Università degli Studi di Salerno
Facoltà di Scienze Matematiche, Fisiche e Naturali

Tesi di Laurea

in

Informatica

"ClusterRMI:
Invocazione di Metodi Remoti su Cluster di Computer"

Relatore:

Chiar.mo Prof. Vittorio Scarano

Candidato:

Roberto Capuano

Matr. 56/100031

ANNO ACCADEMICO 2000–2001

Presentazione

Il nostro lavoro è indirizzato a creare una architettura per la invocazione di metodi remoti su un cluster di computer. Il linguaggio di programmazione scelto è Java. Ogni metodo implementa un servizio messo a disposizione del cluster, e la classe a cui il metodo appartiene definisce un pool di servizi.

L'architettura comprende un insieme di librerie di programmazione ed un insieme di protocolli, per la comunicazione tra i nodi del cluster e per la codifica dei dati.

Ringraziamenti

Vorrei ringraziare prima di tutto il professore Alberto Negro ed il professore Vittorio Scarano: senza di loro quest'opera non esisterebbe.

Per il sostegno in diverse forme in diversi momenti, ringrazio la mia famiglia: mio fratello Gino, alle cui insistenti domande ora rispondo dicendo che sì mi sono laureato, e mia madre Anna, che sicuramente starà piangendo adesso.

In ordine sparso vorrei inoltre ringraziare: Rosario De Chiara & Ugo Erra (ricordate cosa dobbiamo fare domani...), i colleghi tesisti: Nadia Romano, Rosi Lovisi. I dottori Delfina Malandrino, Paolo Luongo, Luigi Catuogno, Umberto Ferraro. I futuri dottori: Diamante "Tino" Brogna, Raffaella Grieco, Romina Miele, Nicola De Santis, Paola Di Lieto, Rosaria Memoli. E chi è già dottore: Christian Cozzolino, Salvatore Giuliano, eppoi Rosario Boccia, Maria Licciardi, Tania Cillo, Ada Compagnone.

Inoltre ringrazio: Ann Wollrath (Sun), Ruth Sivilotti (Myricom) per aver risposto alle mie insistenti domande.

Sempre in ordine sparso vorrei ringraziare qualche altro amico: Fabio Cotta, Massimo Paravizzini, Enrico Scapolatiello, Gianni Acampora, Andrea Coppola, che in diverse forme hanno curato l'intrattenimento del sottoscritto.

A mio padre -

dovunque tu sia, vivrai sempre nei miei ricordi.

Indice

1 Introduzione **1**

2 Motivazioni e Stato dell'Arte **3**

2.1 Motivazioni 3

2.1.1 Facilità in Fase di Sviluppo 4

2.1.2 Efficienza in Fase di Esecuzione 4

Elaborazione Intra-Nodo: Protocollo di Serializzazione . . 4

Elaborazione Inter-Nodo: Protocollo di Comunicazione . . 5

2.2 Stato dell'Arte 5

2.2.1 RMI: Remote Method Invocation 5

Serializzazione 6

Introspezione 7

RMI 8

2.2.2 Jaguar 8

JaguarVIA: implementazione del protocollo VIA. 8
PSO: Pre-Serialized Objects 9
2.2.3 Manta 10
2.2.4 UKA Transport 11

3 Architettura 12

3.1 Introduzione 12
3.1.1 Cluster Registry 13
3.1.2 Applicazione Server 14
3.1.3 Applicazione Client 14
3.1.4 User Repository 15
3.2 Analisi Interna 15
3.2.1 RUN 16
3.2.2 RIO 17
Parte Locale 17
Parte Globale 18
3.2.3 CLIO 18
Caricatore di Classi: ClusterClassLoader 18
Generatore di Classi: StubGenerator, SkeletonGenerator . 18
Modificatore di Classi 19
3.3 Uno Scenario d'Uso 21

3.4 Protocolli di Comunicazione 23
3.4.1 Basso Livello: Protocollo GO-BACK-N 23
3.4.2 Livello Intermedio: Protocollo di Serializzazione 26
3.4.3 Alto Livello: Protocollo di Richiesta dei Servizi 27
3.5 Ottimizzazioni Applicate 27
3.5.1 Object Pooling 28
Thread Pooling 28

4 RUN 29

4.1 Architettura 29
4.2 Livello 1: Trasporto 31
4.2.1 Pacchetti 32
Multiplexing / Demultiplexing 33
4.2.2 Istanze di `Transport` Come Oggetti Attivi 34
4.2.3 `NetAddress`: Astrazione di un Indirizzo di Rete 35
4.2.4 Ottimizzazioni Applicate 35
4.3 Livello 2: Sessione 36
4.3.1 Session, SessionSend, SessionReceive 36
4.3.2 Callgram 37
4.3.3 Protocollo di Comunicazione 38
4.3.4 Ottimizzazioni Applicate 40

4.4 Livello 3: Serializzazione 41

4.4.1 Ottimizzazioni Applicate 44

4.5 Livello 4: Exec 45

4.5.1 Implementazione del Pattern Listener 45

4.5.2 Lato Client: Classe `Manager` 46

4.5.3 Lato Server: Classe `Dispatcher` 46

4.5.4 Ottimizzazioni Applicate 47

4.6 Livello 5: Reference 47

4.7 Livello 6: Stub 48

4.8 Livello 6: Skeleton 49

4.9 Livello 7: Server 50

4.10 Livello 7: Client 51

4.11 Scenario d'Uso 51

4.11.1 Invocazione di un Metodo Remoto 51

4.11.2 Ritorno del risultato da una Metodo Remoto 56

4.12 Le Proprietà del Package `run.session` 60

5 RIO: Cluster Registry 62

5.1 Riferimento di un Servizio 62

5.1.1 Riferimento di un Servizio ed Overloading di Java 65

5.2 Architettura 67

5.3 Funzioni Locali 67
5.3.1 Classe `Local` 68
Lato Server 69
Lato Client 70
5.4 Funzioni Globali 70
5.4.1 Lato Server 71
5.4.2 Lato Client 72
5.4.3 Lato Registry 72
5.4.4 Trasferimento di un Riferimento Registry 74
5.5 Scenario d'Uso 75
5.5.1 Registrazione di un Servizio 76
5.5.2 *Lookup* di un Servizio 77

6 CLIO: Cluster ClassLoader 80

6.1 Classi Attive, Reattive e Passive 80
6.1.1 Classi Attive 81
6.1.2 Classi Reattive 82
6.1.3 Classi Passive 82
6.2 ClusterClassLoader 83
6.2.1 Delegation Model di Java 83
6.2.2 Percorso di Ricerca di ClusterClassLoader 86

6.3 Generator: Generatore di Classi 86
6.3.1 ByteCode Engineering 87
6.3.2 BCEL: ByteCode Engineering Library 88
6.3.3 Generazione di una Classe 89
6.3.4 Motivazioni 90
6.3.5 Classi `Transportable` 93
Campo SUID 96
Costruttore di Default 96
Serializzatore: `writeObject()` 97
Deserializzatore: `readObject()` 97
Dimensione di un Oggetto: `sizeOf()` 98
6.3.6 Classi stub 99
6.3.7 Classi skeleton 102
6.4 SUID: Stream Unique IDentifier 103
6.5 Scenario d'Uso 105
6.5.1 Generazione di una Classe stub 105
6.5.2 Generazione di una Classe `FastTransportable` 107

