Java Distribuited Computing Enviroment

Reti di Calcolatori LS
prof. Antonio Corradi
a.a. 2003/2004

Relazione di
Poggiali Antonio

Sommario

Sommario **2**

Introduzione a JDCE **3**

Calcolo distribuito 3
Le componenti del sistema 4
Client 4
Customer 4
Manager 5
Affidabilità rispetto alla caduta dei nodi 5

Deployment di una Computazione **6**

DataPool 6
Operation 7
ResultCollector 7
Ricircolo dei pacchetti 8
ComputationInitializer 8

Interazione fra le componenti **8**

Identificazione di client, customer e recupero degli oggetti remoti 9
Il controllo dei permessi 9
Componenti attivi e passivi 10
Rilevazione dei guasti 10
La replicazione del manager 10

Il manager **11**

manager.IClient 11
manager.ICustomer 12
Guasti a client e customer 13

Il customer **13**

customer.IManager 13
customer.IClient 13
Terminazione del customer 14

Il client **15**

client.IManager 15

Conclusioni **15**

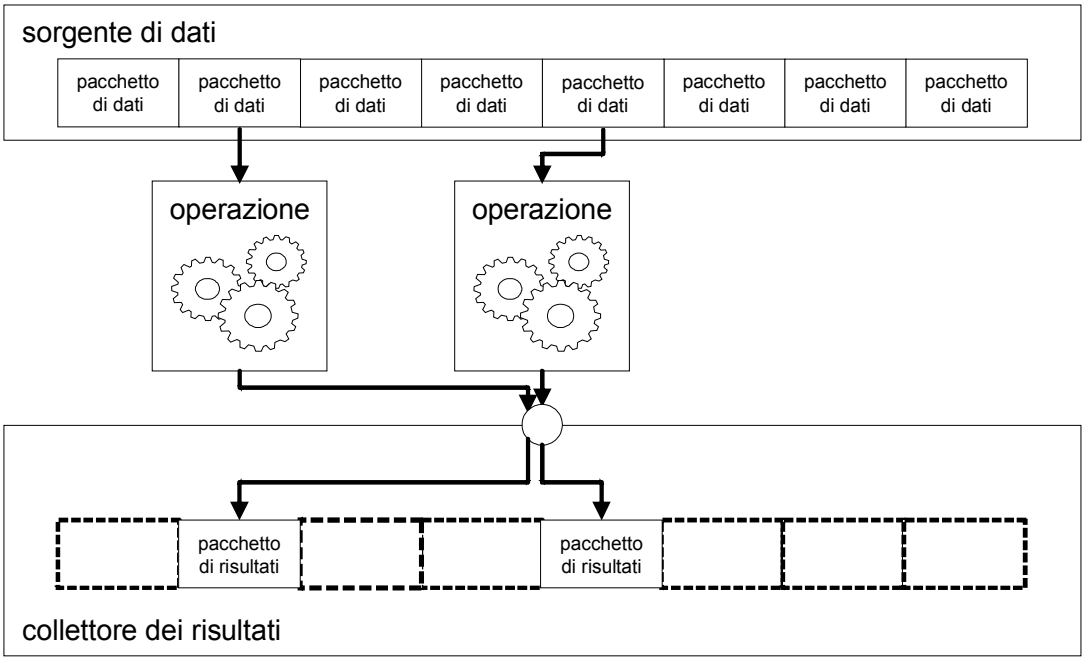
Computazione locale vs. distribuita 15
Il prototipo del sistema e gli sviluppi futuri 16
Politica di distribuzione della potenza di calcolo 17
Politica di distribuzione dei dati 17
Comportamento del client 18

Introduzione a JDCE

JDCE è un ambiente per lo sviluppo e l'esecuzione di applicazioni di calcolo distribuito di tipo cooperativo. In JDCE le macchine che partecipano al calcolo offrendo le proprie risorse non sono note a priori ma vengono liberamente messe a disposizione (e liberamente tolte) dagli utenti delle macchine stesse. JDCE gestisce autonomamente la distribuzione della potenza di calcolo disponibile fra i vari richiedenti e gestisce le problematiche che nascono dall'ambiente distribuito. JDCE offre al programmatore un sistema semplice, basato su interfacce ed ereditarietà, per strutturare e sviluppare le proprie applicazioni di calcolo e per mandarle in esecuzione. JDCE è sviluppato in Java ed utilizza RMI come middleware di comunicazione fra i vari componenti.

