SEPARATOR))//linea vuota
o commenti
					continue;

				// ottiene un'istanza di InetAddress a partire dall'indirizzo IP
				InetAddress inetAddress = InetAddress.getByName(line);
				// risolve il nome
				String hostname = inetAddress.getHostName();
				String hostaddress = inetAddress.getHostAddress();
				// stampa in console l'indirizzo risolto
				System.out.println("host name: " + hostname + " per l'indirizzo " +
hostaddress);

				//controllo se è un ip raggiungibile o corretto
				if(line.equals(inetAddress.getHostAddress())){
					//aggiungo alla lista
					hostsicuri.add(hostaddress);
				}
			}
			//stampo in console gli host sicuri
			System.out.println("Host Sicuri: " + hostsicuri);
			} catch (IOException e) {
				e.printStackTrace();
			}
		} catch (FileNotFoundException e) {
			System.out.println("File di config non trovato");
		}



		String [] machines = ds.getNWorkers(nw);     // search machines
		// may be less than nw, risolta con Best Effort
		List<String> hostsicuri2 = new ArrayList<String>();
		List<String> hostnonsicuri = new ArrayList<String>();
		for(int i=0; i<machines.length; i++) {
			if (hostsicuri.contains(machines[i])){
				hostsicuri2.add(machines[i]);
```

```
                        System.out.println("Host sicuro trovato: " +  machines[i]);
                    } else {
                        hostnonsicuri.add(machines[i]);
                        System.out.println("Host non sicuro trovato " + machines[i]);
                    }
                }

                muskel.ControlThread [] ct  = new muskel.ControlThread[machines.length];

                // actual threads here (to use groups)
                controlThreads = new Thread[machines.length];
                // start threads in a thread group ... to control termination with active count

                // for each machine start a control thread
                for(int i=0; i<machines.length; i++) {
                    try {
                        if (hostsicuri.contains(machines[i])){
                            Sicuro=true; // host sicuro
                        }else{
                            Sicuro=false; // host non sicuro
                        }
                            ct[i] = new muskel.ControlThread(Sicuro,cs, pool, ism, osm,
(InetAddress.getByName(machines[i])).getHostName());
                    } catch (UnknownHostException e) {
                        e.printStackTrace();
                    }
                    controlThreads[i] = new Thread(thGr,ct[i],"controlThread"+i);
                    controlThreads[i].start();
                }

                long t1 = System.currentTimeMillis();
                long elapsed = (t1 > t0 ? t1 - t0 : Long.MAX_VALUE - t0 + t1);
                System.out.println("Total elapsed time (init control threads): "+elapsed+" msec with
"+machines.length+" PEs");

                System.out.println("Manager: thread started");

                GraphId newGid;

                // now start fetching input tasks and delivering them to the pool with the associated
graph
                while(ism.hasNext()) {

                    // fetch an input task
                    Task task = new Task(ism.next());

                    System.out.println("Manager: task fecthed "+task);

                    int numSeCompute = theGraph.getNumSECompute();

                    /*segnalo allo stream la presenza di nuovi produttori
                     *se lo stream li supporta e se il grafo ha
                     *istruzioni che producono task
                    */
                    if(blockingInStream &&  numSeCompute> 0)
                        ((BlockingInputManager) ism).addProducers(numSeCompute);

                    //set graphId
                    MdfGraph graph = theGraph.mdfClone();//clone a new graph for each token
                    //System.out.println("Cloned graph is "+graph);

                    firstInstructionId = graph.getInputInstrId(); // this is the entry point

                    newGid = new GraphId(idGen.generateId());

                    graph.setGid(newGid); // instantiate the graph

                    // add the input token
                    OutputToken outToken = new OutputToken(task, new Destination(
                                                                    new InputPositionId(new
IntegerIdentifier(0)),firstInstructionId, newGid));

                    pool.insert(graph);// put the graph in the pool
```

```
                        //System.out.println(">>>>> "+graph);

                        pool.putToken(outToken);

                        System.out.println("Manager: graph inserted in pool");

                }

                System.out.println("Manager: task insertion terminated");

                long t2 = System.currentTimeMillis();
                elapsed = (t2 > t0 ? t2 - t0 : Long.MAX_VALUE - t0 + t2);
                System.out.println("Total elapsed time (tasks inserted): "+elapsed+" msec with
"+machines.length+" PEs");
                // now wait the control threads ...
                //System.out.println("Thread Group has "+thGr.activeCount()+" active Threads");
                int blocked = 0;

                while((/*(thGr.activeCount() >= machines.length) &&*/ (blocked != machines.length)))
{

                        try {
                                Thread.sleep(100);
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }  // just for releasing the CPU

                        //System.out.println("ACTIVE COUNT "+thGr.activeCount());
                        blocked = 0; // added to avoid accumulation ...

                        for(int i=0;i<machines.length; i++) {
                        //      System.out.println(">>> "+controlThreads[i]+"
("+controlThreads[i].getState()+")");
                                if(((controlThreads[i].getState() == Thread.State.WAITING ) ||
                                                (controlThreads[i].getState() == Thread.State.TERMINATED
))) {
                                        blocked++;
                                }
                        }
                        //System.out.println("La variabile blocked vale: " + blocked);

                        if(blocked == machines.length) {
                                // terminate threads
                                //TODO after gathering statistics ...
                                for(int i=0; i<machines.length; i++)
                                        controlThreads[i].interrupt();
                        }
                }

                System.out.println("All threads blocked ...");

                for(int i=0; i<machines.length; i++) {
                        try {
                                System.out.println("Joining thread "+i);

                                controlThreads[i].join();

                                System.out.println("Thread "+i+" joined");
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                }

                long t3 = System.currentTimeMillis();
                elapsed = (t3 > t0 ? t3 - t0 : Long.MAX_VALUE - t0 + t3);
                System.out.println("Total elapsed time: "+elapsed
                                                        +" msec with "+machines.length+" PEs");
                System.exit(0); // kills also the discovery service thread
        }

}
```

MdfiPool.java

```
/* This is the muskel: a skeleton/RMI based, parallel programming library
Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

package muskel;

import java.util.HashMap;
import java.util.Vector;
import muskel.mdf.*;

/**
 * This is used to store Mdfi from graphs.
 *
 * @author marcod
 *
 */
public class MdfiPool {

	public final boolean debug = Manager.debug;

	HashMap<InstructionTag,Mdfi> pool = null;

	Vector<InstructionTag> fireableInstructions = null;


	public MdfiPool() {

		pool = new HashMap<InstructionTag,Mdfi>();
		fireableInstructions = new Vector<InstructionTag>();

	}

	/**
	 * Inserts <code>instruction</code> into the pool
	 * */
	public synchronized void insert(Mdfi instr) {

		InstructionTag iTag = new InstructionTag(instr.getInstrId(),instr.getGraphId());

		pool.put(iTag, instr);

		if(instr.isFireable())
			fireableInstructions.add(iTag);

		notifyAll();//notifico anche se l'istr non Ã¨ fireable perchÃ¨ c'Ã¨ la extract
	}

	/**
	 * Inserts the <code>graph</code> mdfi into the pool
	 * */
	public synchronized void insert(MdfGraph graph) {

		InstructionTag iTag;

		for(Mdfi instr: graph.getGraphMdfi()) {

			iTag = new InstructionTag(instr.getInstrId(),instr.getGraphId());

			pool.put(iTag, instr);
			if(instr.isFireable())
				fireableInstructions.add(iTag);
```

```
                        //notifico anche se l'istr non Ã¨ fireable perchÃ¨ c'Ã¨ la extract
                        notifyAll();
                }
        }

        /**
         * Extracts the mdfi matching to <code>instrTag</code> from the pool.
         * */
        public synchronized Mdfi extract(InstructionTag instrTag) {
                while(pool.isEmpty()) {
                        try {
                                wait();
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                }

                Mdfi instr = pool.remove(instrTag);

                return instr;
        }


        /**
         * Extracts the mdfi having <code>instr</code> and
         * <code>graph</code> as InstructionId and GraphId.
         * */
        public synchronized Mdfi extract(InstructionId instr, GraphId graph) {

                return extract(new InstructionTag(instr,graph));
        }

        /**
         * Returns the first fireable mdfi (according to a FIFO policy)
         * @return Returns the first fireable mdfi
         * */
        public synchronized Mdfi getFireable() {
                while(fireableInstructions.isEmpty()) {
                        try {
                                //System.out.println("getFireable: empty Fireable instruction
pool\n"+pool);
                                wait();
                        } catch (InterruptedException e) { // if you get there, you're terminating
                                return null;
                        }
                }
                InstructionTag istrTag = fireableInstructions.remove(0); // this is FIFO

                Mdfi instr = pool.remove(istrTag);
                //System.out.println("getFireable: got "+instr);

                return instr;
        }

        /**
         * Inserisce outToken nella rispettiva mdfi (specificata in outToken stesso)
         * @throws IllegalArgumentException Se outToken non contiene un task
         * */
        public synchronized void putToken(OutputToken outToken) {

                Task task;

                try {
                        task = outToken.getTask();
                } catch (TokenNotPresentException e) {//il token non contiene un task

                        throw new IllegalArgumentException("Token uninitialized");
                }

                Destination dest = outToken.getDestination();

                InstructionTag instrTag = new InstructionTag(dest.getInstructionId(),
dest.getGraphId());
```

```
                //System.out.println("Putting input token "+outToken.toString()+" InstrutionId is
"+instrTag.getInstrId()+" GraphId is "+instrTag.getGraphId());

                //prendo la mdfi dal pool
                Mdfi instr = pool.get(instrTag);

                /*inserisco il task di outToken nel vettore di input di instr
                 * (nella posizione specificata in outToken)
                 */
                instr.storeToken(dest.getInputPositionId(), task);

                //controllo se l' instr sia diventata fireable con in nuovo token
                if(instr.isFireable()) {
                        fireableInstructions.add(instrTag);
                        //System.out.println("putToken: added "+instrTag);

                        notifyAll();//notifico i ControlThread in attesa di istruzioni fireable
                }

                return;
        }

        public String toString() {
                String p = "Pool ("+pool.size()+" instr, "+fireableInstructions.size()+"
fireable):\n";

                for(Mdfi instr: pool.values())
                        p = p + instr.toString() + "\n";

                p = p + "\n";

                for(InstructionTag instrTag :fireableInstructions)
                        p = p + instrTag.toString();

                p = p + "\nENDPOOL\n";
                return p;
        }

}
```

NonFireableInstructionException.java

```
/* This is the muskel: a skeleton/RMI based, parallel programming library
Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

package muskel;

public class NonFireableInstructionException extends Exception {

        private static final long serialVersionUID = -2051048925995654504L;

        public NonFireableInstructionException() {
                super();
        }

        public NonFireableInstructionException(String message) {
                super(message);
        }

        public NonFireableInstructionException(String message, Throwable cause) {
```

```
                super(message, cause);
        }

        public NonFireableInstructionException(Throwable cause) {
                super(cause);
        }
}
```

OpCodeNotPresentException.java

```
/* This is the muskel: a skeleton/RMI based, parallel programming library
Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

package muskel;

public class OpCodeNotPresentException extends Exception {

        private static final long serialVersionUID = -7696424837110314635L;

        public OpCodeNotPresentException() {
                super();
        }

        public OpCodeNotPresentException(String message) {
                super(message);
        }

        public OpCodeNotPresentException(String message, Throwable cause) {
                super(message, cause);
        }

        public OpCodeNotPresentException(Throwable cause) {
                super(cause);
        }

}
```

ParDegree.java

```
/* This is the muskel: a skeleton/RMI based, parallel programming library
Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */
```