7 Esempio di Applicazione 109

7.1 Definizione di un Pool di Servizi 109
7.2 Implementazione di un Pool di Servizi 110

7.3 Registrazione di un Pool di Servizi 113

7.4 Implementazione di una Applicazione Client 113

7.4.1 Lookup di un Pool di Servizi 116

7.4.2 Invocazione di un Servizio Remoto 116

7.5 Esecuzione dell'esempio 116

8 Futuri sviluppi 118

9 Conclusioni 120

A Proprietà di Sistema 122

A.1 File delle Proprietà 122

A.2 Elenco delle Proprietà di Sistema 123

B Listati 125

B.1 Package: run.transport 125

B.2 Package: run.session 129

B.3 Package: run.exec 132

B.4 Package: run.reference 133

B.5 Package: run.stub_skeleton 134

B.6 Package: rio.registry 135

B.7 Package: clio 136

Elenco delle figure

3.1 Architettura del sistema: una visione dall'esterno. 13

3.2 Architettura del sistema: una visione dell'interno. 16

4.1 Divisione in livelli del modulo RUN. 30

4.2 Divisione in livelli della logica di comunicazione su rete 32

4.3 Multiplexing dei pacchetti. 33

4.4 Demultiplexing dei pacchetti. 35

4.5 Invocazione di un metodo remoto (UML). 53

4.6 Ritorno del risultato da un metodo remoto (UML). 58

5.1 Funzioni di RIO locali in ogni applicazione. 69

5.2 Funzioni di RIO globali al cluster. 71

5.3 Registrazione di un pool di servizi (UML). 78

5.4 Ricerca di un servizio (UML). 79

6.1 Gerarchia di AppClassLoader 84

6.2 Gerarchia di ClusterClassLoader 85

6.3 Generazione di una classe stub (UML). 106

6.4 Generazione di una classe `FastTransportable` (UML) 108

Elenco delle tabelle

3.1 Elenco dei pacchetti del protocollo GO-BACK-N. 25

4.1 Incapsulazione di un Pacchetto in un datagramma UDP. 33

4.2 Formato dei pacchetti usati da `UDPTransport`. 34

4.3 Classificazione dei campi di una istanza di `Callgram`. 38

4.4 Primitive messe a disposizione da `ContainerOutputStream`. . . . 41

4.5 Primitive messe a disposizione da `ContainerInputStream` 42

4.6 Protocollo di serializzazione: produzioni della grammatica. 43

4.7 Protocollo di serializzazione: simboli terminali della grammatica . 44

5.1 Elenco dei riferimenti e loro uso nel sistema. 64

A.1 Elenco delle proprietà di sistema 124

Listati

4 .1 CallgramListener 45

4 .2 Services 51

6 .1 X una classe che implementa Transportable 94

6 .2 X una classe che implementa FastTransportable 95

6 .3 Rs una interfaccia di servizi remoti 99

6 .4 Aegis_Stub 100

6 .5 Aegis_Skeleton 101

7 .1 Definizione di una interfaccia di servizi remoti 110

7 .2 Implementazione di una interfaccia di servizi remoti 111

7 .3 Implementazione di un argomento di una servizio remoto 112

7 .4 Istanziazione di un pool di servizi remoti 114

7 .5 Invocazione di un servizio remoto 115

A .1 File delle Proprietà di Sistema 123

B .1 Packet 125

B .2 Transport 126
B .3 NetAddress 127
B .4 UDPTransport 127
B .5 Callgram 129
B .6 Session 129
B .7 SessionSend 130
B .8 SessionReceive 131
B .9 Manager 132
B .10 Pusher 132
B .11 Dispatcher 132
B .12 CallThread 132
B .13 Service 133
B .14 RemoteServiceReference 133
B .15 RemoteServiceReference 133
B .16 Stub 134
B .17 Skeleton 134
B .18 Local 135
B .19 Global 135
B .20 ClusterClassLoader 136
B .21 StubGenerator 136

B .22 SkeletonGenerator 136
B .23 RWSGenerator 136
B .24 SUID 137

Capitolo 1

Introduzione

> Un cluster di computer è un sistema di elaborazione, parallelo o distribuito, che consiste di un insieme di computer, indipendenti ma connessi tra loro, che lavorano insieme come una singola risorsa integrata di computazione [BB99].

Ciò che differenza un cluster di computer da una semplice rete è lo strato software. Esso fornisce all'utente del sistema una serie di servizi che gli permettono di vedere il cluster come una unica risorsa di elaborazione. Questo viene comunemente detto SSI: Single System Image.

I nodi di un cluster sono sistemi fortemente accoppiati, in cui l'elaborazione effettuata sull'uno influisce sull'altro. Risulta chiaro quindi che un ruolo cruciale riveste la comunicazione tra i nodi. Un sistema di comunicazione lento

prestazionalmente costituisce un collo di bottiglia per il sistema. Cosí come un sistema di comunicazione veloce ma difficile da usare costituisce un limite nella fase di programmazione.

Il nostro obiettivo è stato quindi coniugare prestazioni e facilità di uso. Per rendere il sistema piú facile da usare abbiamo scelto come linguaggio di programmazione Java. Oltre alla facilità esso fornisce un valore aggiunto rappresentato dalla portabilità e dalla sicurezza.

Mentre la sicurezza su un sistema cluster ha un valore relativo, in quanto risulta spesso separato fisicamente dal resto della rete locale, la portabilità al contrario ha una grande importanza. Sistemi cluster eterogenei, non sono solo plausibili, ma rappresentano una realtà diffusa. Per eterogenei intendiamo sia nel tipo di processori utilizzati, sia nel Sistema Operativo installato. Java permette quell'ulteriore astrazione che permette di ignorare le specifiche hardware e software dei singoli nodi. Coniugare questa atrazione, che facilita l'uso, alla efficienza del risultato è una delle sfide verso cui si orienta questa tesi.

Capitolo 2

Motivazioni e Stato dell'Arte

2.1 Motivazioni

Il nostro lavoro è stato indirizzato a rendere facile ed efficiente la programmazione di sistemi cluster in Java. Per fare ciò, i nostri sforzi si sono indirizzati in due direzioni diverse. Ridurre il ciclo di vita di applicazioni distribuite, eliminando l'uso di post-compilatori. E rendere efficiente a run-time il tempo di computazione inter-nodo, ed intra-nodo. Si è cercato di ridurre il tempo intra-nodo mediante un processo di serializzazione leggero, ed allo stesso tempo di ridurre il tempo inter-nodo, mediante l'uso di un protocollo di comunicazione efficiente.

2.1.1 Facilità in Fase di Sviluppo

Il ciclo di vita di applicazioni distribuite in Java è composto dalle seguenti fasi: (1) editing del codice, (2) compilazione, (3) post-compilazione, (4) esecuzione. La post-compilazione serve a generare quelle classi che non servono direttamente all'applicazione, ma all'infrastruttura di comunicazione. Sono facili e frequenti gli errori durante la fase di post-compilazione, per cui applicazioni formalmente corrette generano errori, cosa che l'uso della tecnologia software OOP dovrebbe invece prevenire.

2.1.2 Efficienza in Fase di Esecuzione

Sono due gli elementi che determinano la velocità di esecuzione di una applicazione distribuita: il tempo di elaborazione intra-nodo ed inter-nodo.

Elaborazione Intra-Nodo: Protocollo di Serializzazione

La maggior parte del tempo speso durante una chiamata ad un metodo remoto è nella fase di serializzazione. Cioè la fase in cui i dati sono codificati per essere trasferiti sulla rete.

Studi come quelli di [WC00] e [PH99] mostrano che questo tempo può essere maggiore di ben cinque volte il tempo effettivo di trasmissione. Considerando che la codifica è solo una questione software di implementazione di un protocollo,

mentre la comunicazione interessa aspetti fisici quali velocità di comunicazione ed errori di trasmissione, ci si può rendere conto della sproporzione di queste cifre.

Elaborazione Inter-Nodo: Protocollo di Comunicazione

Il tempo speso nella comunicazione tra due nodi di un cluster non deve essere rallentato dal protocollo usato. La scelta del protocollo di comunicazione deve rispecchiare le necessità dell'applicazione con cui sarà usato. Una rete locale non crea particolari problemi di affidabilità, e neppure particolari problemi nel determinare il tasso di invio dei pacchetti. L'unico vero problema che crea è di sfruttare al massimo l'hardware che si ha a disposizione.

2.2 Stato dell'Arte

Il panorama dei sistemi con architettura ad invocazione di metodi remoti è ampio e vario. Di seguito diamo una descrizione di quelli che, a nostro giudizio, sono i più interessanti.