Calcolo distribuito

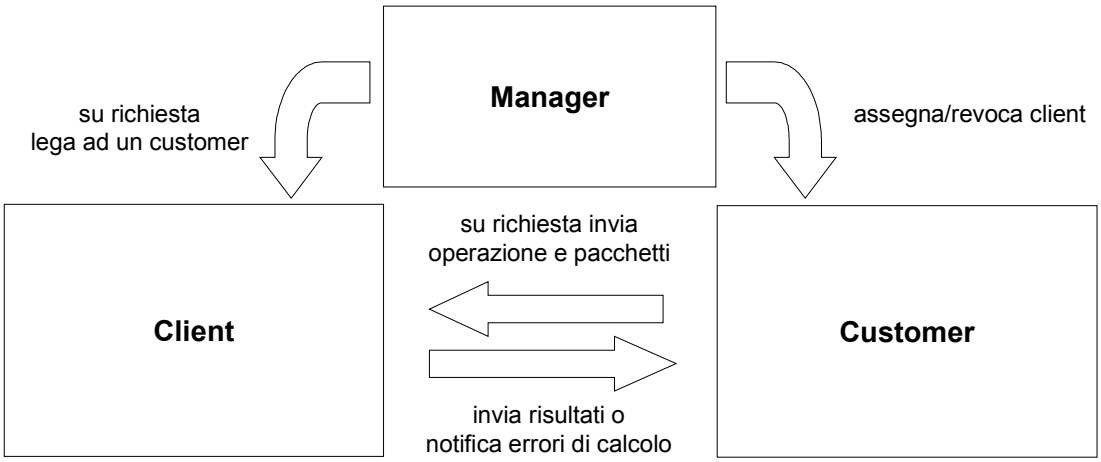
In JDCE un'applicazione di calcolo viene chiamata **computazione** (computation) Una computazione consiste in una serie di **pacchetti** di dati: a ciascun pacchetto viene applicata una **operazione** che lavora solo sui dati che questo contiene e restituisce un nuovo pacchetto che contiene il risultato. JDCE distribuisce il calcolo affidando i vari pacchetti a calcolatori diversi e raccogliendo poi i risultati che questi producono: man mano che i risultati sono raccolti vengono presentati alla computazione..



Ogni computazione è quindi fisicamente articolata su una terna di elementi distinti che il programmatore deve sviluppare: la **sorgente dei dati** (data pool), l'**operazione** da eseguire (operation) ed il **collettore dei risultati** (result collector).

Le componenti del sistema

Gli attori che partecipano al sistema JDCE sono di tre tipi: i **client**, i **customer** ed i(l) **manager**: i client sono gli attori che mettono a disposizione la loro capacità di elaborazione, i customer (committenti) sono gli attori che richiedono capacità di calcolo per eseguire una computazione, il manager è l'attore che si occupa di gestire la capacità di calcolo disponibile (i client disponibili) fra le varie computazioni pervenute (i customer che hanno fatto richiesta); mentre client e customer sono molteplici per ogni ambiente JDCE in funzione c'è un solo attore manager. Ogni attore consiste fisicamente in una macchina connessa alla rete su cui gira una applicazione che svolge determinate operazioni.



Client

L'applicazione client di JDCE, brevemente, si i**scrive** (subscribe) ad un manager e chiede di essere **legato** (bind) ad un customer. Una volta legato il client richiede al customer che gli mandi il codice dell'**operazione**, ed una volta ottenuto chiede un **pacchetto di dati** su cui eseguirlo. Ogni volta che un pacchetto viene elaborato il client comunica al customer il risultato e gli chiede un altro pacchetto. Quando il customer ha concluso la propria elaborazione ne informa il client che ricomincia da capo chiedendo al manager di essere legato. L'utente sulla macchina su cui l'applicazione client esegue può decidere di terminarla in qualsiasi momento, in quel caso il client si **cancella** (unscribe) dal manager

Customer

L'applicazione customer di JDCE richiede che le venga fornita una **computazione** (un'applicazione di calcolo) da portare a termine. Il customer fa da ponte tra i client e la terna di componenti della computazione. Per prima cosa il customer si iscrive al manager (in

maniera simile al client) ed il manager, quando ne ha disponibili, gli **assegna** (grant) uno o più client con cui poter lavorare. Le assegnazioni sono temporanee, il manager potrebbe **revocarle** (revoke) in qualsiasi momento. Il customer serve solo le richieste che gli arrivano dai client che il manager gli ha assegnato. Quando la computazione è terminata il customer si cancella (unscribe) dal manager. I suoi compiti fondamentali nel portare avanti la computazione sono: gestire la perdita di pacchetti dovuti alla caduta dei client replicando le richieste, sterminare eventuali risultati "duplicati" e consegnare alla computazione i risultati in ordine, compensando le differenti velocità dei client.

Manager

L'applicazione manager di JDCE si occupa di tenere traccia dei client e dei customer che si iscrivono, di rispondere alle richieste di legarsi dei client fornendogli un customer, di comunicare ai customer l'assegnazione di client, di rispondere alle richieste di cancellazione di client, revocandoli al customer a cui sono assegnati, e rispondendo alle richieste di cancellazione dei customer. Il ruolo del manager nella architettura JDCE è fondamentale: è il manager che fa da broker fra i client ed i customer, è il manager che decide come distribuire la potenza di calcolo ed è sempre il manager che, nel caso di fallimenti di client o customer, si fa carico della gestione degli errori che ne risultano.