```java
/*
 * Created on Oct 21, 2004
 *
 * TODO To change the template for this generated file go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
package muskel;

/**
 * This is a contract that can be issued to a manager,
 * specifying the amount of processing elements that have
 * to be used to implement the manager controlled computation.
 *
 * @author marcod
 *
 */
public class ParDegree implements Contract {

	/** this is the actual parallelism degree wanted for this contract
	 *
	 */
	private int pardegree = 1;

	/**
	 * constructor: sets up the initial value of the contract
	 * @param nw the parallelism degree to be sustained by this contract
	 */
	public ParDegree(int nw) {
		pardegree = nw;
	}

	/**
	 * allows the current implementation of the contract to be inspected
	 * @return the current parallelism degree
	 */
	public int getParDegree() {
		return pardegree;
	}
	/**
	 * sets the parallelism degree. Used to change the value
	 * @param nw the new vaule of the parallelism degree
	 * TODO va registrato un handler per avvertire il
	 * manager di un cambiamento nel contratto.
	 * occorre anche definirne il tipo ...
	 */
	public void setParDegree(int nw) {
		pardegree = nw;
	}
}
```

PresenceThread.java

```
/* This is the muskel: a skeleton/RMI based, parallel programming library
   Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

   This program is free software; you can redistribute it and/or
   modify it under the terms of the GNU General Public License
   as published by the Free Software Foundation; either version 2
   of the License, or (at your option) any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
```

```
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

/*
 * Created on Mar 24, 2004
 *
 * To change the template for this generated file go to
 * Window - Preferences - Java - Code Generation - Code and Comments
 */
package muskel;

import java.io.IOException;
import java.net.DatagramPacket;
import java.net.InetAddress;
import java.net.MulticastSocket;

/**
 * This is the thread run by the RMI RemoteWorker server to answer the request for presence
 * issued by other StandardEval components.<br>
 * It listens to a Multicast group and answers with the name of the machine where it
 * is running. The asking thread can therefore manage a new worker at that machine.
 * @author Danelutto Marco
 *
 */
public class PresenceThread extends Thread {

    /** this is the multicast group of the muskel package discovery service */
    public static final String multicastGroup = "236.7.8.99";
    /** this is the port of the muskel discovery service */
    public static final int multicastPort = 54321;
    /** this is the muskel discovery message
     * (ASCII protocol: send discovery message,
     * receive name of the machine */
    public static final String DISCOVERYMESSAGE = "muskelDiscoveryMessage";

    /**
     * the actual thread body: runs forever.
     * Waits for a discovery message. Answers with a message hosting
     * the host name where this server is being run
     */
    public void run() {
       try {

                MulticastSocket ms  =  new MulticastSocket(multicastPort);
                InetAddress gia     =  InetAddress.getByName(multicastGroup);
                ms.joinGroup(gia);

                final int MAXBUF = 512;
                byte[] buffer = new byte[MAXBUF];

                DatagramPacket dp = new DatagramPacket(buffer,MAXBUF);

                System.out.println("Entering the main loop");

                while (true) {

                        ms.receive(dp);

                        //System.out.println("Received discovery message from
"+dp.getAddress().toString());
                        // discovery protocol requires a "DISCOVERY" string here

                        String message = new String(dp.getData(),0,dp.getLength());

                        //System.out.println("Message received is: >>"+message+"<<");

                        if(message.compareTo(DISCOVERYMESSAGE) == 0) {
                                // answer with my address
                                InetAddress myIa = InetAddress.getLocalHost();
                                String myName = myIa.toString(); // myIa.getHostName();
                                byte [] myNameBuf = myName.getBytes();
```

```
                                dp.setData(myNameBuf);

                                ms.send(dp); // back to the sender

                                //System.out.println("Sent answer back to
"+dp.getAddress().toString()+" = "+myName);
                        }

                        //log.debug("New receive buffer");
                        dp.setData(new byte[MAXBUF]);//nuovo buffer

                }

        } catch (IOException e) {
                e.printStackTrace();
        }
    }
}
```

PrettyPrint.java

```
/* This is the muskel: a skeleton/RMI based, parallel programming library
Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

package muskel;

import skeleton.Compute;
import skeleton.Farm;
import skeleton.Pipeline;

public class PrettyPrint {

        Compute program = null;

        public PrettyPrint(Compute pgm) {
                program = pgm;
        }

        public String toString() {
                String result = "";
                if(program instanceof Farm) {
                        Compute worker = ((Farm) program).getWorker();
                        String workerString = (new PrettyPrint(worker)).toString();
                        result = " Farm("+workerString+" )";
                        return result;
                }
                if(program instanceof Pipeline) {
                        Compute stage1 = ((Pipeline) program).getFirstStage();
                        Compute stage2 = ((Pipeline) program).getSecondStage();
                        String stage1String = (new PrettyPrint(stage1)).toString();
                        String stage2String = (new PrettyPrint(stage2)).toString();
                        return ("("+stage1String+ " | "+ stage2String+")");
                }
                return (" "+program.toString()+" ");
        }
}
```

RemoteInterpreter.java

```
/* This is the muskel: a skeleton/RMI based, parallel programming library
Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

package muskel;

import java.net.InetAddress;
import javax.rmi.ssl.SslRMIServerSocketFactory;
import javax.rmi.ssl.SslRMIClientSocketFactory;
import java.net.UnknownHostException;
import java.net.URLClassLoader;
import java.net.URL;
import java.rmi.AccessException;
import java.rmi.RMISecurityManager;
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import muskel.mdf.ComputeResult;
import muskel.mdf.Mdfi;


public class RemoteInterpreter extends UnicastRemoteObject implements
                RemoteInterpreterInterface {

        private static final long serialVersionUID = 1L;
        public static boolean Sicuro = false;

        public static final int Port = 12460;

        CodeStorage cs = null;

        // this is for statistics
        int mdfiCount; // the number of MDFi computed

        long timeCompute; // the total time spent computing them

        long maxTimeCompute; // the maximum time spent computing a MDFi

        long minTimeCompute; // the minimum time spent computing a MDFi

        boolean firstTime;


        protected RemoteInterpreter() throws RemoteException {
                super();
                mdfiCount = 0;
                timeCompute = 0;
                firstTime = true;
                System.out.println("Created remote interpreter");
                String url = "http://192.168.123.116/marcod/Muskel/";
                // setting up the proper classloader (URL, at the moment)
                ClassLoader previous = Thread.currentThread().getContextClassLoader();
                // Create a class loader using the URL as the codebase
                // Use previous as parent class loader to maintain current visibility
                ClassLoader current = null;
                try {
                        current = URLClassLoader.newInstance(new URL[] { new URL(url) },
                                        previous);
```

```
        } catch (java.net.MalformedURLException e) {
                System.out.println(e + " while setting classloader in RemoteInterpreter");
                e.printStackTrace();
        }
        Thread.currentThread().setContextClassLoader(current);
        System.out.println("url classloader appended");
        return;
}
protected RemoteInterpreter(int Port, SslRMIServerSocketFactory ServerSocket,
                SslRMIClientSocketFactory ClientSocket ) throws RemoteException {
        super();
        mdfiCount = 0;
        timeCompute = 0;
        firstTime = true;
        System.out.println("Created remote interpreter");
        String url = "http://192.168.123.116/marcod/Muskel/";
        // setting up the proper classloader (URL, at the moment)
        ClassLoader previous = Thread.currentThread().getContextClassLoader();
        // Create a class loader using the URL as the codebase
        // Use previous as parent class loader to maintain current visibility
        ClassLoader current = null;
        try {
                current = URLClassLoader.newInstance(new URL[] { new URL(url) },
                                previous);
        } catch (java.net.MalformedURLException e) {
                System.out.println(e + " while setting classloader in RemoteInterpreter");
                e.printStackTrace();
        }
        Thread.currentThread().setContextClassLoader(current);
        System.out.println("url classloader appended");
        return;
}

// this is the part executing the interpreter
public void setProgram(CodeStorage ccs) throws RemoteException {
        //TODO rifiutare l'esecuzione se l'interprete e' occupato
        //per evitare la sovrapposizione del nuovo codestorage
        //con quello corrente

        try {
                cs = ccs;
                System.out.println("CodeStorage initiated");
                mdfiCount = 0;
                timeCompute = 0;
                firstTime = true;
        } catch (Exception e) {
                System.out.println("Exception " + e + " while loading code");
                e.printStackTrace();
        }
}

public Stats getStats() {
        Stats s = new Stats();
        s.setMdfiNo(mdfiCount);
        s.setAverageTaskTc(timeCompute / mdfiCount);
        s.setMinTimeCompute(minTimeCompute);
        s.setMaxTimeCompute(maxTimeCompute);
        try {
                s.setWorkerName(InetAddress.getLocalHost().getHostName());
        } catch (UnknownHostException e) {
                e.printStackTrace();
        }
        return s;
}

public ComputeResult compute(Mdfi instr) throws RemoteException {
        if (cs == null) {
                System.out.println("compute request with null codestorage");
                return null;
        } else {
                ComputeResult result = null;
                try {
                        //System.out.println("going to compute:" + instr);
                        long t0 = System.currentTimeMillis();
```

```
                        result = instr.compute(cs);
                //      System.out.println("Computed:" + result.toString());
                        long t1 = System.currentTimeMillis();
                        long elapsed = (t1 >= t0 ? t1 - t0 : Long.MAX_VALUE - t0 + t1);
                        timeCompute += elapsed;
                        if (firstTime) {
                                maxTimeCompute = elapsed;
                                minTimeCompute = elapsed;
                                firstTime = false;
                        } else {
                                if (elapsed > maxTimeCompute)
                                        maxTimeCompute = elapsed;
                                if (elapsed < minTimeCompute)
                                        minTimeCompute = elapsed;
                        }
                        mdfiCount++;
                } catch (NonFireableInstructionException e) {
                        e.printStackTrace();
                        result = null; // TODO cosi' non si vedono gli errori da remoto
                        // ...
                } catch (OpCodeNotPresentException e) {
                        e.printStackTrace();
                        result = null; // TODO cosi' non si vedono gli errori da remoto
                        // ...
                } catch (Exception e) {
                        System.out.println("Exception " + e + " while computing mdfi");
                        e.printStackTrace();
                }

                return result;
        }
}

/**
 * @param args
 */

public static void main(String[] args) throws RemoteException {
        int port = Port;
        if (args.length != 0){
                if (args[0].equals("sicuro"))
                        Sicuro = true;
                else
                        port = Integer.parseInt(args[0]);
        }
        System.out.println("=========================================\n");
        try {
                System.out
                                .println("Working on host: " + InetAddress.getLocalHost());
        } catch (UnknownHostException e2) {
                e2.printStackTrace();
        }
        System.out.println("JRE Version:          "
                        + System.getProperty("java.version"));
        System.out.println("OS Information:       "
                        + System.getProperty("os.name") + " "
                        + System.getProperty("os.version") + " "
                        + System.getProperty("os.arch"));
        System.out.println("User Login:           "
                        + System.getProperty("user.name"));
        System.out.println("=========================================\n");
        if (System.getProperty("java.version").compareTo("1.6") < 0) {
                System.err.println("!!!WARNING: Use JDK version 1.6 or higher!!!");
                System.exit(-1);
        }
        if (args.length != 0) {
                if (args[0].equals("-security")) {
                        RMISecurityManager rmiSM = new RMISecurityManager();
                        System.setSecurityManager(rmiSM);
                }
        }

        if (Sicuro){

                RemoteInterpreterInterface worker = new RemoteInterpreter();
```

```
                        System.out.println("Worker created ..");
                        Registry reg = null;
                        try {
                                reg = LocateRegistry.createRegistry(port); // or I'm the first one
                                // running on this
                                // server
                        } catch (RemoteException e) {
                                e.printStackTrace();
                                System.exit(-1);//port already in use, exit.
                        }