2.2.1 RMI: Remote Method Invocation

RMI è il sistema sviluppato da Ann Wollrath per la SUN nel 1996 [SUN97b]. Rappresenta la soluzione standard della SUN per Java, e riutilizza per quanto

possibile tecnologie già presenti nelle API di Java. La nostra analisi partirà da questo punto per poi passare effettivamente ad RMI.

Serializzazione

La serializzazione venne sviluppato come mezzo per memorizzare, per poi ripristinare, lo stato di una istanza di una classe su supporto di memorizzazione [SUN98].

Tra la memorizzazione dello stato dell'oggetto e il suo successivo recupero può variare la definizione della classe. Ma il processo prevede il suo recupero, anche solo parziale. Questo risultato è stato raggiunto aggiungendo, alla codifica dello stato dell'oggetto, tutta una serie di informazioni sulla definizione della classe a cui queste fanno riferimento.

Come detto sopra, la serializzazione fu progettata per memorizzare lo stato di una classe su memoria di massa. Questo scenario rende lecito supporre che tra la memorizzazione ed il successivo recupero, trascorra un intervallo di tempo piú o meno lungo: in cui la definizione della classe può variare. Può accedere quindi che i dati sopravvivano all'applicazione che li ha generati.

Riutilizzare la serializzazione per la trasmissione dei dati su rete geografica ha la sua motivazione: non è detto che l'applicazione client e server usino versioni allineati delle classi. Può accadere cioè che una delle due usi versioni piú recenti di una classe oggetto della trasmissione. Quindi, per riprendere il discorso di sopra,

i dati "vivono" meno dell'applicazione che li ha generati, ma sarà quest'ultima ad avere problemi nel trattarli.

Utilizzare la serializzazione per la trasmissione dei dati su rete locale ha, in generale, meno senso. Su una rete locale è facile che le classi siano memorizzate su un File System distribuito e quindi, è verosimile che entrambe le applicazioni siano allineate alla stessa versione delle classi.

Infine utilizzare la serializzazione per la trasmissione dei dati su un cluster di computer ha ancora meno senso rispetto allo scenario precedente. Infatti, oltre alle considerazioni sulle reti locali, in questo contesto è facile che entrambe le applicazioni siano scritte dalla medesima persona e quindi non vi siano problemi di codice ereditato.

Introspezione

La serializzazione è un processo lento in quanto usa l'introspezione delle classi ([SUN97a]). L'introspezione è un processo di analisi a run-time di una istanza di una classe, per determinare il tipo effettivo dei suoi campi. È implementata con una libreria chiamata di riflessione. Ed è un processo lento, non adatto in un contesto dove le prestazioni siano elemento essenziale.

RMI

RMI è stato sviluppato in primo luogo per operare in ambienti eterogenei, ed, in secondo luogo, per le prestazioni. Lo scenario tipico di utilizzo di RMI vede le applicazioni client e server distribuite su reti geografiche. Ad esempio un applet che comunica con l'applicazione server, per richiedere un servizio.
RMI utilizza il processo di serializzazione di Java per la trasmissione dei dati, e TCP/IP come protocollo di comunicazione. In generale tali scelte sono poco efficienti nel contesto locale. Nonostante ciò, RMI è facile da usare, e quindi ampiamente diffuso.

2.2.2 Jaguar

Il progetto Jaguar è orientato alla comunicazione locale. Risolve il problema della velocità di trasmissione utilizzando il protocollo VIA ([VIA97]) e per la codifica dei dati usando un processo di pre-serializzazione dei dati. Entrambe necessitano di una modifica alla Java Virtual Machine (JVM).

JaguarVIA: implementazione del protocollo VIA.

VIA è un protocollo di comunicazione per reti ad alte pretazioni ([VIA97]). Tale risultato è stato ottenuto mettendo a disposizione un accesso diretto in modalità utente alla memoria della scheda di rete. L'accesso alla memoria della scheda

permette di evitare la copia dal buffer dell'applicazione al buffer della scheda. L'uso della modalità utente permette di risparmiare il tempo per commutare la modalità di esecuzione della CPU.

Le implementazioni tradizionali di questo protocollo in Java risentono dell'uso dei lenti metodi nativi per accedere a locazioni di memoria fuori dallo Heap. Gli autori in [WC00] hanno modificato la JVM inserendo due nuovi opcodes: PEEK, POKE per l'accesso veloce e trasparente alla memoria fuori dallo Heap.

PSO: Pre-Serialized Objects

L'idea degli oggetti pre-serializzati è quella di mantenere in memoria una versione già serializzata dell'oggetto da trasmettere, in modo da averla sempre disponibile per la trasmissione. Anche questa soluzione richiede la modifica della JVM, in particolare la modifica di due opcodes: `getfield`, `putfield`.

L'istanza di classe viene memorizzata in quello che si definisce *contenitore*, e tutti gli accessi all'istanza sono filtrati dalle versioni modificate degli operatori di accesso. Ogni volta che un dato viene memorizzato nel contenitore, questo viene serializzato. Ogni volta che un dato viene recuperato, viene anche de-serializzato. L'istanza è sempre pronta per la trasmissione. Al momento giusto basterà una semplice copia: dal contenitore al buffer della scheda.

In pratica per accelerare un singolo tipo di operazione (la trasmissione dei

dati), si rallentano tutte le altre. Inoltre alcune operazioni che sono "veloci" in Java (assegnamento di un riferimento ad un campo di una classe) non lo sono piú in Jaguar (l'assegnamento fa scattare la serializzazione di tutto l'oggetto, con la copia nel contenitore)

2.2.3 Manta

In primo luogo diciamo che Manta [MNV+00] è tutt'ora allo stato embrionale. A differenza dei precedenti sistema, non c'è nulla di utilizzabile. Il sistema può essere visto come un ulteriore passo in avanti rispetto a Jaguar. Invece che modificare la JVM per aggiungervi nuove funzionalità, quelle stesse vengono aggiunte all'applicazione mediante la compilazione del sorgente in codice nativo. La compilazione permette tutta una serie di analisi del codice altrimenti impossibili, come ad esempio la determinazione di quali classi saranno serializzate. E per queste classi si procede alla generazione di veloci routines di serializzazione, che evitano l'introspezione della classe a run-time. La compilazione, inoltre, permette l'uso diretto di hardware del sistema (attraverso il protocollo VIA ad esempio), senza l'intralcio dei metodi nativi di Java.

Guardando in maniera critica il sistema si può dire che sono stati raggiunti ottimi risultati ma al costo di perdere alcune caratteristiche di Java, quali la portabilità ad esempio. Inoltre la documentazione in [MNV+00] cita ottimi risul-

tati in fase di comunicazione, ma presenta lacune nel documentare i risultati nella serializzazione di oggetti complessi quali i grafi.

2.2.4 UKA Transport

UKA Transport è stato sviluppato dall'Università di Karlsrhue, e rientra nel progetto JavaParty dell'organizzazione JavaGrande ([Jav98]). Come descritto in [PH99] il progetto si finalizza di ottimizzare una invocazione di un metodo remoto ottimizzando le fasi di cui è composta. Quindi ottimizzare la fase di serializzazione / deserializzazione con un protocollo leggero, comunicazione veloce con un protocollo di trasporto orientato ai pacchetti, pre-fork dei thread, ed object pooling.

È il progetto che piú si avvicina al nostro, per la scelta delle ottimizzazioni nella fase di invocazione remota, ma da cui ci differenziamo per il nostro approccio orientato al cluster, il che comporta realizzare una architettura che comprende un pool di servizi su rete locale, un registry con visibilità sull'intero cluster, la generazione di classi a run-time, e la generazione di serializzatori.

Capitolo 3

Architettura

In questo capitolo presenteremo l'architettura del sistema. L'analisi verterà sia sulla visione d'insieme del sistema che sui meccanismi interni. La implementazione e una analisi approfondita delle proprietà di ciascun sottosistema verrà descritta nei successivi capitoli. Mostreremo invece: (1) come sono assegnati i ruoli ai diversi enti che operano sul cluster, (2) come viene distribuito il carico di lavoro, ed infine (3) definiremo i protocolli usati.