Affidabilità rispetto alla caduta dei nodi

Uno degli obiettivi di JDCE è fornire una infrastruttura solida a cui fare affidamento; il programmatore scrive il codice di una computazione e lo affida ad un customer per portarla a termine senza doversi preoccupare di cosa fanno i client o il manager. Per questo motivo il sistema, nel caso che il nodo manager cada, permette ai customer di continuare a lavorare con i client che gli erano stati assegnati dal manager prima che cadesse.

In questo scenario la caduta del manager impedirà a nuovi attori (customer e client) di entrare nel sistema ed ai client che chiedono di essere legati di ottenere una risposta, uno scenario comunque da evitare. Per ovviare a questi problemi il sistema prevede la possibilità di replicare il manager su più nodi, pronti a subentrare in caso di fallimento del nodo principale.

La caduta di un nodo customer porta alla perdita della computazione che stava portando a termine. La possibilità di replicare nodi customer è stata presa in considerazione ma scartata perché replicare le strutture del customer su macchine diverse sarebbe troppo onerosa in termini di trasferimento dei dati, ad esempio: se un customer sta gestendo una computazione che fa compressione video occorre copiare il video non compresso (la sorgente dei dati) ed il

video compresso (il risultato, man mano che viene calcolato) su tutti i nodi che replicano il customer, parliamo di gigabyte da trasferire!

Deployment di una Computazione

Una computazione è costituita da tre elementi: una sorgente di dati una operazione ed un collettore di risultati. Sviluppare una computazione per JDCE vuol dire scrivere uno o più moduli che implementano le interfacce di questi tre componenti.

DataPool

In JDCE è definita l'interfaccia computation.DataPool che descrive il comportamento che una sorgente di dati deve avere. Il customer sfrutta questi metodi per recuperare i pacchetti dai inviare ai clienti. Ogni pacchetto deve essere identificato da un numero che lo caratterizza univocamente rispetto a tutti gli altri pacchetti del DataPool.

```
package computation;
public interface DataPool {
   public Packet getNextPacket()
        throws NoMorePacketException;
   public Packet getPacket(int number)
        throws InexistingPacketException;
   public boolean hasPacket();
   public void inexistingPacketHandler();
   public void inexistingPacketHandler(int number);
}
```

La sorgente di dati è vista come un flusso sequenziale di pacchetti numerati in ordine crescente che possono essere recuperati con il metodo getNextPacket(); una volta terminata la sequenza dei pacchetti il metodo lancerà l'eccezione NoMorePacketException. Il metodo hasPacket() restituisce vero se la getNextPacket() ritorna un pacchetto, falso altrimenti. La sorgente permette anche l'accesso random ai pacchetti con il metodo getPacket(int number) in cui occorre specificare il numero del pacchetto desiderato, se questo non esiste viene lanciata una eccezione InexistingPacketException.

I due metodi inexistingPacketHandler servono al customer per segnalare, prima di abortire la computazione, che la richiesta di un pacchetto (sequenziale o random) che doveva essere completata correttamente ha lanciato un'eccezione inaspettata; e quindi c'è un errore nella logica del DataPool.

Operation

In JDCE è definita la classe astratta computation.Operation.

```
package computation;
import java.io.Serializable;
public abstract class Operation implements Serializable{
  public abstract Serializable compute(Packet data);
}
```

Questa classe (e le sue figlie) descrivono le operazioni che i client eseguono sui pacchetti; è serializzabile perché deve venire trasferita ai client quando ne fanno richiesta.

Invocando il metodo compute() si ottiene il risultato dell'elaborazione sul pacchetto passato come parametro, risultato che deve essere serializzabile per poter essere restituito al customer. Il client costruisce i pacchetti di risultati invocando il costruttore della classe ResultPacket e passandogli come argomenti un pacchetto di dati e la classe Operation da utilizzare; il costruttore di ResultPacket garantisce che il pacchetto risultato abbia lo stesso numero del pacchetto di dati da cui è calcolato.

ResultCollector

In JDCE è definita l'interfaccia computation.ResultCollector che descrive il comportamento che un collettore dei risultati deve avere. Il customer sfrutta questi metodi per consegnare alla computazione i pacchetti di risultati inviati dai clienti.

```
package computation;
public interface ResultCollector {
  public void commit(ResultPacket result);
  public boolean hasFinished();
  public void signalError(int packetIndex)
       throws AbortComputationException;
}
```

Il metodo commit(ResultPacket result) consegna il pacchetto di risultati. Il metodo signalError(int packetIndex) indica che l'elaborazione del pacchetto con il numero packetIndex ha causato un errore sul cliente (ad esempio una divisione per zero) e quindi non è disponibile alcun risultato. Se il metodo lancia l'eccezione AbortComputationException richiede al customer di abortire la computazione. Il customer di JDCE consegna i risultati o notifica gli errori rigorosamente in ordine e senza duplicati. Il metodo hasFinished() deve restituire vero quando tutti i pacchetti della computazione sono stati consegnati (o hanno restituito errore)

Ricircolo dei pacchetti

È possibile utilizzare i pacchetti di risultati che vengono restituiti dal customer per costruire nuovi pacchetti dati per il DataPool costruendo una elaborazione ciclica. Per il customer, salvo errori, la computazione ha termine quando il metodo hasFinished() restituisce vero.