                        System.out.println("Register created on port " + port);
                        // now register the object
                        try {
                                String workerName = "muSkelMDFWorker";
                                reg.rebind(workerName, (Remote) worker);
                                System.out.println(worker);
                        } catch (AccessException e1) {
                                e1.printStackTrace();
                        } catch (RemoteException e1) {
                                e1.printStackTrace();
                        }

                }else{

                        SslRMIServerSocketFactory ssf = new SslRMIServerSocketFactory();
                        SslRMIClientSocketFactory csf = new SslRMIClientSocketFactory();
                        RemoteInterpreterInterface worker = new RemoteInterpreter(port,ssf,csf);
                        System.out.println("Worker with SSL created ..");
                        Registry reg = null;
                        try {
                                 // Create SSL-based registry
                            reg = LocateRegistry.createRegistry(port,
                                  new  javax.rmi.ssl.SslRMIClientSocketFactory(),
                                new javax.rmi.ssl.SslRMIServerSocketFactory());
                                // or I'm the first one
                                // running on this
                                // server
                        } catch (RemoteException e) {
                                e.printStackTrace();
                                System.exit(-1);//port already in use, exit.
                        }
                        System.out.println("Register created on port " + port);
                        // now register the object
                        try {
                                String workerName = "muSkelMDFWorker";
                                reg.rebind(workerName, (Remote) worker);
                                System.out.println(worker);
                        } catch (AccessException e1) {
                                e1.printStackTrace();
                        } catch (RemoteException e1) {
                                e1.printStackTrace();
                        }catch (Exception e1){
                                System.out.println("RemoteInterpreter err: " + e1.getMessage());
                                e1.printStackTrace();
                        }
                }
                System.out.println("muskel Worker registered ...");
                // now start the presence thread ...
                PresenceThread pt = new PresenceThread();
                pt.start();
                System.out.println("Presence thread started ...");
        }

}
```

RemoteInterpreterInterface.java

```
/* This is the muskel: a skeleton/RMI based, parallel programming library
Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

This program is free software; you can redistribute it and/or
```

```
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

package muskel;

import java.rmi.Remote;
import java.rmi.RemoteException;
import muskel.mdf.ComputeResult;
import muskel.mdf.Mdfi;;

public interface RemoteInterpreterInterface extends Remote {

	// used to store program at remote nodes, initially
	public void setProgram(CodeStorage cs) throws RemoteException;

	// used to compute remotely a MDF instruction
	public ComputeResult compute(Mdfi instruction) throws RemoteException;

	// used to gather remote node statistics
	public Stats getStats() throws RemoteException;

}
```

Stats.java

```
/* This is the muskel: a skeleton/RMI based, parallel programming library
Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

package muskel;

/**
 * this is the record used to report statistics from the remote nodes.
 * It is sent when the control thread associated to the remote node
 * terminates due to termination of the distributed MDF interpreter
 * (no more fireable instructions, no more tokens on the input stream)
 *
 */
import java.io.Serializable;

/**
 * @author marcod
 *
 */

public class Stats implements Serializable {

	private static final long serialVersionUID = 4438873568941147196L;

	int mdfiNo; // number of instructions executed since last stat reset
	long averageTaskTc; // average number of milliseconds spent in evaluating a task
	long minTimeCompute; // the min time spent computing a task
```

```
        long maxTimeCompute; // the max time spent computing a task
        String workerName;

        public void setMdfiNo(int i) { mdfiNo = i; }
        public int getMdfiNo() { return mdfiNo; }

        public void setAverageTaskTc(long t) { averageTaskTc = t; }
        public long getAverageTaskTc() { return averageTaskTc; }

        public void setMinTimeCompute(long t) { minTimeCompute = t; }
        public long getMinTimeCompute() { return minTimeCompute; }

        public void setMaxTimeCompute(long t) { maxTimeCompute = t; }
        public long getMaxTimeCompute() { return maxTimeCompute; }

        public void setWorkerName(String s) { workerName = s; }
        public String getWorkerName() { return workerName; }

        public String toString() {
                String r = "stats for "+workerName + ": ";
                r = r + " mdfiNo=" + (new Integer(mdfiNo)).toString();
                r = r + " avgTc=" + (new Long(averageTaskTc)).toString();
                r = r + " minTc=" + (new Long(minTimeCompute)).toString();
                r = r + " maxTc=" + (new Long(maxTimeCompute)).toString();
                return r;
        }
}
```

TokenNotPresentExcception.java

```
/* This is the muskel: a skeleton/RMI based, parallel programming library
Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

package muskel;

public class TokenNotPresentException extends Exception {


        private static final long serialVersionUID = 1L;

        public TokenNotPresentException() {
                super();
        }

        public TokenNotPresentException(String message) {
                super(message);
        }

        public TokenNotPresentException(String message, Throwable cause) {
                super(message, cause);
        }

        public TokenNotPresentException(Throwable cause) {
                super(cause);
        }
}
```

Package muskel.mdf

ComputeResult.java

```
package muskel.mdf;

import java.io.Serializable;
/**
 * ComputeResult ospita il risultato
 * prodotto dal metodo compute di Mdfi,
 * cioe' un vettore di OutputToken
 * */
public class ComputeResult implements Serializable {

	private static final long serialVersionUID = 3465514833572047651L;

	private OutputToken[] outTok;
	/**
	 * Crea un nuovo ComputeResult contenente il
	 * vettore di {@link OutputToken} specificato
	 * */
	public ComputeResult(OutputToken[] outTok) {
		this.outTok = outTok;
	}
	/**
	 * Restituisce il vettore di {@link OutputToken}
	 * contenuto in this.
	 * */
	public OutputToken[] getOutTokens() {
		return outTok;
	}
}
```

Destination.java

```
/* This is the muskel: a skeleton/RMI based, parallel programming library
Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

package muskel.mdf;

import java.io.Serializable;

/**
 * used to store a destination in compact form.
 * A triple InputPositionId, InstructionId, GraphId
 *
 * @author marcod
 * @author albanese
 */
public class Destination implements Serializable, Cloneable {

	private static final long serialVersionUID = -498727500300417353L;

	private InputPositionId p = new InputPositionId(new IntegerIdentifier(0)); // position in the
input token vector
	private InstructionId i = Mdfi.NoInstrId;
	private GraphId g = Mdfi.NoGraphId;

	/**
	 * The Destination constructor
	 *
	 * @param pp position of the dest token in the input vector
```

```
 *             of the target instruction (positions start at 0)
 * @param ii identifier of the target instruction.
 *                    In case the token is destined to the output stream,
 *                    this must be a Mdfi.NoInstrId constant.
 * @param gg graph identifier of the target instruction.
 *                    While constructing the graph, this should be set to Mdfi.NoGraphId
 */
public Destination(InputPositionId pp, InstructionId ii, GraphId gg) {
        p = pp;
        i = ii;
        g = gg;
}
/**
 * The Destination constructor, to be used when constructing MDF graphs.
 * In this case there is no need to specify the graph id,
 * as it will be "relocated" later
 *
 * @param pp position of the Destination token in the input vector
 *                    of the target instruction (positions start at 0 ???)
 * @param ii identifier of the target instruction.
 *             In case the token is destined to the output stream,
 *             this is to be a Mdfi.NoInstrId constant.
 */
public Destination(InputPositionId pp, InstructionId ii) {
        this(pp, ii, Mdfi.NoGraphId);
}

/**
 * Returns the position in the destination instruction
 * input token vector
 * @return the position in the destination instruction
 *                    input token vector
 */
public InputPositionId getInputPositionId() {
        return p;
}
/**
 * Accessory method
 * @return the destination instruction id
 */
public InstructionId getInstructionId() {
        return i;
}

 /**
  * accessory method
  * @return the graph id of the target instruction
  */
public GraphId getGraphId() {
        return g;
}

public void setInputPositionId(InputPositionId p) {
        this.p = p;
}
public void setInstructionId(InstructionId i) {
        this.i = i;
}
public void setGraphId(GraphId g) {
        this.g = g;
}
        /**
 * overwrites Object toString method
 * @return the string representation of the Destination object
 */
public String toString() {
        return ("Destination : <"+p+","+i+","+g+">");
}
        /**
 * Deep copy
 * */
@Override
public Object clone() {

        InputPositionId cloneP = (InputPositionId) this.p.clone();
```

```
                InstructionId cloneI = (InstructionId) this.i.clone();
                GraphId cloneG = (GraphId) this.g.clone();

                return new Destination(cloneP, cloneI, cloneG);
        }
}
```

GenericId.java

```
package muskel.mdf;

import java.io.Serializable;

/**
 * Generic Mdfi Id superclass
 * */
public class GenericId implements Cloneable, Serializable {

        private static final long serialVersionUID = -4895486775094105801L;

        protected Identifier id;

        public GenericId(Identifier id) {
                this.id = id;
        }
        /** Getter */
        public Identifier getId() {
                return id;
        }
        @Override
        public String toString()
        {
                return id.toString();
        }
        /**
         * shallow copy perche' Identifier e' immutabile
         * */
        @Override
        public Object clone() {

                return new GenericId(this.id);
        }
        @Override
        public int hashCode() {
                final int prime = 31;
                int result = 1;
                result = prime * result +
                                        ((id == null) ? 0 : id.hashCode());
                return result;
        }
        @Override
        public boolean equals(Object obj) {
                if (this == obj)
                        return true;
                if (obj == null)
                        return false;
                if (!(obj instanceof GenericId))
                        return false;
                final GenericId other = (GenericId) obj;
                if (id == null) {
                        if (other.id != null)
                                return false;
                } else if (!id.equals(other.id))
                        return false;
                return true;
        }

}
```

GraphId.java

```
package muskel.mdf;

/** Macro Data Flow Graph Identifier*/
```

```
public class GraphId extends GenericId{

	private static final long serialVersionUID = 8673538422613856748L;

	/** Costruisce un nuovo GraphId specificando l' Identifier*/
	public GraphId(Identifier id) {
		super(id);
	}

	@Override
	public Object clone() {

		return new GraphId(this.id);
	}
}
```

Identifier.java

```
package muskel.mdf;

import java.io.Serializable;

/**
 * Interfaccia per gli identificatori.
 * E' richiesta l'immutabilitÃ  per permettere
 * la clone implementata come shallow copy;
 * */
public interface Identifier extends Comparable<Identifier>, Serializable {

	public boolean equals(Object id);

	public int hashCode();
}
```

IdGenerator.java

```
package muskel.mdf;

import java.io.Serializable;

/**
 * Identifier Generator Interface
 *
 * */
public interface IdGenerator extends Serializable {
	/**
	 * Generate a new identifier. Must be unique.
	 * @return  Generate a new identifier
	 * */
	public Identifier generateId();

}
```

InputPositionId.java

```
package muskel.mdf;
/**
 * Id di una posizione all'interno del vettore
 * degli input di una Mdfi
 * L'Id deve essere un IntegerIdentifier.
 *
 * @author albanese
 *
 * */
public class InputPositionId extends GenericId {

	private static final long serialVersionUID = -419660152059173135L;

	/** Costruisce un nuovo InputPositionId specificando l'Identifier*/
	public InputPositionId(IntegerIdentifier id) {
		super(id);
```

```
	}

	@Override
	public Object clone() {

		return new InputPositionId((IntegerIdentifier) this.id);
	}
}
```

InputToken.java