3.1 Introduzione

Il framework da noi utilizzato è un sistema ad invocazione di metodi remoti. Ogni metodo rappresenta l'implementazione di un servizio, la classe a cui il metodo appartiene definisce un pool di servizi. In figura 3.1 mostriamo le interazioni del sis-

tema. In essa possiamo notare degli enti classici in un sistema di servizi remoti: il registry dei servizi, l'applicazione server, l'applicazione client (che per brevità chiameremo semplicemente server e client), il repository delle classi. Analizzeremo, di seguito, i compiti di ciascuno di essi.

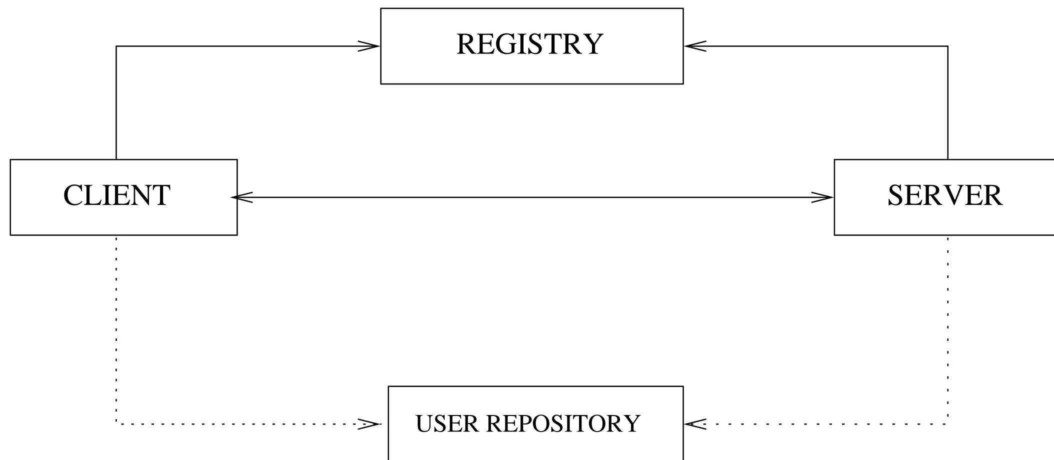

Figura 3.1: Architettura del sistema: una visione dall'esterno.

3.1.1 Cluster Registry

La nostra architettura prevede l'utilizzo di un registry. Il registry come suggerisce il nome è un registro che memorizza l'associazione tra il nome del servizio e l'indirizzo dello host che lo ospita.

Il registry in se stesso non è necessario. Infatti basta codificare nell'applicazione client l'indirizzo dell'host del server. Ad esempio, questo può essere fatto in applicazioni orientate alla elaborazione numerica, in cui la distribuzione delle

applicazioni sul cluster è fissa. Ma in generale il registry risulta essere comodo per permettere di automatizzare il processo di ricerca dei servizi.

Il registry può non essere unico nel cluster. Si può creare ridondanza di registri, con eventuali sovrapposizioni dei dati memorizzati. Uno scopo può essere distribuire meglio il carico di lavoro, oppure rendere il sistema piú affidabile.

3.1.2 Applicazione Server

L'applicazione server fornisce un pool di servizi al cluster. Ogni pool è individuato da un nome. All'interno di questo pool, ogni singolo servizio ha un proprio nome.

Una applicazione può ospitare piú pool di servizi. Ogni pool è implementato da una classe. Il singolo servizio è implementato mediante un metodo.

Il server rende noto al cluster la presenza del pool di servizi registrandolo sul registry. Da quel momento rimane in attesa delle richieste dai client.

3.1.3 Applicazione Client

L'applicazione client è il fruitore del servizio messo a disposizione dal server. Il client interroga il registry per conoscere dove è ospitato il servizio. Da quel momento inizia la comunicazione punto-punto tra il client ed il server.

3.1.4 User Repository

Lo "User Repository" come suggerisce il nome, è un deposito di classi dell'utente. Le classi generate automaticamente dal sistema, in risposta ad una richiesta dell'utente, vengono memorizzate in questo deposito. Esso è condiviso via NFS tra client e server.

3.2 Analisi Interna

Il sistema visto internamente è composto dai seguenti moduli:

RUN: Il modulo che si occupa: (1) della comunicazione tra i nodi del cluster, (2) della gestione delle invocazioni remote dal lato client e dal lato server.

CLIO: Il modulo che raggruppa le funzioni di (1) class loader, e (2) generatore / modificatore di classi.

RIO: Il modulo che agisce da registro dei servizi.

Analizzeremo adesso singolarmente le caratteristiche di ciascuno di questi moduli aiutandoci con la figura 3.2. Ma per una esposizione completa rimandiamo al capitolo 4 per il modulo RUN, al capitolo 5 per il modulo RIO, ed al capitolo 6 per il modulo CLIO.

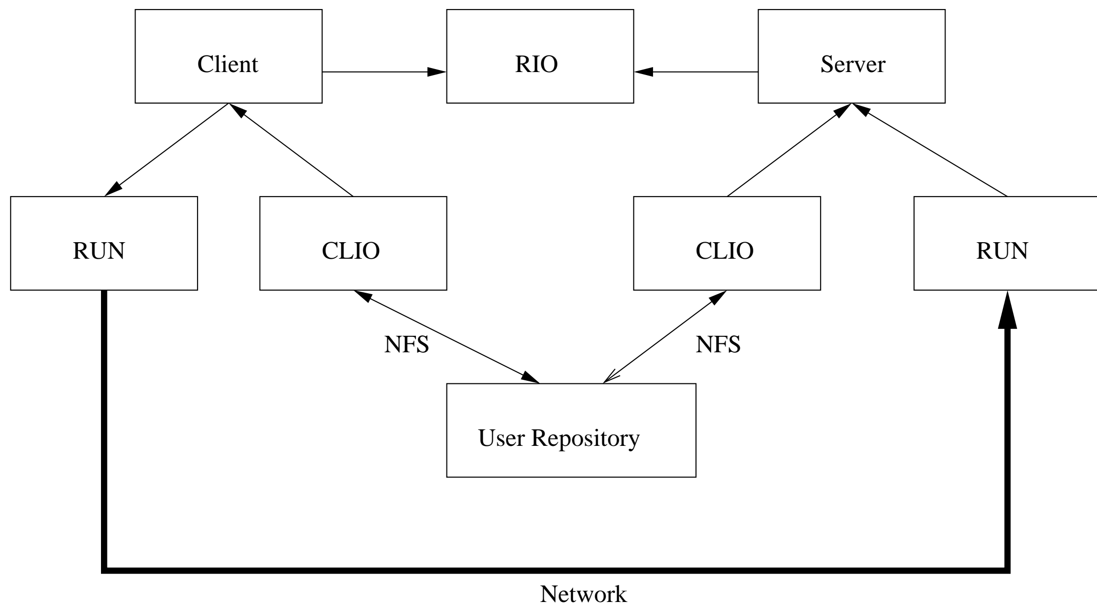

Figura 3.2: Architettura del sistema: una visione dell'interno.

3.2.1 RUN

Il client ed il server si scambiano informazioni. Un client invia la richiesta di un servizio, con i parametri della richiesta. Il server risponde con il risultato dell'elaborazione. Chiameremo queste informazioni ad alto livello con il termine *messaggi*. I messaggi per essere trasmessi da un nodo all'altro devono essere divisi in "pacchetti". Un pacchetto è la minima informazione trasmettibile sulla rete.

Dal punto di vista della codifica e trasmissione dei dati, i servizi offerti da RUN sono:

Invio di una richiesta: Codifica del servizio richiesto e dei parametri della richiesta.

Serializzazione: Codifica dei dati per la trasmissione. Si codificano il tipo effettivo dei singoli oggetti con degli identificatori. Ed i riferimenti ad altri oggetti vengono sostituiti con numeri progressivi.

Trasmissione dei dati: Il messaggio viene diviso in pacchetti, spedito sulla rete, e ricostruito sull'altro host. Eventuali errori di trasmissione vengono corretti, con la ritrasmissione.