ComputationInitializer

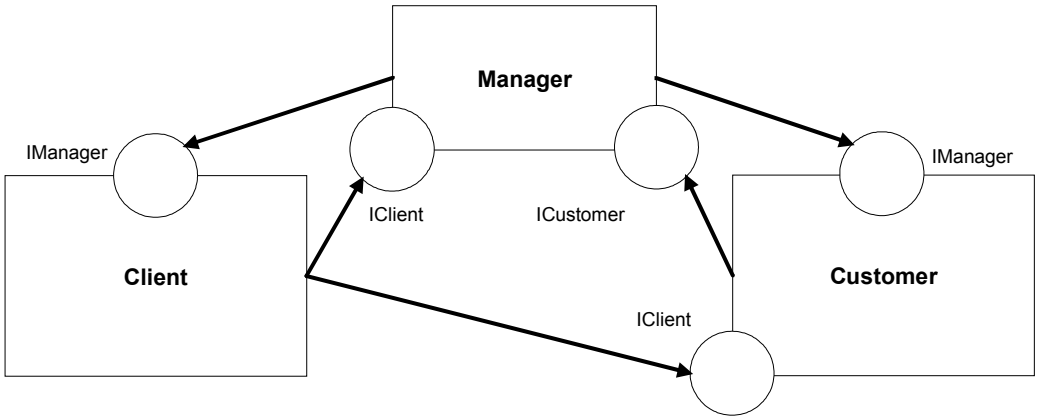
Per istruire il customer su quale DataPool, ResultCollector ed Operation impiegare e su come costruirsi gli oggetti JDCE definisce la classe computation.ComputationInitializer.

```
package computation;
public abstract class ComputationInitializer {
  public abstract DataPool getDataPool();
  public abstract ResultCollector getResultCollector();
  public abstract Operation getOperation();
  public ComputationInitializer(String[] args) {}
}
```

Quando il customer viene avviato gli deve venire indicato il nome il nome di una sua classe figlia, il customer costruirà un oggetto invocando il costruttore e passandogli i restanti parametri della line di comando. Il costruttore dovrà costruire i tre oggetti DataPool ResultCollector ed Operation specifici della computazione che il customer andrà poi a recuperare con i restanti metodi.

Interazione fra le componenti

Le tre componenti fondamentali di JDCE, client, customer e manager, comunicano tra loro utilizzando RMI, ciascun componente definisce una interfaccia di comunicazione specifica per ciascuna delle altre due parti che devono comunicare con lui, ad esempio il manager definisce le due interfacce manager.ICustomer e manager.IClient per poter essere invocato rispettivamente dai customer e dai client, il customer definisce le due interfacce customer.IManager e customer.IClient per poter essere invocato dal manager e dai client mentre il client espone la sola interfaccia client.IManager,



I riferimento agli oggetti remoti che implementano le interfacce devono essere reperiti da un registro RMI; in JDCE viene fatta l'ipotesi (forte) che il nodo che mantiene il registro non fallisce e quindi non ci sono politiche di recovery su registri secondari. La caduta del nodo registro impedirà a nuovi client e customer di reperire il manager, e quindi di iscriversi al servizio, e impedirà ai client di reperire i customer (customer.IClient) in seguito ad una bind, ma non impedirà ai customer di continuare a lavorare con i client che avevano assegnati.

Identificazione di client, customer e recupero degli oggetti remoti

La semantica di RMI non permette all'oggetto remoto di risalire all'identità di chi ne sta facendo uso invocando le sue funzioni. Per identificare i vari client ed i vari customer occorre un sistema per assegnare a ciascuno una identità: al momento di richiedere una operazione l'identità del richiedente sarà fornita come parametro della funzione invocata. Il manager permette a client e customer di reperire nomi unici attraverso il metodo getValidName(), definito nelle sue interfacce estendendo l'interfaccia NameResolver, che restituisce un token. I token sono oggetti serializzabili da cui è possibile estrarre un nome con il metodo getName().

Il token serve anche ad identificare la stringa di bind e di lookup da inviare al registro RMI per iscriversi in modo che chiunque altro, una volta ricevuto il token sia in grado di recuperare un riferimento all'oggetto remoto. Gli spazi dei nomi di manager, client e customer sono ulteriormente separati sul registro RMI: ogni componente quando si iscrive antepone al nome un prefisso standard (costanti di JDCE) che dipende della sua tipologia (client, customer, manager), prefisso che andrà utilizzato anche al momento del lookup.