```
package muskel.mdf;
/**
 * Un input Token contentente un task e un bit di presenza.
 * La differenza rispetto ad un Token Ã¨ puramente logica.
 *
 * @author albanese
 * */
public class InputToken extends Token {

	private static final long serialVersionUID = -1867553850460846344L;

	/**
	 * Creates a input token with a given value
	 * @param task the value to be stored in the input token
	 */
	public InputToken(Task task) {
		super(task);
	}

	/**
	 * This is to create an empty input token, just a placeholder
	 */
	public InputToken()	{
		super();
	}

	/**
	 * This is to store a task in an empty input token
	 * or in an already setup token
	 * @param task the new task for the input token
	 */
	public void updateInputToken(Task task) {
		super.updateToken(task);
	}

	public String toString() {
		return super.toString();
	}

	public Object clone() {

		Task clonedTask =
			(task == null) ? null : (Task) task.clone();

		InputToken clone = new InputToken();
		clone.task = clonedTask;
		clone.presenceBit = this.presenceBit;

		return clone;
	}
}
```

InstrucitonId.java

```
package muskel.mdf;

/** Macro Data Flow Instruction Identifier*/

public class InstructionId extends GenericId {

	private static final long serialVersionUID = 5464752645110361180L;
```

```
	/** Costruisce un nuovo InstructionId specificando l'Identifier*/
	public InstructionId(Identifier id) {
		super(id);
	}

	@Override
	public Object clone() {

		return new InstructionId(this.id);
	}
}
```

IntegerIdentifier.java

```
package muskel.mdf;

public class IntegerIdentifier implements Identifier {

	private static final long serialVersionUID = 1370223080233032298L;

	private Integer intId;

	public IntegerIdentifier(Integer id) {
		this.intId = id;
	}

	public Integer getId() {
		return intId;
	}

	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result + ((intId == null) ? 0 : intId.hashCode());
		return result;
	}

	@Override
	public boolean equals(Object id) {
		if (this == id)
			return true;
		if (id == null)
			return false;
		if (!(id instanceof IntegerIdentifier))
			return false;
		final IntegerIdentifier other = (IntegerIdentifier) id;
		if (intId == null) {
			if (other.intId != null)
				return false;
		} else if (!intId.equals(other.intId))
			return false;
		return true;
	}

	public int compareTo(Identifier arg0) {

		IntegerIdentifier arg = (IntegerIdentifier) arg0;

		return intId.compareTo(arg.getId());
	}

	public String toString()
	{
		return intId.toString();
	}
}
```

IntegerIdGenerator.java

```
package muskel.mdf;
```

```java
/**
 * A simple Integer Identifier Generator Implementation.
 * Generates a new, progressive, Integer id.
 * */
public class IntegerIdGenerator implements IdGenerator {

	private static final long serialVersionUID = 453779581267041580L;

	private static int nextId = 0;
	private static final int NO_ID = -1;

	private static final IntegerIdentifier NOID = new IntegerIdentifier(NO_ID);

	/**
	 * Generates a new Integer id.
	 *
	 * @return A new Integer id
	 * */
	public IntegerIdentifier generateId() {

		return new IntegerIdentifier(nextId++);
	}

	/**
	 * Returns the NoId special Integer Identifier.
	 *
	 * @return The NoId special Integer Identifier.
	 * */
	public static IntegerIdentifier getNoId() {
		return NOID;
	}
}
```

MdfGraph.java

```java
/* This is the muskel: a skeleton/RMI based, parallel programming library
Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

package muskel.mdf;

import java.util.Collection;
import java.util.HashMap;
import java.util.Map;
import muskel.Manager;

/**
 * This is the class used to store a whole graph.
 * Accessory methods are provided to interact with and use the graph.
 *
 * The Input Mdfi must have an unique input token.
 *
 * The Output Mdfi must have an unique Destination and
 * the Destination InstructionId must be Mdfi.NoInstrId
 *
 * Usage example:
 *
 *             MdfGraph graph = new MdfGraph();
 *
 *             graph.setInputInstruction(i1);
 *             graph.addInstruction(i1);
```

```
 *             graph.addInstruction(i2);
 *             graph.addInstruction(i3);
 *             graph.addInstruction(i4);
 *             graph.setOutputInstruction(i4);
 *
 * @author marcod
 * @author albanese
 */
public class MdfGraph {

	public final boolean debug = Manager.debug;

	Map<InstructionId, Mdfi> graph = null;
	// used to store the instructions

	/**
	 * L'id della prima mdfi del grafo
	 * deve avere un unico token di input
	 * */
	InstructionId inputMdfiId = null;

	/**
	 * L'id dell'ultima mdfi del grafo l'unica
	 * che ha NoIstructionId come id di destinazione
	 * deve avere, inoltre, un unico token di uscita
	 * */
	InstructionId outputMdfiId = null;

	GraphId gid = Mdfi.NoGraphId;  // this is the graph id, actually

	/**
	 * numero di mdfi con SECompute
	 * */
	private int numSECompute = 0;

	/**
	 * Standard constructor
	 */
	public MdfGraph() {
		this(Mdfi.NoGraphId);
	}

	/**
	 * Constructor with a given graph id (not used at the moment)
	 * @param ggid
	 */
	public MdfGraph(GraphId ggid) {
		graph = new HashMap<InstructionId, Mdfi>();
		gid = ggid;
	}


	/**
	 * Getter method for the graph
	 */
	public Map<InstructionId, Mdfi> getGraph() {
		return graph;
	}

	/**
	 * Returns a collection containing the graph Mdfi
	 */
	public Collection<Mdfi> getGraphMdfi() {
		return graph.values();
	}

	/**
	 * used to set the graph id when the graph is instantiated
	 * @param ggid
	 */
	public void setId(GraphId ggid) {

		gid = ggid;

		/*ottengo una vista dei valori del Map come
```

```
		 * Collection in modo da poter iterare
		*/
		Collection<Mdfi> iterableGraph = graph.values();

		//aggiorno nelle mdfi il graphId
		for (Mdfi mdfi : iterableGraph) {
			mdfi.setGraphId(ggid);
		}

	}

	/**
	 * this is used to get the graph id.
	 * It will be used to instantiate new instructions ...
	 * not derived from skeleton compilation
	 * @return the id of the graph
	 */
	public GraphId getGraphId() {
		return gid;
	}

	/**
	 * Return the graph's SE Mdfi number
	 *
	 * */
	public int getNumSECompute() {
		return numSECompute;
	}

	/**
	 * Adds an instruction to the graph
	 * @param instr the instruction to be added
	 */
	public void addInstruction(Mdfi instr) {

		//aggiungo usando come chiave l'instructionId di instr
		graph.put(instr.getInstrId(), instr);

		//se instr e' ha side effects aggiorno il relativo contatore
		if(instr.isSeMDFi())
			numSECompute++;
	}

	/**
	 * Sets the first graph instruction.
	 * A Mdfi is a valid Input Mdfi if:
	 * - it has an unique InputToken
	 *
	 * @param intputInstr The instruction to be set
	 * @throws IllegalArgumentException If inputInstr
	 *                    is null or if it doesn't have an unique input Token
	 */
	public void setInputInstruction(Mdfi intputInstr) {

		if(!validateInputMdfi(intputInstr))
			throw new IllegalArgumentException();

		//aggiungo usando come chiave l'instructionId di firstInstr
		inputMdfiId = intputInstr.getInstrId();

	}

	/**
	 * Sets the last instruction to the graph.
	 * A Mdfi is a valid Output Mdfi if:
	 * - it has an unique Destination
	 * - the Destination Instruction Id must be Mdfi.NoInstrId
	 *
	 * @param outputInstr the instruction to set
	 * @throws IllegalArgumentException If outputInstr is null or if it isn't valid.
	 */
	public void setOutputInstruction(Mdfi outputInstr) {

		if(!validateOutputMdfi(outputInstr))
			throw new IllegalArgumentException();
```

```
        //aggiungo usando come chiave l'instructionId di outputInstr
        outputMdfiId = outputInstr.getInstrId();

}

/**
 * Controlla se una mdfi puo' essere usata come mdfi di input di un grafo.
 * Essa deve avere un unico Token di Input
 *
 * */
private boolean validateInputMdfi(Mdfi intputInstr) {

        if (intputInstr == null)
                return false;

        //La Mdfi di input deve avere un unico input Token.
        if (intputInstr.getInCount() != 1)
                return false;

        return true;
}

/**
 * Controlla se una mdfi puo' essere usata come mdfi di output di un grafo
 *
 * A Mdfi is a valid Output Mdfi if:
 * - it has an unique Destination
 * - the Destination Instruction Id is Mdfi.NoInstrId
 *
 * */
private boolean validateOutputMdfi(Mdfi outputInstr) {

        if (outputInstr == null)
                return false;

        //La Mdfi di output deve avere un unico output Token.
        if (outputInstr.getOutCount() != 1)
                return false;

        //prendo il primo (e unico, come appena controllato) outputToken della mdfi
        OutputToken outToken = outputInstr.getOutTokenVector()[0];

        //prelevo la testinazione dal token
        Destination dest = outToken.getDestination();

        //l'InstructionId contenuto in dest DEVE essere NoInstrId
        if(! (dest.getInstructionId().equals(Mdfi.NoInstrId)) )
                return false;

        //tutto ok
        return true;
}

/**
 * This is used to get the id of the input instruction,
 * the one where the input token has to be stored
 * @return the id of the first instruction
 */
public InstructionId getInputInstrId() {
        return inputMdfiId;
}

/**
 * This is used to get the id of the output instruction
 * @return the id of the last instruction
 */
public InstructionId getOutputInstrId() {
        return outputMdfiId;
}

/**
 * Stores he given destination, if not null, in the last instruction
 * (i.e. the only one whose unique output token dest is NoInstrId).
 * @param dest the Destination to be given to the last instruction
```

```
 * @return the InstructionId of the last instruction
 * @throws IllegalArgumentException if dest is null
 */
public InstructionId setOutputInstrId(Destination dest) {

        if(dest == null)
                throw new IllegalArgumentException();

        if(debug) System.out.println("setOutput Dest is "+dest.toString());
        if(debug) System.out.println("IN GRAPH "+graph.toString());

        Mdfi outputMdfi = graph.get(outputMdfiId);

        if(debug) System.out.println("CHANGING "+outputMdfi);

        int firstPosition = 0;//prima posizione nell'array delle destinazioni

        /*aggiorno la prima e unica
         * essendo, essa, una mdfi di output) destinazione.
        */outputMdfi.setDestination(firstPosition, dest);

        if(debug) System.out.println("BECOMES "+outputMdfi);

        return(outputMdfiId);
}

/**
 * Used to set graph id to a given value.
 * It is used when a graph is instantiated with the new input stream token
 * @param ggid the graph id
 * @throws IllegalArgumentException If ggid is null
 *  */
public void setGid(GraphId ggid) {

        if(ggid == null)
                throw new IllegalArgumentException();

        /*ottengo una vista dei valori del Map
         * come Collection in modo da poter iterare
        */Collection<Mdfi> iterableGraph = graph.values();

        for (Mdfi mdfi : iterableGraph) {
                mdfi.setGraphId(ggid);
                mdfi.setDestGraphIds(ggid);
        }

        gid = ggid;

}

/**
 * This is the method computing an MDF Graph on a given input token. <br>
 * @param task the value of the input task
 * @return the value of the unique computed token
 */

public Task[] compute(Task[] task) {
        // TODO : this is dummy !
        Task[] dummy = null;
        System.out.println("Unimplemented compute method in MdfGraph");
        return dummy;
}

/**
 * overrides Object toString
 * @return the pretty printed graph
 */
public String toString() {
        String out = "GRAPH is\n";