Il modulo inoltre si occupa delle generazione e del controllo dei Thread[1] per l'esecuzione delle richieste.

3.2.2 RIO

RIO è diviso per le sue funzionalità in due parti: una parte locale non condivisa, ed una parte globale condivisa.

Parte Locale

Le funzionalità della parte locale operano strettamente con il modulo RUN: forniscono informazioni sui servizi attivi all'interno dell'applicazione server.

[1] Il modulo RUN, per motivi di efficienza, è stato sviluppato come un sistema multi-thread ed a eventi.

Parte Globale

Le funzionalità della parte globale sono quelle tipiche di un registry. Sono state implementate come qualunque altro servizio creato dall'utente, solo che è fornito di base.

3.2.3 CLIO

Sono tre le funzioni raggruppate in CLIO: (1) caricatore di classi, (2) generatore di classi, e (3) modificatore di classi. Analizziamole una ad una:

Caricatore di Classi: ClusterClassLoader

Nella nostra architettura, le classi vengono caricate secondo lo schema usuale di Java: dalle directory specificate nel CLASSPATH. A queste viene aggiunto il repository dell'utente. Se una classe non viene trovata secondo questo schema di ricerca, ed è una delle classi necessarie per la comunicazione, allora viene generata. Se una classe viene trovata, ma necessita di essere modificata, allora viene chiamato il sottomodulo adatto.

Generatore di Classi: StubGenerator, SkeletonGenerator

Lo stub e lo skeleton, sono le due classi che piú a diretto contatto lavorano con i servizi del server, e l'applicazione client.

La classe stub interagisce con l'applicazione client. Gli fornisce i servizi offerti dal server ma solo nominalmente, in realtà instrada le richieste all'effettivo server mediante il modulo RUN. Lo skeleton è la classe che interagisce con l'applicazione server. Riceve le richieste da remoto, le decodifica, invoca il servizio richiesto, e restituisce il risultato.

Queste due classi sono specifiche dei servizi implementati sul server. E non possono essere codificate prima della definizione del servizio. In [SUN97b] tale vincolo è stato soddisfatto mediante l'uso di un post-compilatore. Abbiamo scelto invece di soddisfare questo vincolo generando a run-time le classi necessarie all'infrastruttura di comunicazione, per poi memorizzarle nel repository. Il tempo speso è lo stesso. Ma si ha il vantaggio di non allungare il ciclo di vita del software.

Modificatore di Classi

Se una istanza di classe viene trasmessa da un nodo all'altro, deve prima essere serializzata. La serializzazione standard è un processo lento. In quanto necessita di una fase di introspezione dell'istanza, per determinarne i tipi effettivi dei campi. Inoltre il tempo speso nell'introspezione non viene riutilizzato. Se la stessa istanza di classe deve di nuovo essere serializzata, subirà per intero l'introspezione.

Ma non tutte le classi devono subire questo processo di analisi / trasmissione.

Il programmatore indica che istanze di una classe possono essere soggette di una trasmissione implementando l'interfaccia `Transportable`. CLIO all'atto del caricamento della classe vi aggiunge tre metodi:

`readObject()` lettura dell'istanza della classe da un canale.

`writeObject()` scrittura dell'istanza della classe su un canale.

`sizeOf()` dimensione dell'istanza della classe.

un costruttore pubblico di default:

`init()` inizializza l'oggetto chiamando il costruttore della classe padre, la vera inizializzazione verrà effettuata da `readObject()`

e un campo:

`SUID` un intero a 64-bit, che è l'identificativo unico della classe. Due classi possono avere lo stesso FQN[2] ma essere diverse. Oppure possono individuare effettivamente la stessa classe, ma in due versioni distinte.

Questo intero è calcolato in maniera tale, da identificare in maniera univoca ogni signola definizione di classe.

I tre metodi servono, per cosí dire, per *riciclare* il tempo speso nell'introspezione. Una classe subisce l'introspezione solo al primo caricamento. Il processo sarà ripetuto solo ad una sua successiva modifica.

[2]FQN (Full Qualified Name): un nome di classe preceduto dal nome completo del package, e.g.: java.util.Array

3.3 Uno Scenario d'Uso

Analizzeremo in questa sezione un tipico scenario d'uso del sistema.

Passo 1: Il server registra localmente il servizio: Nel registry locale dell'applicazione, viene memorizzata l'associazione tra il nome del pool di servizi e l'istanza della classe che lo implementa. La registrazione provoca la generazione delle classi stub / skeleton, che vengono memorizzate nel repository dell'utente.

Passo 2: Il server invia la richiesta di registrazione al cluster registry: L'applicazione server, conosce l'ubicazione del servizio di cluster registry dai suoi files di configurazione. Gli invia una richiesta contenente: (1) il nome del pool di servizi, (2) le firme dei metodi che implementano ciascuno servizio. La firma comprende nome del metodo, tipo dei parametri, e tipo del risultato.

Passo 3: Il cluster registry analizza la richiesta: Il cluster registry verifica se non sia già stato associato un pool con quel nome. In caso negativo registra la coppia (nome, descrizione) del pool di servizi.

Passo 4: Il client interroga il cluster registry: L'applicazione client conosce l'ubicazione del cluster registry tramite i files di configurazione. Invia una richiesta contenente il nome del pool di servizi richiesti.

Passo 5: Il cluster registry risponde: Il cluster registry esamina le sue strutture dati, e risponde inviando: (1) le informazioni sull'indirizzo dell'host che ospita il servizio, (2) la lista dei metodi con relativa firma che implementano i servizi. Con questo passo cessa l'attività del cluster registry.

Passo 6: Il client riceve la risposta dal registry: Da questo momento in poi, vi sarà una comunicazione punto-punto tra il client ed il server.

Passo 7: Il Client invia la richiesta al server: L'applicazione Client effettua la chiamata remota. Per fare ciò utilizza lo stub preparato in precedenza dall'applicazione Server. Lo recupera utilizzando il repository comune. Le classi che implementano i parametri della chiamata remota vengono modificati, con l'aggiunta di veloci routine di serializzazione / deserializzazione.

Passo 8: Il Server riceve la richiesta: Decodifica i parametri utilizzando i deserializzatori preparati dal Client. Esegue il metodo del pool di servizi.

Passo 9: Il Server invia la risposta: La classe che implementa il valore di ritorno viene modificata dal server con l'aggiunta di veloci routine di serializzazione / deserializzazione. Il risultato viene codificato ed inviato.

Passo 10: Il Client riceve la risposta: Quando riceve la risposta, l'applicazione Client utilizza la routine di deserializzazione codificata dal server. E può poi accedere ai dati.

Come si può notare il carico di lavoro è distribuito tra le due controparti della comunicazione: Client e Server. Si noti che sia le classi generate, che quelle modificate sono memorizzate nel repository. Ad una successiva richiesta saranno riutilizzate, senza bisogna di ulteriori elaborazioni.

3.4 Protocolli di Comunicazione

In questa sezione analizzeremo i tre diversi protocolli utilizzati dal sistema. Come accennato sopra, essi possono essere classificati in tre diversi livelli:

- A basso livello: invio / ricezione dei pacchetti.
- A livello intermedio: serializzazione / deserializzazione dei dati.
- Ad alto livello: invio della richiesta e ricezione della risposta.

Tutti e tre sono implementati nel modulo RUN.

3.4.1 Basso Livello: Protocollo GO-BACK-N

Una sessione offre un flusso affidabile di trasmissione. Riusciamo ad ottenerlo mediante un flusso inaffidabile di trasmissione di pacchetti, grazie all'uso di un protocollo di ritrasmissione dei pacchetti persi.

A basso livello, l'unità elementare di trasmissione è il pacchetto. I pacchetti possono essere divisi in due categorie: pacchetti di controllo e di dati. I pac-

chetti di controllo servono per inizializzare la sessione, chiuderla, e richiedere la ritrasmissione di pacchetti non ricevuti. I pacchetti di dati sono numerati, per individuare il segmento di informazioni che trasportano. Entrambi i tipi contengono informazioni sul nodo mittente e sul nodo destinazione.