Il controllo dei permessi

Qualsiasi operazione di client e customer sul manager, o di client sui customer, richiede che venga fornito un token che contiene l'identità del chiamante. Se il chiamante non è autorizzato, per qualunque motivo, ad eseguire quella operazione verrà sollevata un'eccezione InvalidPermissionException. Ad esempio, se un client chiede un pacchetto di dati ad un customer occorre che il manager abbia concesso (grant) il permesso al customer, se il

customer non ha il permesso solleverà una InvalidPermissionException (che abbrevieremo con IPE) al client.

Componenti attivi e passivi

In JDCE le uniche componenti "attive" sono i client. Manager e customer, tranne quando si avviano e terminano, hanno un comportamento reattivo e lavorano per rispondere alle richieste dei client. Anche le comunicazioni fra di loro avvengono come conseguenza di richieste dei client. Manager e customer implementano le loro interfacce con metodi sincronizzati per evitare che richieste concorrenti di più client generino interferenze nell'aggiornamento delle proprie strutture interne (di fatto serializzano le richieste dei client).

Questo scenario introduce la possibilità di blocchi critici! Ad esempio: il client "A" accede al manager, e lo blocca, mentre il client "B" accede al customer "FFT", e lo blocca; a questo punto se per completare l'operazione richiesta da "A" il manager deve invocare un metodo del customer "FFT" e il customer per completare la richiesta di "B" deve invocare un metodo del manager siamo in deadlock! Per evitare blocchi critici il protocollo di comunicazione è strutturato in modo che i customer, tranne all'inizio ed alla fine della loro vita, non comunichino con il manager; in questo modo eliminiamo la possibilità di deadlock.

Rilevazione dei guasti

Se un componente fallisce chi comunica con lui se ne accorge dall'eccezione RemoteException che RMI lancia quando si invoca un metodo di un oggetto remoto scomparso. Gestendo la RemoteException è possibile implementare meccanismi di rilevazione e reazione ai guasti. In JDCE è definita l'interfaccia remota Alive che definisce il metodo boolean alive() allo scopo di chiedere all'oggetto remoto che implementa questa interfaccia se è vivo.

La replicazione del manager

JDCE per migliorare la propria disponibilità prevede che il nodo manager possa essere replicato da una o più copie passive; le copie possono essere aggiunte o rimosse dinamicamente durante l'esecuzione.

Fra le copie a disposizione del manager una viene eletta copia master ed è l'unica che viene mantenuta aggiornata: se la copia master fallisce il manager ne sceglie una fra quelle del gruppo e la elegge a master trasmettendogli tutto lo stato, se il manager fallisce la copia master diventa il nuovo manager ed elegge una nuova copia master.

Il manager espone alle copie una interfaccia che permette loro di entrare nel gruppo, di uscirne, e di chiedere al manager se è vivo (Alive). Le copie espongono una interfaccia che

permette al manager di mantenerle aggiornate sullo stato di client e customer. Durante il normale funzionamento del manager ad ogni operazione richiesta da parte di client e customer, prima di restituire i risultati al chiamante, il manager sincronizza la copia master trasmettendo le modifiche che l'operazione ha generato nelle sue strutture dati. La copia master rimane in attesa aggiornando il proprio stato ed invocando periodicamente alive() sul manager per controllarne la sopravvivenza.

Quando il manager fallisce i componenti client e customer se ne accorgono a causa della RemoteException che i metodi del defunto manager sollevano. Una volta rilevato il guasto i componenti client e customer eseguono un protocollo che consiste in un periodo di attesa seguito dall'interrogazione del registro RMI per ottenere un nuovo riferimento al manager e nella ritrasmissione della richiesta fallita. Durante il tempo di attesa di client e customer la copia master accede al registro RMI e si registra al nome del vecchio manager in modo che gli altri componenti, appena attivi, possono reperirla ed utilizzarla come nuovo manager.

Questo protocollo grazie all'unica copia sincronizzata ed alla semplicità della gestione del gruppo risulta molto leggero ma è a rischio di fallimento se manager e copia master falliscono contemporaneamente; la possibilità che questo accada è comunque oggettivamente remota.

Il manager

Il manager ha due interfacce, una per i client ed una per i customer. Entrambe estendono l'interfaccia NameResolver che definisce il metodo getValidName() per il recupero di un token che contiene un nome unico nel sistema JDCE. Per brevità ometteremo da tutti i metodi di tutte le interfacce presentate le dichiarazioni di lancio della RemoteException.

manager.IClient

```
package manager;

public interface IClient extends Remote, NameResolver {

   public void subscribeClient (Token token)
        throws InvalidPermissionException;

   public void unscribeClient (Token client)
        throws InvalidPermissionException;

   public String bindClient (Token client)
        throws InvalidPermissionException;

}
```

Il metodo subscribeClient permette ai client di iscriversi a JDCE fornendo un token di identità; il manager crea le strutture necessarie a "tracciare" il client, fra cui un riferimento remoto all'oggetto. Se un'altro client (o lo stesso client) si sono iscritti con lo stesso nome

(contenuto nel token) o se il manager non trova nel registry RMI un riferimento al client verrà lanciata l'eccezione IPE.