        /*ottengo una vista dei valori del Map
         *come Collection in modo da poter iterare
        */
        Collection<Mdfi> iterableGraph = graph.values();
```

```
        int i = 0;

        for (Mdfi mdfi : iterableGraph) {
                out = out + "instr:"+i+":"+mdfi+"\n";
                i++;
        }

        return out;
}

/**
 * Controlla se il grafo Ã¨ stato inizializzato.
 * CioÃ¨ se sono state definite le mdfi di in e/o di out
 *
 * */
private boolean isInitialized()
{
        return (inputMdfiId != null) && (outputMdfiId != null);
}


/**
 * overrides the Object clone.
 * Used to create a new instance of the graph
 * each time a new item is got from the input stream.
 * @return the (empty) graph
 * @throws IllegalStateException Se il grafo non e' stato
 *                               inizializzato correttamente
 *                                       (non sono definite le mdfi di in e/o di out)
 */
public MdfGraph mdfClone() {

        //controllo se sono state definite le mdfi di in e di out
        if(!isInitialized())
                throw new IllegalStateException("Graph not initialized.");

        //creo un grafo vuoto
        MdfGraph ng = new MdfGraph();

        ng.graph = new HashMap<InstructionId, Mdfi>();

        /*ottengo una vista dei valori del Map come Collection
         * in modo da poter iterare
        */
        Collection<Mdfi> iterableGraph = this.graph.values();

        //mdfi di input
        Mdfi toCloneInputInstr = this.graph.get(this.getInputInstrId());

        //mdfi di output
        Mdfi toCloneOutputInstr = this.graph.get(this.getOutputInstrId());

        //clono le mdfi e le aggiungo al nuovo grafo

        //clono l'istruzione di input
        Mdfi clonedInputInstr = toCloneInputInstr.mdfClone();
        //la aggiungo al grafo
        ng.addInstruction(clonedInputInstr);
        //la imposto come istruzione di input del grafo clonato
        ng.setInputInstruction(clonedInputInstr);

        //poi le altre tranne quella di output
        for (Mdfi mdfi : iterableGraph) {

                // non Ã¨ una mdfi ne' di in ne' di out
                if(toCloneInputInstr != mdfi && toCloneOutputInstr != mdfi) {
                        //clono
                        Mdfi clonedMdfi = mdfi.mdfClone();
                        //aggiungo
                        ng.addInstruction(clonedMdfi);
                }
        }

        //clono l'istruzione di output
        Mdfi clonedOutputInstr = toCloneOutputInstr.mdfClone();
```

```
                //la aggiungo al grafo
                ng.addInstruction(clonedOutputInstr);
                //la imposto come istruzione di output del grafo clonato
                ng.setOutputInstruction(clonedOutputInstr);

                //clono il graphId
                ng.gid = (GraphId) this.gid.clone();

                return ng;
        }


        public Object clone() {

                return mdfClone();
        }
}
```

Mdfi.java

```
/* This is the muskel: a skeleton/RMI based, parallel programming library
Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

package muskel.mdf;

import java.io.Serializable;
import skeleton.Compute;
import skeleton.SECompute;
import muskel.CodeStorage;
import muskel.Manager;
import muskel.NonFireableInstructionException;
import muskel.OpCodeNotPresentException;
import muskel.TokenNotPresentException;

/**
 * This is the class of MDF instructions: each MDFi has a
 * instrId opCode tokenInVector tokenOutVector readyArgNo inTokenCount outTokenCount
 * @author marcod
 * @author albanese
 *
 */
public class Mdfi implements Serializable {

        public final boolean debug = Manager.debug;

        private static final long serialVersionUID = 1L;

        private static final Identifier NO_ID = IntegerIdGenerator.getNoId();

        public final static InstructionId NoInstrId = new InstructionId(NO_ID);
        public final static GraphId NoGraphId = new GraphId(NO_ID);
        public final static OperationId NoOpCode = new OperationId(NO_ID);

        private InstructionId instrId = NoInstrId;
        private GraphId graphId = NoGraphId;
        private OperationId opCode = NoOpCode;

        InputToken [] inTokenVector = null;

        Destination [] dests = null;
```

```
        OutputToken [] outTokenVector = null;

        int readyArgNo = 0;
        int inCount = 0;
        int outCount = 0;

        boolean fireable = false;

        /**flag per identificare le mdfi con side effects*/
        boolean seMDFi = false;

    /** Creates a MDFi taking care of storing the code in the CodeStorage
     * This is used only in case of user defined ParCompute code.
     * Same parameters of the previous, complete constructor,
     * but the GID, which is left undefined.
     * Plus the manager, which is needed to access the global CodeStorage
     * @param manager the manager used to compute the instruction
     * @param iid the instruction id
     * @param op the code to be computed by the instruction.
     *                Should be a Compute (sub)class object
     * @param inc the input token number
     * @param outc the output token number
     * @param did the vector of the destinations for output tokens.
     *                The i-th output token is directed to the i-th Destination
     */
        public Mdfi(Manager manager, InstructionId iid, Compute op, int inc, int outc, Destination[]
did) {

                // first set up all parameters but the opcode checking the op type
                this(iid,NoGraphId, NoOpCode, inc, outc, did, (op instanceof SECompute));
                // then fix the opcode, after storing the Compute code in the CodeStore
                opCode = manager.storeOpCode(op);

                if(debug) System.out.println("first constructor: outToken dims to
"+outTokenVector.length);
        }

   /**
    *
    * Creates a MDFi
    * @param iid the instruction id
    * @param op The code id in the CodeStorage to be
    *                computed by the instruction.
    * @param inc the input token number
    * @param outc the output token number
    * @param did the vector of the destinations for output tokens.
    *                 The i-th output token is directed to the i-th Destination
    * @param seMDFi the flag representing a mdfi having a side action compute
    */
        public Mdfi(InstructionId iid, GraphId gid,
                        OperationId op, int inc, int outc, Destination[] did, boolean seMDFi) {

                this.instrId = iid;
                this.graphId = gid;
                this.opCode = op;
                inCount = inc;
                outCount = outc;
                this.dests = did;

                this.seMDFi = seMDFi;

                inTokenVector = new InputToken[inc];
                for(int i=0; i<inCount; i++)
                        inTokenVector[i] = new InputToken();
                        // initialize the Token input vector (all empty at the moment)

                outTokenVector = new OutputToken[outCount];
                for(int i=0; i<outCount; i++)
                        outTokenVector[i] = new OutputToken(dests[i]);

                readyArgNo = 0;
                fireable = false;

                if(debug)
```

```
                        System.out.println("first constructor: outToken dims to "
        +outTokenVector.length);
        }

        /**
         * Used to store an input token
         * @param position the position in the input token vector
         * @param value the value of the token
         */
        public void storeToken(InputPositionId position, Task value) {
                //ricavo l'int dall InputPositionId
                int pos = ((IntegerIdentifier) position.getId()).getId();

                inTokenVector[pos].updateToken(value);
                readyArgNo++;
                if(readyArgNo==inCount)
                        fireable = true;
        }

        /**
         * Computes the MDF instruction, taking the opcode from the CodeStorage
         * @param cs the CodeStorage to be used to retrieve instruction code
         * @return the vector of the result tokens (OutputTokens, actually)
         * @throws NonFireableInstructionException
         * @throws OpCodeNotPresentException
         */
        public ComputeResult compute(CodeStorage cs) throws NonFireableInstructionException,
OpCodeNotPresentException  {
                // log.debug("MDFI COMPILE: going to compute");
                if(fireable) {
                        // log.debug("MDFI COMPILE: going to compute fireable");
                        // first compute the values
                        Task[] tokenValues = new Task[inCount];
                        for(int i=0; i<inCount; i++) {
                                try {
                                        tokenValues[i] = inTokenVector[i].getTask();
                                }catch(TokenNotPresentException e) {
                                        throw new NonFireableInstructionException();
                                } catch(ArrayIndexOutOfBoundsException e) {
                                    e.printStackTrace();
                                }
                        }
                        // log.debug("MDFI COMPILE: computing fireable: got tokens");
                        if(debug) System.out.println("going to compute with "+tokenValues.length+"
input tokens ");

                        Compute comp = cs.getCompute(opCode);

                        Task[] result;
                        ComputeResult toReturn;

                        Task[] generatedTask = null;
                        if(seMDFi) {
                                /*
                                 * nel caso di un SEcompute e' necessario
                                 * rendere atomicala sequenza delle
                                 * seguenti 2 operazioni:
                                 * - compute(), dove viene impostata la
                                 *       variabile condivisa generatedTask,
                                 * - getGeneratedTask() dove viene prelevata
                                 *       la variabile stessa.
                                 *
                                 * Devo, percio', sincronizzarmi su comp
                                 * (contenuto nel CodeStorage)
                                 * che contiene la variabile condivisa.
                                 * Cosi' facendo sono sicuro di essere
                                 * l'unico ad operare su comp.
                                */
                                synchronized (comp){
                                        result = (Task[]) comp.compute(tokenValues);

                                        SECompute seComp = (SECompute) comp;

                                        Task[] gtTemp = seComp.getGeneratedTask();
```

```
                                if(gtTemp != null) {

                                        generatedTask = new Task[gtTemp.length];
                                        //clono i task generati
                                        for(int i = 0; i< gtTemp.length; i++)
                                                generatedTask[i] = (Task) gtTemp[i].clone();

                                        //reimposto il valore per il prossimo utilizzo
                                        seComp.resetGeneratedTask();
                                }
                        }

                }
                else {//caso Compute
                        result = (Task[]) comp.compute(tokenValues);
                }
                /*  ATTENZIONE  L'i-esimo valore viene passato all'i-esima destinazione.
                 *  Quindi una mdfi con n destinazioni DEVE avere
                 *  una Compute che calcoli con n output.
                 *  E' a carico di chi scrive la Mdfi controllare che il num
                 *  di destinazioni sia compatibile (cioÃ¨ #dest = #result)
                 *  con la Compute utilizzata.
                 *  */

                //i-esimo risultato all'i-esima destinazione
                for(int i=0; i<outCount; i++)
                        outTokenVector[i] = new OutputToken(result[i], dests[i]);


                //genero il risultato in base al tipo di mdfi
                if(seMDFi)
                        toReturn = new SEComputeResult(outTokenVector, generatedTask);

                else toReturn = new ComputeResult(outTokenVector);

                // eventually return the OutputToken vector
                return toReturn;

        } else
                throw new NonFireableInstructionException();
}

/**
 * test for fireable instruction
 * @return returns true if all the input
 * tokens are present and the instruction can be computed
 */
public boolean isFireable() {
        return fireable;
}

public InstructionId getInstrId() {
        return instrId;
}


public GraphId getGraphId() {
        return graphId;
}


public InputToken[] getInTokenVector() {
        return inTokenVector;
}

public OutputToken[] getOutTokenVector() {
        return outTokenVector;
}


/**
 *Il un numero di token di ingresso della Mdfi
 */
public int getInCount() {
```

```
		return inCount;
	}

	/**
	 *Il un numero di token di uscita della Mdfi
	 */
	public int getOutCount() {
		return outCount;
	}

	public boolean isSeMDFi() {

		return seMDFi;
	}

	/**
	 * Assigns a graph id to the instruction, possibly created with a Mdfi.NoGraphId value.
	 * @param gid the graph id to be assigned
	 * @throws IllegalArgumentException If gid is null
	 */
	public void setGraphId(GraphId gid) {

		if (gid == null)
			throw new IllegalArgumentException();

		graphId = gid;
	}

	/**
	 * Sets the graph id in the destination tokens
	 * @param gid the graph id to be used
	 */
	public void setDestGraphIds(GraphId gid) {
		for(int i=0; i<outCount; i++) {
			 //cosi' modifico anche outputToken perchÃ¨ punta all'i-esimo dest
			dests[i].setGraphId(gid);