A basso livello abbiamo deciso di implementare il protocollo di trasmissione GO-BACK-N. Il protocollo è descritto ampiamente in [BG92]. Qui descriviamo solo le modificano che vi abbiamo apportato.

Il protocollo è basato sul concetto di finestra scorrevole di pacchetti (Sliding Window). Una finestra di pacchetti è una successione di pacchetti. Il nodo mittente, ad intervalli di tempo, invia una finestra di pacchetti. Man mano che il nodo destinazione li riceve invia degli acknownledge al mittente. Questi autorizzano a traslare la finestra, cioè ad inviare altri pacchetti. Se il nodo mittente non riceve l'acknownledge, inizia a ritrasmettere la finestra di pacchetti.

Abbiamo deciso di implementare questo protocollo in quanto, con una opportuna scelta della dimensione della finestra dati, offre adeguate prestazioni su reti veloci, come mostrato in [PH99].

Il tipo di pacchetti implementati, con il loro significato è mostrato in tabella 3.1

A differenza del protocollo standard, qui sono stati aggiunti due nuovi tipi di pacchetti: `RET` , `ACKR` . Questi hanno lo stesso significato delle controparti

Tipo pacchetto	Descrizione	Parametri
INIT	Inizializzazione di una sessione per l'invio di una richiesta	Numero di sessione, dimensione del segmento dati
ACKI	Accettazione dell'inizializzazione	Numero di sessione
DISC	Disconnesse della sessione	Numero di sessione
ACKD	Accettazione della disconnessione	Numero di sessione
DATA	Pacchetto di dati	Numero di sessione, Numero di pacchetto, Dati
RN	Richiesta di un pacchetto	Numero di sessione, Numero di pacchetto
RET	Inizializzazione di una sessione per l'invio di una risposta	Numero di sessione, dimensione del segmento dati, Numero di sessione della richiesta
ACKR	Accettazione della inizializzazione	Numero di sessione

Tabella 3.1: Elenco dei pacchetti del protocollo GO-BACK-N.

`INIT`, `ACKI` solo che trattano dell'invio dei dati di una risposta, invece che della richiesta.

Come si puó notare dalla tabella, abbiamo arricchito l'informazione contenuta in ogni pacchetto. Tutti i pacchetti, ad esempio, contengono il numero di sessione a cui appartengono. I pacchetti di `INIT/RET`, contengono informazioni sul servizio richiesto.

3.4.2 Livello Intermedio: Protocollo di Serializzazione

I parametri ed il risultato di un servizio devono essere codificati per poter essere trasmessi da un nodo ad un altro. Quando vengono inviati subiscono il Marshalling[3],cioé serializzati nell'ordine in cui appaiono nella firma del metodo. Alla ricezione subiscono l'Unmarshalling[4], cioé una deserializzazione nello stesso ordine con il quale sono stati serializzati. Si noti che quando viene inviata la risposta, che contiene un solo parametro, queste fasi elaborano un solo oggetto.

Il protocollo di serializzazione che abbiamo implementato, è una versione "leggera" di quello implementato in [SUN98]. Abbiamo già discusso in 2.2.1 delle motivazione che ci hanno spinto a questa scelta:

Qui le riassumiamo:

Velocità Un protocollo complesso è lento da gestire.

Località L'allineamento tra le versioni delle classi del client e del server, ci permette di ridurre la descrizione dei dati trasmessi.

[3]Incolonnati
[4]I dati *rompono le righe*

3.4.3 Alto Livello: Protocollo di Richiesta dei Servizi

Ad alto livello l'unità di elaborazione è quello che abbiamo chiamato *Callgram*[5]. Un Callgram serve sia ad inviare la richiesta di un servizio, con gli eventuali parametri, sia il risultato del servizio.

Quando usato per trasmettere la richiesta di un servizio, esso contiene:

- La descrizione del servizio richiesto.
- L'indirizzo del client
- I dati degli argomenti della richiesta.

Quando usato per trasmettere il risultato dell'elaborazione, esso contiene:

- L'indirizzo del server
- I dati del risultato dell'elaborazione

3.5 Ottimizzazioni Applicate

Durante la nostra esposizione cercheremo di mostrare dove e quando sono state applicate delle ottimizzazioni. In linea generale possiamo dire che abbiamo cercato, per quanto possibile di diminuire l'allocazione di oggetti e di riciclare il loro uso. Questo obiettivo è perseguito con tecniche quali l'object pooling.

[5]Callgram deriva da Datagram, che a suo volta deriva da Telegram. Al di là dei neologismi, i termini indicano i dati trasmessi

3.5.1 Object Pooling

L'object pooling è una tecnica per aumentare le prestazioni di un sistema. Cerca di ottimizzare i due elementi essenziali di un programma: spazio di allocazione, e tempo di esecuzione. Si basa su: (1) pre-allocare una serie di oggetti e conservare i riferimenti in un pool, (2) prelevare le istanze da questo pool, (3) restituire le istanze al pool una volta che non sono piú necessarie. Già è stata dimostrata l'utilità di questa tecnica in svariate occasioni. Particolarmente quando si tratta di gestire oggetti di notevoli dimensioni, oppure con una lunga fase di inizializzazione come ad esempio i threads.

Thread Pooling

I thread sono oggetti attivi, con una lunga fase di inizializzazione. Abbiamo applicato cura nel fare in modo che ogni classe che deve dividere il suo flusso di esecuzione in due thread, prelevi l'altro da un pool di thread pre-forkati. E lo restituisca al pool una volta terminato il compito.

Abbiamo astratto questa logica nella classe astratta `ThreadPool`, da cui abbiamo derivato specializzazione per gli usi particoloari.

Capitolo 4

RUN

Il modulo RUN fornisce le funzionalità di: (1) riferimento remoto di un pool di servizi, (2) invocazione remota di un servizio, (3) codifica dei dati, (4) trasmissione con diversi tipi di protocollo di trasporti, (5) creazione e gestione dei thread per l'esecuzione in parallelo di servizi.

4.1 Architettura

RUN costituisce da solo tre quarti di tutto il sistema. Utilizza tre diversi protocolli:

- A basso livello, utilizza un protocollo di comunicazione basato sul GO-BACK-N per la trasmissione dei pacchetti.

- A livello intermedio, utilizza un protocollo di serializzazione per la trasmissione di istanze di classi.
- Ad alto livello, utilizza un protocollo basato sui messaggi per codificare le richieste di servizi.

L'architettura del modulo, mostrato in figura 4.1, è a livelli (layer), ognuno indipendente dall'altro.

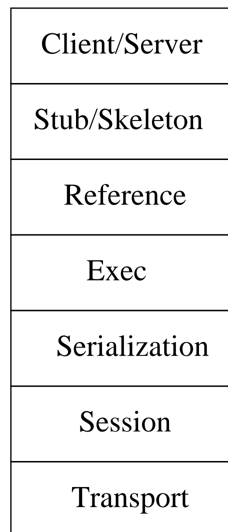

Figura 4.1: Divisione in livelli del modulo RUN.

Iniziamo, nel descriveremo l'architettura del modulo, dal livello inferiore per poi proseguire verso l'alto

4.2 Livello 1: Trasporto

Il livello Transport interagisce con le librerie di comunicazione su rete. Tra queste vi sono le API del package `java.io`, per l'utilizzo della suite di protocolli TCP/IP. Altri esempi possono essere le librerie GM e VIA per le reti Myrinet.

Un generico trasporto è rappresentato dalla classe astratta `Transport`. Sono due gli elementi di interesse di questa classe: (1) il metodo astratto send() che gestisce l'invio di un pacchetto sul trasporto, e (2) il metodo astratto `run()` necessario per l'estensione dalla classe `Thread`. La prima serve a gestire l'invio di pacchetti, la seconda ne gestisce la ricezione. Ricordiamo che mentre l'invio è un evento sincrono che puó quindi essere gestito da una chiamata di un metodo, la ricezione è un evento asincrono che non possiamo prevedere quando avverrà e viene gestito da una routine in continua esecuzione.