Il metodo unsubscribeClient elimina il client da JDCE. Se il client risultava legato ad un customer occorrerà informarlo con una revoca dell'assegnamento. La IPE viene sollevata se il token contiene un nome inesistente.

Il metodo bindClient fornisce al client il nome del customer a cui si deve legare. Il manager nelle sue strutture dati tiene traccia dello stato dei client etichettandoli come liberi o legati (e a chi sono legati). Il metodo bindClient provvede anche ad informare il customer a cui il client verrà legato (vedere l'interfaccia customerIManager). Se un client che chiede di essere legato risulta già etichettato come "legato" ci sono tre diverse possibilità: il customer è fallito ed il client è tornato a chiedere una bind, il customer sta terminando ed ha liberato il client (vedi dopo), oppure il customer ha liberato il client perché al momento non ha pacchetti da affidargli. Il manager verifica cosa sia successo e prende le dovute contromisure. Se non ci sono customer il client viene sospeso, verrà risvegliato appena un customer si renderà disponibile. L'eccezione IPE viene sollevata se il token contiene un nome inesistente.

manager.ICustomer

```
package manager;

public interface ICustomer extends Remote, NameResolver {

   public void subscribeCustomer (Token token)
        throws InvalidPermissionException;

   public void unscribeCustomer (Token token)
        throws InvalidPermissionException;

}
```

I metodi subscribeCustomer ed unscribeCustomer sono del tutto simili a quelli subscribe/unscribe dell'interfaccia IClient, a differenza che lavorano con i customer. Quando un customer si iscrive il manager controlla se ci sono client sospesi sulla bind e li risveglia; quando si cancella il manager etichetta come "liberi" tutti i client che gli erano legati (NB al client non viene notificato niente).

Più complessa è invece la terminazione dei customer: dal momento in cui la computazione termina al momento in cui viene invocata la unscribeCustomer può passare diverso tempo. In questo lasso di tempo è facile che almeno un client che gli era legato torni a chiedere una bind al manager: in questo caso il customer viene "cancellato d'ufficio" dal manager.

Guasti a client e customer

Il manager ha la responsabilità di rilevare i guasti di client e customer, in modo da eliminare dalle sue strutture registrazioni di componenti terminati che non hanno invocati i

metodi di cancellazione. Nel caso il manager si accorga di un guasto tenta anche di "pulire" il registro RMI eliminando la registrazione dell'oggetto remoto del componente fallito; normalmente sono i componenti stessi a eliminare le proprie registrazioni sul registro RMI. A questo scopo le interfacce per il manager di client e customer estendono l'interfaccia Alive ed il manager durante la normale esecuzione dei metodi esegue dei controlli per evitare "zombi" nelle sue strutture.

Il customer

Anche il customer ha due interfacce: una per il manager ed una per i client.

customer.IManager

```
package customer;
public interface IManager extends Remote, Alive {
   public void grantBind(Token token)
        throws InvalidPermissionException;
   public void revokeBind(Token token)
        throws InvalidPermissionException;
}
```

L'interfaccia estende anche Alive ereditando il metodo alive(). Nel customer il metodo alive ha un ulteriore applicazione; restituendo falso informa il manager che la computazione è finita, quindi non ha più bisogno dei client, ma non ha ancora invocato il metodo unscribeCustomer.

Il metodo grantBind() informa il customer che il manager gli ha assegnato il client identificato dal token passato come parametro. L'interruzione IPE viene sollevata se il client risultava già assegnato al customer.

Il metodo revokeBind() informa il customer che non ha più il permesso di utilizzare il client identificato dal token passato come parametro. L'interruzione IPE viene sollevata se il client non risultava assegnato al customer.

customer.IClient

```
package customer;
public interface IClient extends Remote {
   public Operation getComputation(Token token)
        throws InvalidPermissionException, NoMorePacketException;
   public Packet getPacket(Token token)
        throws nvalidPermissionException, NoMorePacketException;
   public void setComputationResult(Token token, ResultPacket result)
        throws InvalidPermissionException, NoMorePacketException;
```

```
  public void notifyComputationError(Token token, int packetIndex)
       throws InvalidPermissionException, NoMorePacketException;
}
```

L'eccezione NoMorePacketException (che abbrevieremo con NMPE) viene lanciata da tutti i metodi dell'interfaccia ed ha per tutti lo stesso significato: il customer al momento non ha più pacchetti da fare elaborare al client. Il client come risposta tornerà dal manager a chiedere di essere legato (bind). L'eccezione InvalidPermisioneException (IPE) viene lanciata da tutti i metodi dell'interfaccia quando il client non risulta assegnato al customer. Alcuni metodi possono lanciare la IPE anche in altre situazioni.

Il metodo getComputation una istanza della classe Operation specifica della computazione che il customer sta gestendo. L'IPE può venire sollevata anche nel caso in cui al customer risulti che il client ha già ricevuto una volta l'istanza di operation.