			//dests[i] = new Destination(dests[i].getInputPositionId(),
dests[i].getInstructionId(), gid);
			//outTokenVector[i].setDest(dests[i]);
		}
	}

	/**
	 * Modifica la destinazione del ppos-esimo token di output
	 * @param ppos the index of the destination to be changed
	 * @param dest the new destination
	 */
	public void setDestination(int ppos, Destination dest) {

		this.dests[ppos] = dest;
		this.outTokenVector[ppos] = new OutputToken(dest);

		if(debug) System.out.println("Changing dest pos "+ppos+" with
"+dest.toString()+"\ngives: "+this.toString());

	}

	/**
	 * Overrides the Object toString
	 * @return the pretty printed instruction
	 */
	public String toString() {
		String out = "MDFI <"+instrId+","+graphId+">:";
		out = out +
"op="+opCode+":in="+inCount+":out="+outCount+":fireable="+fireable+":tokens:";
		for(int i=0; i<inCount; i++)
			out = out + inTokenVector[i].toString();
		out = out + ":destTokens:";
		for(int i=0; i<outCount; i++)
			out = out + outTokenVector[i].toString();
		return out;
	}

	/**
```

```
         * used by clone()
         * */
        private Mdfi() {}

        /**
         * This is used to clone an instruction.
         * Called by mdfClone in MdfGraph, when a new instance
         * of the graph is to be stored with the input
         * token from the input stream as the graph input token.
         * <br>
         * Does not clone the opcode, as this is data flow
         *
         * @return a copy of the current instruction
         */
        public Mdfi mdfClone() {

                Mdfi ni = new Mdfi();

                ni.instrId = (InstructionId) instrId.clone();
                ni.graphId = (GraphId) graphId.clone();
                ni.opCode = this.opCode;

                ni.inTokenVector = (inTokenVector == null) ? null : cloneInTokenArray();
                ni.dests = (dests == null) ? null : cloneDestinationArray();
                ni.outTokenVector = (outTokenVector == null) ? null : cloneOutputTokenArray();

                ni.readyArgNo = readyArgNo;
                ni.inCount = inCount;
                ni.outCount = outCount;
                ni.fireable = fireable;
                ni.seMDFi = seMDFi;

                return ni;
        }

        /**
         * Clona con deep copy il vettore dei token d'ingresso
         * */
        private InputToken[] cloneInTokenArray()
        {
                InputToken[] clone = new InputToken[this.inTokenVector.length];
                for(int i=0; i<this.inTokenVector.length;i++)
                        clone[i] = (InputToken) this.inTokenVector[i].clone();

                return clone;
        }

        /**
         * Clona con deep copy il vettore dei token d'uscita
         * */
        private OutputToken[] cloneOutputTokenArray()
        {
                OutputToken[] clone = new OutputToken[this.outTokenVector.length];
                for(int i=0; i<this.outTokenVector.length;i++)
                        clone[i] = (OutputToken) this.outTokenVector[i].clone();

                return clone;
        }

        /**
         * Clona con deep copy il vettore delle destinazioni
         * */
        private Destination[] cloneDestinationArray()
        {
                Destination[] clone = new Destination[this.dests.length];
                for(int i=0; i<this.dests.length;i++)
                        clone[i] = (Destination) this.dests[i].clone();

                return clone;
        }

}
```

OperationId.java

```
package muskel.mdf;
/**
 * Id utilizzato come OpCode in una Mdfi.
 * L'id e' solitamente utilizzato per indicizzare un Compute contenuto in CodeStorage
 * */
public class OperationId extends GenericId {

	private static final long serialVersionUID = -7489946694214420383L;

	public OperationId(Identifier id) {
		super(id);
	}

	@Override
	public Object clone() {

		return new OperationId(this.id);
	}
}
```

OutoutToken.java

```
package muskel.mdf;

/**
 * This is the class hosting a destination token,
 * as output from a MDF instruction.
 * It's a token with destination address embedded in.
 *
 * @author marcod
 *
 */
public class OutputToken extends Token implements Cloneable {

	private static final long serialVersionUID = 4563691392373961077L;

	private Destination dest;

	/**
	 * Crea un OutputToken con task e destination specificati
	 * Il bit di presenza del token e' a true;
	 *
	 * */
	public OutputToken(Task t, Destination d) {
		super(t);
		dest = d;
	}

	/**
	 * Crea un OutputToken con destination specificato
	 * Il bit di presenza del token e' a false e il task e' null
	 *
	 * */
	public OutputToken(Destination d) {

		super();
		dest = d;
	}

	public void setDest(Destination dest) {
		this.dest = dest;
	}

	public String toString() {

		return super.toString() + " " + dest;
	}

	/**
	 * deep copy
```

```
         * */
        public Object clone() {

                Destination clonedDest =
                                (this.dest == null) ? null : (Destination) this.dest.clone();

                OutputToken clone = new OutputToken(clonedDest);

                clone.task = (this.task == null) ? null : (Task) this.task.clone();

                clone.presenceBit = this.presenceBit;

                return clone;
        }

        public Destination getDestination() {
                return dest;

        }
}
```

SEComputeResult.java

```
package muskel.mdf;

/**
 * SEComputeResult ospita sia il risultato prodotto dal metodo compute di Mdfi,
 * cioe' un vettore di {@link OutputToken},
 * sia un vettore di nuovi {@link Task} prodotti dalla Mdfi stessa.
 * */
public class SEComputeResult extends ComputeResult {

        private static final long serialVersionUID = -3959544943968671714L;

        private Task[] generatedTasks;

        /**
         * Crea un nuovo SEComputeResult contenente il vettore di
         * {@link OutputToken} e quello di {@link Task} specificati
         * */
        public SEComputeResult(OutputToken[] outTok, Task[] generatedTasks)
        {
                super(outTok);
                this.generatedTasks = generatedTasks;
        }

        /**
         * Restituisce il vettore di {@link Task} contenuto in this.
         * Esso puo' essere null.
         * */
        public Task[] getGeneratedTasks() {
                return generatedTasks;
        }

}
```

Task.java

```
import java.io.Serializable;

/**
 * Un Task utilizzabile come valore di un Token
 * (sia di input che di output)
 * L'Object contenuto DEVE essere immutabile,
 * per permettere la clone() (di tipo shallow copy).
 *
 * @author albanese
 * */
public class Task implements Cloneable, Serializable {
```

```
	private static final long serialVersionUID = -3987064469265405519L;

	private Object value;

	public Task(Object value) {
		this.value = value;
	}

	public Object getValue() {
		return value;
	}

	/**
	 * Clona l'oggetto; esso deve essere immutabile
	 * in quanto viene eseguita una shallow copy
	 * */
	public Object clone() {

		return new Task(this.value);
	}

	public String toString() {

		return value.toString();
	}
}
```

Token.java

```
package muskel.mdf;

import java.io.Serializable;
import muskel.TokenNotPresentException;

/**
 * Un Token contentente un task e un bit di presenza.
 *
 * @author albanese
 * */
public class Token implements Serializable, Cloneable {

	private static final long serialVersionUID = 1L;

	protected Task task = null;
	protected boolean presenceBit = false;

	/**
	 * Creates a token with a given value
	 * @param task The value to be stored in the token
	 */
	public Token(Task task) {
		this.task = task;
		presenceBit = true;
	}

	/**
	 * This is to create an empty token, just a placeholder
	 */
	public Token() {
		task = null;
		presenceBit = false;
	}

	/**
	 * This is to store a task in an empty token
	 * or in an already setup token
	 * @param task The new task for the token
	 */
	public void updateToken(Task task) {
		this.task = task;
		presenceBit = true;
	}

	/**
```

```
         * @return Il task contenuto nel Token
         * @throws TokenNotPresentException Se il task
         *                     non e' presente (il bit di presenza e' false).
         *
         * */
        public Task getTask() throws TokenNotPresentException {
                if(presenceBit)
                        return task;
                else
                        throw new TokenNotPresentException();
        }

        public String toString() {
                if(task == null)
                        return("<null,"+presenceBit+">");
                else
                        return("<"+task.toString()+","+presenceBit+">");
        }

        /**
         * deep copy
         * */
        public Object clone() {

                Task clonedTask =
                        (task == null) ? null : (Task) task.clone();

                Token clone = new Token();
                clone.task = clonedTask;
                clone.presenceBit = this.presenceBit;

                return clone;
        }
}
```

Package muskel.stream

BlockingInputManager.java

```
package muskel.stream;
/**
 * Interfaccia degli gli InputManager che forniscono metodi
 * next e hasNext bloccanti e permettono agli
 * SECompute di aggiungere task.
 * */
public interface BlockingInputManager extends InputManager {

        /**
         * Aggiunge task allo stream
         * */
        public void addTask(Object task);


        /**
         * Segnala allo stream la presenza di numSECompute potenziali nuovi produttori
         * */
        public void addProducers(int numSECompute);

        /**
         * Segnala allo stream la presenza di un potenziale nuovo produttore
         * */
        public void addProducer();

        /**
         * Segnala allo stream che un produttore ha smesso di produrre
         * */
        public void removeProducer();
}
```

ConsoleOutuputManager.java

```
/* This is the muskel: a skeleton/RMI based, parallel programming library
Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy
```

```
This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

package muskel.stream;

/**
 * This is used to deliver output tokens to the user Console
 * toString() is used to print the items.
 *
 * @author marcod
 *
 */
public class ConsoleOutputManager implements muskel.stream.OutputManager {

        /**
         * this is the method actually delivering
         * the output token onto the user console,
         * using a println and converting the object to a string
         * with a call to toString() method.
         * @param result the result token
         *
         */
    public void deliver(Object result) {

        System.out.println("Delivering task: " + result.toString());
    }
}
```

DynamicInputManager.java

```
package muskel.stream;

import java.util.concurrent.LinkedBlockingQueue;


/**
 * Wrapper per un NonBlockingInputManager che sono "statici",
 * cioe' che non permettono l'aggiunta di elementi da produrre a runtime.
 *
 * DynamicInputManager consuma, durante la sua creazione,
 * NonBlockingInputManager (che dovra' contenere almeno un elemento)
 * passato per parametro; inoltre permette a dei produttori
 * in inserire nello stream nuovi elementi.
 *
 * I produttori devono segnalare allo stream sia
 * la volonta' di iniziare a produrre, che quella di smettere.
 *
 * */
public class DynamicInputManager implements BlockingInputManager {

        private LinkedBlockingQueue<Object> queue;

        /*numero di produttori dichiaratisi interessati a produrre*/
        private int nProducers;

        private boolean closed;



        /**
         * Crea un nuovo stream usando riempiendolo con gli elementi
```

```
         * di inStream, esso deve conterne almeno uno.
         * @param inStream uno stream contentente almeno un elemento
         * */
        public DynamicInputManager(NonBlockingInputManager inStream)       {

                queue = new LinkedBlockingQueue<Object>();

                closed = false;
                //uso -1 per evitare l'ambiguita' dello 0: stream chiuso o mai usato?
                //alla prima addProducers nProducers conterra' il numeto effettivo di produttori
                nProducers = -1;

                //inserisco nella coda gli elementi dello stream
                Object task;
                while ((task = inStream.next()) != null) {
                        queue.add(task);
                        System.out.println("Accodato task ottenuto dallo stream iniziale: "+ task);
                }
        }