Specializzazioni di questa classe, permetteranno di utilizzare un protocollo specifico. Ad esempio abbiamo implementato l'uso del protocollo UDP mediante il trasporto: `UDPTransport`.

In figura 4.2 mostriamo l'interazione di `UDPTransport` con il Sistema Operativo. La parte evidenziata è un codice nativo e mostra la comune astrazione del protocollo TCP/IP, mentre il resto è in bytecode ed appartiene al modulo RUN. Tutti i trasporti istanziati sono memorizzati in una `TransportTable`.

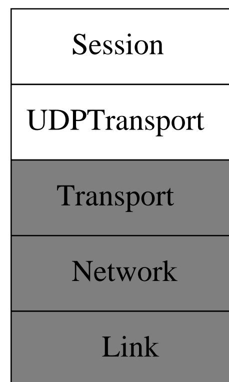

Figura 4.2: Divisione in livelli della logica di comunicazione su rete

4.2.1 Pacchetti

L'elemento che accomuna le implementazioni concrete di `Transport` è una comunicazione orientata ai pacchetti. Un pacchetto è rappresentato da una istanza della classe `Packet`. Esso contiene informazioni: (1) sul nodo mittente, (2) la sessione a cui appartiene, (3) il tipo di pacchetto, (4) eventuale segmento dati. L'istanza di `UDPTransport` si occupa di costruire una istanza di `Packet` dalle informazioni ricevute da un datagramma UDP, e di fare il viceversa nel caso di un invio.

In tabella 4.1 mostriamo un datagramma UDP, e come i campi di una istanza di `Packet` sono incapsulati nella sua sezione dati.

In tabella 4.2 omettiamo gli header, e mostriamo solo la sezione dati dei datagrammi. In particolare mostriamo la codifica di tutti i tipi di pacchetti riconosciuti dal protocollo.

Header IP	Header UDP	Opcode	Id. Sessione	......

Tabella 4.1: Incapsulazione di un Pacchetto in un datagramma UDP.

Multiplexing / Demultiplexing

Vi è una sola istanza di `UDPTransport` per indirizzo di rete per applicazione. Questo rende necessario un multiplexing dei pacchetti all'atto dell'invio. Come mostrato in figura 4.3, le sessioni inviano i pacchetti usando la stessa istanza di `UDPTransport`. Il multiplexing è realizzato sincronizzando gli accessi la trasporto. Il reciproco demultiplexing, mostrato in figura 4.4, è effettuato da `SessionTable` mediante l'identificativo di sessione.

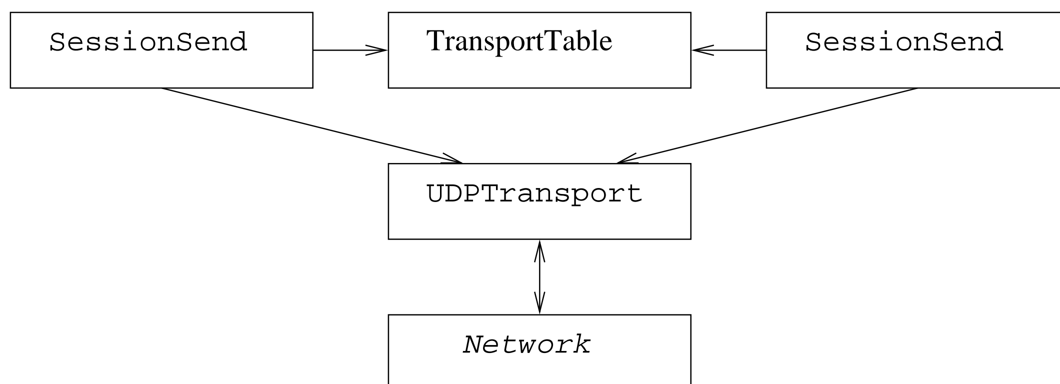

Figura 4.3: Multiplexing dei pacchetti.

Pacchetto: INIT

INIT	id. Sessione	pacchetti totali	dim. Callgram	nome pool	nome servizio	firma servizio

Pacchetto: RET

RET	id. Sessione	pacchetti totali	dim. Callgram	valori di ritorno	riferimento Sessione

Pacchetto: RN

RN	id. Sessione	pacchetti totali	SN

Pacchetto: DATA

DATA	id. Sessione	pacchetti totali	SN	dimensione dati	dati

Pacchetto: DISC

DISC	id. Sessione

Pacchetto: ACKI

ACKI	id. Sessione

Pacchetto: ACKR

ACKR	id. Sessione

Pacchetto: ACKD

ACKD	id. Sessione

Tabella 4.2: Formato dei pacchetti usati da `UDPTransport`.

4.2.2 Istanze di `Transport` Come Oggetti Attivi

Le istanze della classe `Transport`, in linea di principio sono oggetti *attivi*. Hanno un loro flusso di controllo, in quanto gestiscono eventi asincroni, cioé la ricezione di pacchetti.

L'implementazione concreta di `Transport` da noi fornita: `UDPTransport`, ha addirittura due thread: (1) per la gestione dei pacchetti in arrivo usa `FIFOInThread`, (2) per la gestione dei pacchetti in partenza usa `FIFOOutThread`. Come suggeriscono i nomi la politica dei gestione dei pacchetti è FIFO.

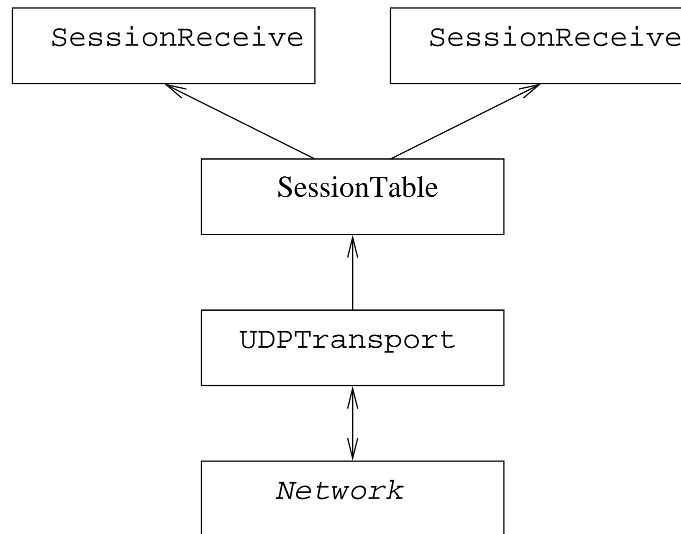

Figura 4.4: Demultiplexing dei pacchetti.

4.2.3 `NetAddress`: Astrazione di un Indirizzo di Rete

Il livello Transport è in grado di gestire diversi tipi di protocolli di trasporto mediante la classe astratta `Transport`. L'astrazione di un indirizzo di rete è fornito dalla classe `NetAddress`.

Sue estensioni concrete sono: `IPAddress` per gli indirizzi IP, e `VIAAddress` per gli indirizzi nel protocollo VIA. Quest'ultima è stata implementata solo per scopi dimostrativi.

4.2.4 Ottimizzazioni Applicate

L'istanziazione di un oggetto `UDPDatagram` è un procedimento molto lungo, quindi `UDPTransport` funziona con due sole istanze: una per i datagrammi in uscita ed

una per quelli in ingresso. Il mapping dei pacchetti di RUN a datagrammi UDP viene effettuato gestitendo due code di pacchetti: una in ingresso ed una di uscita.

4.3 Livello 2: Sessione

Nel livello sessione è stata implementata la logica del protocollo GO-BACK-N. Le sessioni si differenziano in due categorie, in base a se implementano la logica di trasmisione o la logica di ricezione di un Callgram. Il livello Session interagisce con il livello Transport mediante la classe astratta `Transport`.

4.3.1 Session, SessionSend, SessionReceive

Una sessione è rappresentata dalla classe astratta `Session`, mostrata nel listato B.6 a pagina 129. Essa è caratterizzata da una serie di metodi astratti: `init(), acki(), data(), ...` che rappresentano la gestione degli omonimi pacchetti del protocollo di comunicazione.