Il metodo getPacket restituisce al client un pacchetto di dati da elaborare, il customer nelle sue strutture dati tiene traccia di quali pacchetti ha affidato ai client. L'IPE può venire sollevata anche quando il client ha già ricevuto un pacchetto e non ha ancora consegnato i risultati e quando il client non ha scaricato la classe Operation.

I metodi setComputationResult e notifyComputationError vengono invocati dai client per restituire i dati elaborati o per segnalare che c'è stato un errore durante l'elaborazione. L'IPE può venire sollevata anche quando il client consegna risultati di un pacchetto diverso da quello che gli era stato affidato (al limite il client non aveva alcun pacchetto) e quando il client non ha scaricato la classe Operation.

Terminazione del customer

Spieghiamo nei dettagli perché tra il momento in cui la computazione termina ed in momento in cui il customer invoca la unscribeCustomer c'è un certo tempo. Ogni volta che un pacchetto o un errore di calcolo vengono consegnati al ResultCollector della computazione il customer verifica tramite il metodo hasFinished se la computazione è finita. A questo punto la cosa più logica sarebbe informare immediatamente il manager ed invocare la unscribeCustomer ma per le considerazioni sui deadlock che abbiamo fatto è evidente che rischieremmo il blocco critico del customer e del manager! La soluzione che il customer adotta è quella di lanciare una nuova attività (thread) che si metterà in coda al manager (che serializza tutte le richieste) per invocare la unscribeCustomer.

Il client

Il client è l'unico componente veramente attivo di JDCE, infatti sono i client con le loro richieste a far progredire le computazioni agendo sul manager e sui customer.

Il client espone un'unica interfaccia per il manager.

client.IManager

```
package client;
public interface IManager extends Alive{
}
```

L'unica funzione dell'interfaccia è permettere al manager di invocare alive() per controllare che il client non sia fallito.

Conclusioni

Computazione locale vs. distribuita

Quando il modello di computazioni distribuita di JDCE può essere utilizzato per avere uno speed-up? Per capirlo occorre prima analizzare le prestazioni del modello di elaborazione locale in pipeline.

Nel modello locale in pipeline un processo legge i pacchetti di dati e li passa ad un altro processo che li elabora e poi li passa ad un'altro processo che li presenta all'utente; tipicamente lettura e presentazione avvengono su disco. Quando i tempi di lettura/scrittura sono tali da rendere "trascurabili" i tempi di elaborazione la computazione si dice I/O bound. Sicuramente le computazioni I/O bound non sono efficienti per il calcolo distribuito: il punto critico è la lettura e la finalizzazione dei dati, la distribuzione dell'elaborazione non elimina questi tempi, anzi introduce l'overhead dei tempi di trasmissione.

Per le computazioni CPU bound invece lo scenario cambia. Nel modello locale in pipeline se ci sono N pacchetti e ciascuno richiede un tempo T_e di elaborazione il tempo complessivo di elaborazione sarà

$$T_{e-locale} = NT_e$$

(i tempi di lettura e scrittura dei dati sono trascurabili perché minori di T_e). Distribuendo la computazione dobbiamo considerare i tempi T_{send} e $T_{receive}$ per la trasmissione e la ricezione di un pacchetto di dati. A differenza del modello locale che lavora in pipeline in JDCE non è possibile che un cliente ottenga un pacchetto mentre ne sta elaborando un'altro (sarebbe uno

sviluppo futuro molto interessante però). Nella migliore delle ipotesi, senza ritrasmissioni, la sorgente deve trasferire via rete tutti i pacchetti e ricevere tutti i risultati con un tempo complessivo di

$$T_{rete} = N(T_{send} + T_{receive})$$

a cui bisogna aggiungere il tempo che i client impiegano ad elaborare i pacchetti, approssimabile con

$$T_{e-jdce} = \frac{NT_e}{\#client}$$

sotto l'ipotesi che il numero di client assegnati rimanga costante per tutta la computazione.

Ponendo

$$T_{rete} + T_{e-jdce} < T_{e-locale} \text{ otteniamo}$$

$$N(T_{send} + T_{receive}) + \frac{NT_e}{\#client} < NT_e \text{ da cui si ricava}$$

$$T_e > \frac{(T_{send} + T_{receive})\#client}{\#client - 1}$$

che rappresenta la condizione "minima" (lower bound) sul tempo di elaborazione T_e dei pacchetti che rende JDCE conveniente rispetto al modello locale in pipeline. Questa disuguaglianza per un numero di client elevato diventa

$$T_e > (T_{send} + T_{receive})$$

In pratica per avere un miglioramento delle performance di una elaborazione utilizzando JDCE la condizione necessaria è che il tempo di elaborazione di un pacchetto sia maggiore del tempo di trasmissione dei dati e dei risultati. Non è però detto che la relazione sia sufficiente: nel ricavarla non abbiamo considerato gli overhead che JDCE introduce per il coordinamento delle componenti.