        /**
         * Restituisce true se ci sono elementi pronti da consumare,
         * si blocca se non ci sono elementi pronti ma sono presenti
         * potenziali produttori. Qualora i produttori non aggiungano
         * nessun elemento durante l'attesa,il metodo restituisce false
         * */
        @Override
        public synchronized boolean hasNext() {

                if(!closed && nProducers > 0) {
                        /*
                         * lo stream non e' stato ancora chiuso, cioe'
                         * non ho mai avuto zero consumatori
                         *
                         * Se lo stream iniziale contiene almeno un elemento,
                         * e this viene usato da compute produttori
                         * (SECompute), esso avra' 0 produttori solo al termine dell'esecuzione
                         * */

                        if (!queue.isEmpty()) {
                                System.out.println("La coda NON e' vuota ci sono task da consumare");
                                return true; //la coda NON e' vuota ci sono task da consumare
                        }
                        else {
                                //la coda e' vuota, ma potrebbe riempirsi, aspetta
                                while(queue.isEmpty()) {

                                        if(closed) {
                                                System.out.println("I produttori che attendevamo non
hanno prodotto, lo stream e' chiuso");
                                                /*Durante la wait i produttori
                                                 * che attendevamo non hanno prodotto,
                                                 * lo stream e' chiuso
                                                */return false;
                                        }

                                        try {
                                                System.out.println("La coda e' vuota, ma potrebbe
riempirsi, aspetta...");
                                                wait();

                                                System.out.println("...Svegliato");
                                        } catch (InterruptedException e) {e.printStackTrace();}

                                }

                                System.out.println("La coda e' stata riempita. Posso estrarre.");
                                return true;//la coda e' stata riempita ci sono task da consumare
                        }
                }
                else {//non ci sono produttori (stream chiuso) ma la coda potrebbe non essere vuota

                        System.out.println("Non ci sono produttori, potrebbero esserci task
rimanenti");
```

```
                        if (queue.isEmpty()) {
                                System.out.println("Non ci sono altri task, lo stream e' esaurito");
                                return false; //la coda e' vuota, lo stream e' esaurito
                        }
                        else {
                                System.out.println("Ci sono task da consumare");
                                return true; //la coda NON e' vuota, ci sono task da consumare
                        }
                }
        }


        @Override
        public Object next() {

                //se la coda e' vuota aspetta
                Object toReturn = null;
                try {
                        toReturn = queue.take();
                        System.out.println("Task ottenuto: " + toReturn);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }

                return toReturn;
        }

        public synchronized void addTask(Object task) {

                try {
                        //se c'era qualcuno in attesa di un task lo sveglio
                        queue.put(task);
                        System.out.println("Accodato task, totale: "+ queue.size());
                } catch (InterruptedException e) {}//eccezione se non c'e' spazio della coda,
improbabile
                notifyAll();//sveglio i consumatori che si sono messi in attesa durante la hasNext()
        }

        public synchronized void addProducers(int n) {

                if(!closed) {

                        if(nProducers == -1) {//e' il primo produttore che aggiungo
                                nProducers = n;
                                System.out.println("Aggiunto primo produttore, totale: "+ nProducers);
                        }
                        else {
                                nProducers += n;

                                if(n==1)
                                        System.out.println("Aggiunto un produttore, totale: "+
nProducers);
                                else System.out.println("Aggiunti " + n +" produttori, totale: "+
nProducers);
                        }
                }
                else throw new IllegalStateException("Stream Closed");
        }

        public synchronized void addProducer() {

                if(!closed) {

                        if(nProducers == -1) {//e' il primo produttore che aggiungo
                                nProducers = 1;
                                System.out.println("Aggiunto primo produttore, totale: "+ nProducers);
                        }
                        else {
                                nProducers++;
                                System.out.println("Aggiunto produttore, totale: "+ nProducers);

                        }
                }
                else throw new IllegalStateException("Stream Closed");
        }
```

```
        public synchronized void removeProducer() {

                if(!closed) {

                        //decremento il num dei produttori se questo e' maggiore di 0
                        if (nProducers > 0) {
                                nProducers--;
                                System.out.println("Rimosso produttore, totale: "+ nProducers);

                                if (nProducers == 0) {
                                        /*il numero di produttori e' arrivato a zero
                                         * quindi ho finito e dichiaro lo stream chiuso
                                        */
                                        closed = true;
                                        notifyAll();
                                        /*sveglio i consumatori in attesa di nuovi
                                         * probabili task durante la hasNext()
                                         * ma siccome sono tutti terminati non
                                         * arriveranno nuovi task
                                         */
                                        System.out.println("Stream chiuso");
                                }
                        }
                        else //nProducers <= 0
                                throw new IllegalStateException("Non ci sono produttori da
eliminare");

                }
                else throw new IllegalStateException("Stream Closed");
        }
}
```

InputManager.java

```
* This is the muskel: a skeleton/RMI based, parallel programming library
Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

package muskel.stream;

/**
 * this is the interface that must be implemented
 * to supply the class providing the program input stream
 *
 * @author marcod
 *
 */
public interface InputManager {

    /**
     * this is the metod used by the manager to get
     * the next task to compute, if any. In case there is no more tasks,
     * a null is returned as the EndOfStream mark
     *  @return null if there is no more tasks, the task to be computed otherwise
     */

    public Object next();
```

```
    /** this is the method used to check whether there are other tasks
     * available onto the input stream
     *  @return true is there is at least one more task, false otherwise
     */
    public boolean hasNext();

}
```

IntegerStreamInputManager.java

```
/* This is the muskel: a skeleton/RMI based, parallel programming library
 Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

 This program is free software; you can redistribute it and/or
 modify it under the terms of the GNU General Public License
 as published by the Free Software Foundation; either version 2
 of the License, or (at your option) any later version.

 This program is distributed in the hope that it will be useful,
 but WITHOUT ANY WARRANTY; without even the implied warranty of
 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 GNU General Public License for more details.

 You should have received a copy of the GNU General Public License
 along with this program; if not, write to the Free Software
 Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

package muskel.stream;

/**
 *
 * Sample input manager. Delivers a number of Integer items onto the input
 * stream, when the next() method is invoked from within the manager
 *
 * @author marcod
 *
 */
public class IntegerStreamInputManager implements NonBlockingInputManager {

        private int howMuch;

        /**
         * creates a new IntegerStreamInputManager
         *
         * @param ithe number of items to be delivered on the input stream upon
         *            Manager request.
         *
         */
        public IntegerStreamInputManager(int i) {
                howMuch = i;
        }

        /**
         * delivers next item
         *
         * @return the new stream item (an Integer)
         */
        public Object next() {
                if (howMuch > 0) {
                        System.out.println("delivered input task : " + howMuch);
                        return new Integer(howMuch--);
                } else {
                        return null;
                }
        }

        /**
         * @return true if there are other items to be delivered onto the input
         *         stream, false otherwise
         */
        public boolean hasNext() {
                if (howMuch == 0) {
                        return false;
```

```
                } else {
                        return true;

                }
        }
}
```

IntegerVectorStreamInputManager.java

```
/* This is the muskel: a skeleton/RMI based, parallel programming library
 Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

 This program is free software; you can redistribute it and/or
 modify it under the terms of the GNU General Public License
 as published by the Free Software Foundation; either version 2
 of the License, or (at your option) any later version.

 This program is distributed in the hope that it will be useful,
 but WITHOUT ANY WARRANTY; without even the implied warranty of
 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 GNU General Public License for more details.

 You should have received a copy of the GNU General Public License
 along with this program; if not, write to the Free Software
 Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

package muskel.stream;

import java.util.Random;
import java.util.Vector;
/**
 *
 * Sample input manager. Delivers Integer Vector
 * (having the specified dimension) items onto the input
 * stream, when the next() method is invoked from within the manager
 *
 * @author albanese
 *
 */
public class IntegerVectorStreamInputManager implements NonBlockingInputManager {

        private int howMuch;
        private int dimension;
        private Random randomGenerator;

        //i numeri casuali generati sono in [0, maxInt-1]
        private static final int MAX_INT = 10001;
                /**
         * creates a new IntegerVectorStreamInputManager
         *
         * @param howMuch
         *            the number of items to be delivered on the input stream upon
         *            Manager request.
         * @param dimension The dimension of vector to generate
         */
        public IntegerVectorStreamInputManager(int dimension, int howMuch) {
                this.dimension = dimension;
                this.howMuch = howMuch;

                randomGenerator = new Random();
        }

        /**
         * delivers next item
         *
         * @return the new stream item (an Integer)
         */
        public Object next() {
                if (howMuch > 0) {
                        System.out.println("delivered input task : " + howMuch);

                        howMuch--;
                        return generateVector();
                } else {
                        return null;
```

```
                }
        }

        private Vector<Integer> generateVector() {

                Vector<Integer> v = new Vector<Integer>(dimension);
                for(int i = 0; i < dimension;i++)
                        v.add(randomGenerator.nextInt(MAX_INT));
                return v;
        }
        /**
         * @return true if there are other items
         *                      to be delivered onto the input stream,
         *                      false otherwise
         */
        public boolean hasNext() {
                if (howMuch == 0) {
                        return false;
                } else {
                        return true;

                }
        }
}
```

NonBlockingInputManager.java

```
package muskel.stream;
/**
 * Interfaccia marker che dovraÃ  essere implementata
 * dagli InputManager che forniscono metodi next
 * e hasNext non bloccanti
 * */
public interface NonBlockingInputManager extends InputManager {

}
```

OutputManager.java

```
/* This is the muskel: a skeleton/RMI based, parallel programming library
Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

package muskel.stream;

/**
 * this is the interface implemented to provide
 * the class post processing the results computed
 * by the skeleton program
 *
 * @author marcod
 */
public interface OutputManager {

    /**
     * this is the method used to deliver a result
     * by the program manager<br>
     * Users must supply an actual deliver method to properly handle the
     * current result
     */
    public void deliver(Object result);
```

```
}
```

Package muskel.util

JavaSourceFromString.java

```
package muskel.util;

import java.net.URI;
import javax.tools.SimpleJavaFileObject;

/**
 * A file object used to represent source coming from a string.
 */
public class JavaSourceFromString extends SimpleJavaFileObject {
    /**
     * The source code of this "file".
     */
    final String code;

    /**
     * Constructs a new JavaSourceFromString.
     * @param name the name of the compilation unit represented by
     *                          this file object
     * @param code the source code for the compilation unit
     *                          represented by this file object
     */
    public JavaSourceFromString(String name, String code) {
        super(URI.create("string:///" + name.replace('.','/')
                                        + Kind.SOURCE.extension), Kind.SOURCE);
        this.code = code;
    }

    @Override
    public CharSequence getCharContent(boolean ignoreEncodingErrors) {
        return code;
    }
}
```

Package skeleton

Compute.java

```
/* This is the muskel: a skeleton/RMI based, parallel programming library
Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

/*
 * Created on Mar 22, 2004
 *
 * To change the template for this generated file go to
 * Window - Preferences - Java - Code Generation - Code and Comments
 */
package skeleton;

import java.io.Serializable;
import muskel.mdf.Task;
```

```java
/**
  * This is the interface that has to be implemented by
  * the sequential portions of codeused in the program computation.
  *    Each sequential skeleton is an object that implements this interface.
  * We explicitly chose not to use 1.5 features, at the moment.
  * <br>
  * The typical usage is the following:
  * <ul>
  * <li> suppose you have to compute in parallel f(x1) ... f(xn)
  *            out of x1 ... xn<br>
  * <li> suppose f takes as input an InputTask task
  *     and produces and OutputResult result
  * <li> then you'll use a code such as:
  * <pre>
  * public class myF extends Compute {
  *    public Task[] compute(Task[] task) {
  *      OutputResult res = f((InputTask) task);
  *      Task[] toReturn = {new Task(res)};
  *      return toReturn;
  *    }
  * }
  * </pre>
  * every time you need to use the f function to compute
  * a pipeline stage, a farm worker, etc.
  * </ul>
  *
  * @author Danelutto Marco
 */
public abstract class Compute implements Cloneable, Serializable  {
        /**
         * the method computes a Task[] result out of a Task[] input data set
         * @return the result computed
         */
        public abstract Task[] compute(Task[] task);