Il prototipo del sistema e gli sviluppi futuri

A conclusione della fase di studio è stato realizzato un prototipo del sistema completo di tutte le componenti e provvisto di tutte le funzionalità tranne, per il momento, la replicazione del manager e la consegna in ordine dei pacchetti di risultati al ResultCollector. Il prototipo è composto da tre diversi programmi eseguibili, uno per ogni componente, dotati di interfaccia da linea di comando sviluppata con CLI (Command Line Interfaces, http://jakarta.apache.org/commons/cli/) e capacità di log file degli output con due diversi livello di verbose.

Politica di distribuzione della potenza di calcolo

A seconda della politica di distribuzione della potenza di calcolo disponibile che si decide di adottare si ottengono risultati diversi in termini di throughput (numero di customer completati), latenza (tempo di attesa per il completamento), parallelismo ed overhead di comunicazione. Il manager assume che tutti i client abbiano la stessa potenza di calcolo e la stessa banda: sotto queste ipotesi la potenza di calcolo assegnata ad un customer coincide con il numero di client assegnati.

Una politica paritaria di ripartizione della potenza di calcolo cercherebbe di ripartire equamente i client fra i customer; quando un nuovo customer si iscrive il manager revoca alcuni client ai customer già iscritti e li assegna a quello nuovo. All'altro estremo delle possibilità, passando per tutte le politiche basate su priorità (fissa, mobile, rotante, ecc.) c'è la politica FIFO. La FIFO assegna tutta la potenza disponibile al primo customer che ne fa' richiesta; quando questo termina tutti i client vengono assegnati al secondo e così via.

Il vantaggio della politica FIFO rispetto a tutte le altre è che minimizza l'overhead di comunicazione: una volta assegnati i client non vengono più revocati questo si traduce in un minor numero di chiamate ad oggetti remoti. Al momento il prototipo di JDCE che è stato realizzato utilizza una politica FIFO; sia per limitare l'overhead di comunicazione, sia perché è facilmente implementabile.

Per le prossime versioni del sistema sarà necessario uno studio "formale" dell'efficienza delle varie politiche per scegliere la migliore o, ancora meglio, permettere all'utente la scelta della politica di distribuzione della potenza di calcolo.

Politica di distribuzione dei dati

Il customer ha il compito di portare a termine la computazione di tutti i pacchetti di dati che il DataPool fornisce e di consegnare in ordine i risultati al ResultCollector. Nel prototipo che è stato implementato il customer si avvale di tre strutture: una lista dei pacchetti letti, una lista dei pacchetti ritornati ed una lista dei pacchetti in attesa di consegna; in verità le prime due strutture contengono solo i numeri dei pacchetti.

Quando un pacchetto viene letto in accesso sequenziale dal DataPool il suo numero viene inserito nella lista di quelli letti. Quando un client consegna un pacchetto il suo numero viene tolto dalla lista di quelli letti ed inserito in quella dei consegnati, a meno che non fosse già nella lista dei consegnati, in questo caso viene scartato perché è un duplicato. I pacchetti consegnati dai client vengono restituiti al ResultCollector in sequenza, se un pacchetto di risultati è fuori sequenza viene temporaneamente inserito nella lista dei pacchetti in attesa di consegna. Il metodo getPacket() tenta sempre un accesso sequenziale al DataPool; se questo

non è possibile estrae dalla lista dei pacchetti letti il numero più basso, accede in modo random al DataPool estraendo il pacchetto con quel numero e lo restituisce al client.

Questa politica di gestione è estremamente semplice (per non dire rozza) ma evita di utilizzare time-out per la scadenza della consegna ed evita di introdurre nella comunicazione client-customer chiamate al client per sapere se è ancora vivo (che implicherebbero chiamate al manager per informarlo della morte dei client); per contro introduce un certo grado di replicazione delle elaborazioni e delle trasmissioni e quindi uno spreco di risorse. E' ovviamente necessario studiare una politica migliore per le prossime versioni del sistema.

Comportamento del client

Il client è un componente estremamente stupido. Il suo compito è unicamente quello di elaborare i dati, tutte le decisioni vengono prese dal manager e dai customer.

Per prima cosa il client si inizializza recuperando un riferimento al manager, chiedendo un token di identità, iscrivendosi al registro RMI e poi iscrivendosi al manager. A questo punto il client chiede al manager di essere legato ad un customer; una volta ottenuto il nome recupera dal registro RMI un riferimento all'oggetto remoto e scarica il codice della classe Operation della computazione. Una volta ottenuta la computazione il client chiede un pacchetto di dati, lo computa e ritorna il risultato al customer (o segnala errore) e continua ad eseguire queste tre operazioni fino a che il customer non lo interrompe con una eccezione. Una volta interrotto il client ricomincia il ciclo chiedendo al manager di essere legato ad un customer e così via.

Le politiche di distribuzione della potenza di calcolo e di distribuzione dei pacchetti implementate nel prototipo hanno caratterizzato in modo determinante il codice del client (ed in particolare il suo comportamento alle eccezioni); uno degli sviluppi futuri più importanti sarà quello di "potenziare" il client rendendo disponibili operazioni di controllo remoto da parte di manager e customer.