        /**
         *  Just to be able to clone in case of usage of ObjectStreams
         *  Shallow copy (the Compute Object are immutable)
         * @return this object
         */
        public Object clone() {

                return this;
        }
}
```

Farm.java

```java
/* This is the muskel: a skeleton/RMI based, parallel programming library
Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

/*
 * Created on Mar 22, 2004
 *
 * To change the template for this generated file go to
 * Window - Preferences - Java - Code Generation - Code and Comments
 */
package skeleton;

import muskel.mdf.Task;
```

```java
/**
 * This is the "emabarassingly parallel" skeleton.
 * Each task supplied to the Farm is actually computed
 * independently of the other tasks by invoking the compute
 * method of the inner skeleton (alias the worker).
 * <br>
 *
 * @author Danelutto Marco
 *
 */
public class Farm extends Compute {

	private static final long serialVersionUID = 1L;
	Compute worker = null;

	/**
	 * creates a new Farm skeleton.
	 * @param worker is the skeleton used to implement the Farm
	 * workers, i.e. it is the skeleton modelling the computation
	 * that has to be independently performed on each one of
	 * the tasks submitted to the Farm.
	 */
	public Farm(Compute worker) {
		this.worker = worker;
	}

	/**
	 * This is only needed to compiler into MDF
	 *
	 * @return the worker code
	 */
	public Compute getWorker() {
		return worker;
	}
	/**
	 * computes a task sequentially.
	 * @param task the input data to be processed
	 * @return the result of the computation, as an Objects
	 */
	public Task[] compute(Task[] task) {
		return worker.compute(task);
	}
}
```

Farm.java

```java
/* This is the muskel: a skeleton/RMI based, parallel programming library
 Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

 This program is free software; you can redistribute it and/or
 modify it under the terms of the GNU General Public License
 as published by the Free Software Foundation; either version 2
 of the License, or (at your option) any later version.

 This program is distributed in the hope that it will be useful,
 but WITHOUT ANY WARRANTY; without even the implied warranty of
 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 GNU General Public License for more details.

 You should have received a copy of the GNU General Public License
 along with this program; if not, write to the Free Software
 Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */


package skeleton;

import muskel.mdf.MdfGraph;
import muskel.mdf.Task;

/**
 * This is the class used to provide user defined MDF graphs as (parts of)
 * skeletons. <br>
 *
```

```
 * ParCompute items can be used where Compute items are. Unlike Compute items,
 * requiring the user provides the Task[] compute(Task[]) method, the ParCompute
 * items must be created providing a MdfGraph macro data flow graph, with a
 * single token in input and a single token in output. <br>
 * The computation of the ParCompute goes through the supplied graph rather than
 * compiling MDF instructions using the compute method as opcode.<br>
 *
 *<p>
 *Sample use is as follows:
 *
 *<pre>
                Manager manager = new Manager();

                Compute inc1 = new Splitter();

                IntegerIdGenerator intIdGen = new IntegerIdGenerator();

                InstructionId i1_Id = new InstructionId(intIdGen.generateId());
                InstructionId i2_Id = new InstructionId(intIdGen.generateId());
                InstructionId i3_Id = new InstructionId(intIdGen.generateId());
                InstructionId i4_Id = new InstructionId(intIdGen.generateId());

                Destination d1 = new Destination(
                                new InputPositionId(new IntegerIdentifier(0)), i2_Id, Mdfi.NoGraphId);

                Destination d2 = new Destination(
                                new InputPositionId(new IntegerIdentifier(0)), i3_Id, Mdfi.NoGraphId);

                Destination[] dests = {d1, d2};
                Mdfi i1 = new Mdfi(manager,i1_Id,inc1,1,2,dests);

                Compute sq1 = new Square();

                Destination d3 = new Destination(
                                new InputPositionId(new IntegerIdentifier(0)), i4_Id, Mdfi.NoGraphId);

                Destination[] dests1 = {d3};

                Mdfi i2 = new Mdfi(manager, i2_Id, sq1, 1, 1, dests1);

                Compute sq2 = new Square();

                Destination d4 = new Destination(
                                new InputPositionId(new IntegerIdentifier(1)), i4_Id, Mdfi.NoGraphId);

                Destination[] dests2 = {d4};
                Mdfi i3 = new Mdfi(manager, i3_Id, sq2, 1, 1, dests2);

                Compute add = new Comp();

                Destination d5 = new Destination(
                                new InputPositionId(new IntegerIdentifier(0)), Mdfi.NoInstrId,
Mdfi.NoGraphId);

                Destination[] dests3 = {d5};

                Mdfi i4 = new Mdfi(manager, i4_Id, add, 2, 1, dests3);

                MdfGraph graph = new MdfGraph();

                graph.setInputInstruction(i1);
                graph.addInstruction(i1);
                graph.addInstruction(i2);
                graph.addInstruction(i3);
                graph.addInstruction(i4);
                graph.setOutputInstruction(i4);

                ParCompute main = new ParCompute(graph);

 </pre>
 * In this case a graph not expressible with the standard muskel
 * skeletons is built. <br>The graph can be used in a farm,
 * as an example, by telling
 <pre>
 Farm f = new Farm(mdfg);
```

```
 manager.setProgram(f);
 </pre>
 * or computed as is by telling
 <pre>
 manager.setProgram(mdfg);
 </pre>
 * just before issuing the <code> manager.compute()</code> call.
 *
 * @author marcod
 */
public class ParCompute extends Compute {

	private static final long serialVersionUID = 1L;

	protected MdfGraph program = null;

	/**
	 * ParCompute nodes are created by passing an MDF graph as the node
	 * "program"
	 *
	 * @param g
	 *            the graph to be computed
	 */
	public ParCompute(MdfGraph g) {
		program = g;
	}

	/**
	 * This is the method that has to be computed in order to implement the
	 * Compute interface. It is currently null (actually, it always returns a
	 * null, but it can be derived interpreting on the fly the MDF graph.<br>
	 * Users can extend the class providing an actual method, but this will
	 * not be used but for computing the sequential program.
	 *
	 * @param task is the input task value
	 * @return the result of the computation
	 */
	public Task[] compute(Task[] task) {

		return program.compute(task);
	}

	/**
	 * this is the method used to fetch the graph when compiling
	 * the class as a skeleton parameter.
	 *
	 * @return the associated MDF graph implementing the wrapped code
	 *                      (may be parallel)
	 */
	public MdfGraph getMdfGraph() {
		return program;
	}

	public Object clone() {
		return new ParCompute((MdfGraph) program.clone());
	}

}
```

Pipeline.java

```
/* This is the muskel: a skeleton/RMI based, parallel programming library
Copyright (C) 2006  Marco Danelutto, Dept. Computer Science, Univ. Pisa, Italy

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.
```

```
You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA. */

/*
 * Created on Mar 22, 2004
 *
 * To change the template for this generated file go to
 * Window - Preferences - Java - Code Generation - Code and Comments
 */
package skeleton;

import muskel.mdf.Task;


/**
 * This is the pipeline skeleton class.
 * The muskel pipeline is a two stage pipeline.
 * Multiple stage pipelines can be defined nesting
 * two stage pipelines. That is, provided that
 * <code>StageX</code> are all skeletons
 * (i.e. they implement the Compute interface)
 * the following code represents a three stage pipeline:
 * <pre>
 * Pipeline p1 = new Pipeline(Stage1, Stage2);
 * Pipeline main = new Pipeline(p1, Stage3);
 * </pre>
 *
 * This is for the sake of simplicity. There is no
 * problem in implementing an n-stage pipeline,
 * using a Vector of stages instead of a fixed set
 * of variables hosting the stages.
 * <p>
 *
 * @author Danelutto Marco
 *
 */
public class Pipeline extends Compute {

	private static final long serialVersionUID = 1L;

	Compute stage1 = null;
	Compute stage2 = null;

	/**
	 * constructs a two stage pipeline
	 * @param stage1 is the skeleton representing
	 *                     the computation to be performed at the first stage
	 * @param stage2 is the skeleton representing
	 *                     the computation to be performed at the second stage
	 */
	public Pipeline(Compute stage1, Compute stage2) {
		this.stage1 = stage1;
		this.stage2 = stage2;
	}

	/**
	 * This returns the first stage, used in the MDF conpiler
	 * @return The first stage
	 */
	public Compute getFirstStage() {
		return stage1;
	}
	/**
	 * this returns the second stage, used in the MDF compiler
	 * @return The second stage
	 */
	public Compute getSecondStage() {
		return stage2;
	}
	/**
	 * computes the pipeline sequentially.
	 * @param task is the data input to be processed by the pipeline
	 * @return the result computed by the pipeline as a Task[]
	 */
```

```
	public Task[] compute(Task[] task) {
		Task[] stage1Result = stage1.compute(task);

		return stage2.compute(stage1Result);
	}
}
```

SECompute.java

```
package skeleton;

import muskel.mdf.Task;

/**
 * Compute con Side Effects, cioe' con la capacita' di generare
 * un nuovo task da calcolare.
 * La sottoclasse dovra' implementare il solo metodo compute, all'interno della quale
 * potra' impostare il task generato con il metodo setGeneratedTask.
 * <br>
 * E' cruciale che la generazione dei task sia condizionata dall'input del metodo compute,
 * altrimenti si creeranno dei loop che porteranno alla generazione continua di task.
 * <br>
 * <br>
 * Es. Nel corpo del metodo compute potremmo avere (posto x l'input):
 * <blockquote><pre>
 * if(x > y) setGeneratedTask(new Task[]{new Task(newTask)});
 * </pre></blockquote>
 * Se SECompute e' condiviso da piu' thread, sara' necessaria la sincronizzazione.
 * Nello specifico e' necessario rendere atomico il calcolo e il recupero dei task generati.
 * <br>
 * <br>
 * Es. Posto seComp, di tipo SECompute, il codice per il calcolo e il recupero dei task generati
deve essere:
 * <blockquote><pre>
 * Task[] generatedTask = null;
 * Task[] result;
 *
 * synchronized (seComp) {
 *      result = (Task[]) seComp.compute(tokenValues);
 *
 *      Task[] gtTemp = seComp.getGeneratedTask();
 *
 *      if(gtTemp != null) {
 *
 *              generatedTask = new Task[gtTemp.length];
 *              //clono i task generati
 *              for(int i = 0; i< gtTemp.length; i++)
 *                      generatedTask[i] = (Task) gtTemp[i].clone();
 *
 *              //reimposto il valore per il prossimo utilizzo
 *              seComp.resetGeneratedTask();
 *      }
 * }
 * </pre></blockquote>
 * */
public abstract class SECompute extends Compute {

	private Task[] generatedTasks = null;

	/**
	 * Restituisce il task generato dalla SECompute
	 * */
	public Task[] getGeneratedTask() {
		return generatedTasks;
	}

	/**
	 * Imposta il task specificato come task generato dalla SECompute.
	 * Va utilizzato nel metodo compute
	 * */
	protected void setGeneratedTask(Task[] tasks) {
		generatedTasks = tasks;
	}
```

```
        /**
         * Reimposta i task generati dalla SECompute.
         * */
        public void resetGeneratedTask(){
                generatedTasks = null;
        }

        @Override
        public abstract Task[] compute(Task[] task);

}
```