Sono due le estensione concrete di `Session`: `SessionSend` che implementa la logica di trasmissione dei Callgram, e la classe `SessionReceive` che implementa la logica di ricezione. In pratica si implementano i due aspetti del protocollo (invio e ricezione) implementando i metodi astratti che gestiscono i pacchetti.

Ad esempio, il metodo astratto `init()` di `Session` rappresenta la gestione dell'omonimo pacchetto INIT. Il metodo concreto `init()` di `SessionSend` si oc-

cupa di gestire l'invio del pacchetto, cioé inizializzare una istanza di `Packet` con gli opportuni campi. Il metodo concreto `init()` di `SessionReceive` si occupa di gestire la ricezione del pacchetto, cioé inizializzare la sessione stessa e preparare la creazione di un Callgram.

4.3.2 Callgram

Un messaggio scambiato dai due nodi rappresenta la coppia: (controllo, dati). Il controllo contiene le informazioni che permettono di individuare il servizio richiesto. I dati rappresentano le informazioni per l'esecuzione del servizio, o il suo risultato. La coppia verrà indicata con il termine *Callgram*.

Le informazioni di controllo sono codificate nel pacchetto di inizializzazione (del tipo `INIT` e `RET`), gli argomenti della chiamata nei pacchetti dati (del tipo `DATA`).

Il messaggio *Callgram* scambiato tra due nodi, viene memorizzato all'interno di un oggetto della classe `Callgram`. In tabella 4.3 descriviamo i suoi campi, che sono stati divisi in base alla fonte dei dati che contengono, cioé se contengono informazioni locali al nodo, oppure informazioni trasmesse da un altro nodo.

Per la descrizione del campo `Listener` rimandiamo al par. 4.5.1 a pagina 45, per una descrizione del campo `Container` rimandiamo al par. 4.4 a pagina 41, per una descrizione del riferimento al servizio rimandiamo al par. 5.1 a pagina 62.

Locali	Session	Riferimento all'istanza della classe Session che gestisce la trasmissione
	Listener	Riferimento al Listener a cui verranno notificati gli eventi relativi a questo Callgram
Trasmessi	Operation	Operazione codificata nel callgram
	Container	Argomenti della chiamata remota
	service_reference	Servizio richiesto
	return_value	Eventuale errore di trasmissione

Tabella 4.3: Classificazione dei campi di una istanza di `Callgram`.

4.3.3 Protocollo di Comunicazione

L'invio della richiesta vede il client nel ruolo di chi invia dati (usa quindi `SessionSend`), ed il server nel ruolo di chi riceve (usa quindi `SessionReceive`). Il pacchetto `INIT` inizializza la sessione di comunicazione. È il primo pacchetto inviato dalla `SessionSend` sul client al nodo server. Come primo effetto, viene creato sul nodo server una istanza della classe `SessionReceive`. Il pacchetto `INIT` contiene informazioni (1) sul pool di servizi richiesto, e all'interno di questo (2) sul singolo servizio richiesto, (3) la dimensione del *Callgram* inviato, ed (4) il numero di sessione. Quest'ultimo, uguale sui due nodi, individua in maniera univoca la sessione.

Il server una volta ricevuto il pacchetto `INIT`, risponde con un pacchetto `ACKI`. Da quel momento `SessionSend` invia finestre di pacchetti `DATA`, secondo

il protocollo GO-BACK-N. Ognuno di essi reca informazioni: (1) sul numero di sessione a cui appartiene, (2) sul numero progressivo del segmento di dati che trasporta (numero di sequenza: Sequence Number), e (3) il segmento dati stesso.

`SessionReceive` periodicamente risponde con pacchetti `RN`, che richiedono uno specifico pacchetto. Questi pacchetti da un lato gli permettono di richiedere pacchetti non ricevuti, dall'altro permettono alla `SessionSend` di far traslare la finestra di pacchetti. Infatti se arriva un flusso continuo di pacchetti `RN` la finestra è traslata senza soluzione di continuità. Altrimenti si rallenta la trasmissione, con una serie di ritrasmissioni.

L'ultimo pacchetto trasmesso da `SessionSend` è `DISC`, che termina la comunicazione. A cui `SessionReceive` risponde con un pacchetto `ACKD`, che termina la comunicazione dall'altro lato. Nel caso gli arrivino altri pacchetti, di qualsiasi tipo, riferiti a quella sessione, risponderà con pacchetti `ACKD`.

L'invio della risposta fa invertire i ruoli della trasmissione. Ora è il server ad inviare dati (ed usa `SessionSend`), ed il client a riceverli (ed usa `SessionReceive`). L'invio del *Callgram* contenente la risposta è inizializzata con un pacchetto `RET`. Esso oltre ad indicare (1) la dimensione della parte dati, (2) il numero di pacchetti totali, (3) l'identificativo della sessione, contiene (4) l'identificativo della vecchia sessione. Questo dato serve al lato client per recapitare il *Callgram* al thread che aveva inoltrato la richiesta. L'invio del *Callgram* segue la stessa logica

di sopra, con l'invio di pacchetti: `DATA` e `RN`. E termina sempre con una coppia di pacchetti `DISC`, `ACKD`.

4.3.4 Ottimizzazioni Applicate

Gli oggetti `Packet` sono molto semplici, ma vengono allocati / deallocati dalle sessioni con molta frequenza. Abbiamo usato quindi una classe `PacketPool` per gestire un pool di pacchetti pre-allocatti. Le sessioni sono oggetti attivi, per ottimizzare il tempo di risposta da una richiesta remota preleviamo le sessioni da due pool: `SessionSendPool`, `SessionReceivePool`.

writeObject()	Scrittura di una istanza di un oggetto
writeByte() writeBoolean() writeChar() writeShort() writeInt() writeLong() writeFloat() writeDouble()	Scrittura di dati primitivi
writeString()	Scrittura di stringhe
writeArray()	Scrittura di array / matrici di primitivi / oggetti

Tabella 4.4: Primitive messe a disposizione da `ContainerOutputStream`.

4.4 Livello 3: Serializzazione

Nel paragrafo precedente si è fatto cenno nella tabella 4.3 ad un campo di tipo `Container`. La classe `Container` codifica la logica di serializzazione / deserializzazione da uno stream di dati attraverso due classi interne: `Container.ContainerOutputStream`, che implementa la logica di serializzazione, `Container.ContainerInputStream` che implementa la logica di deserializzazione. In tabella 4.4 mostriamo le primitive offerte dalla prima, mentre in tabella 4.5 quelle della seconda. I dati raw, cioé senza alcuna strutturazione, sono memorizzati in una istanza di classe `Frame`.

Per codificare / decodificare usiamo un protocollo di serializzazione / deserializzazione dei dati. Abbiamo formalizzato il protocollo mediante la definizione di una grammatica context-free ([ASU86]), in tabella 4.6 mostriamo le produzioni,

readObject()	Lettura di una istanza di un oggetto
readByte() readBoolean() readChar() readShort() readInt() readLong() readFloat() readDouble()	Lettura di dati primitivi
readString()	Lettura di stringhe
readArray()	Lettura di array / matrici di primitivi / oggetti

Tabella 4.5: Primitive messe a disposizione da `ContainerInputStream`

mentre nella tabella 4.7 descriviamo il significato dei simboli terminali.

Quello che é interessante notare, è la codifica delle classi: effettuate mediante il suo FQN[1], e il valore SUID. Già abbiamo discusso ampiamente delle motivazioni che ci hanno spinto a questa scelta nel par. 3.4.2. Per una discussione sul significato di `SUID` rimandiamo al par. 6.4 a pagina 103.

Quello che ci preme sottolineare, è che per serializzazione / deserializzare efficietemente un oggetto, questo deve contenere i metodi `readObject()`, `writeObject()`. Questi metodi sono aggiunti automaticamente ad una classe che implementa l'interfaccia `Transportable` mediante l'intervento del modulo CLIO. Discuteremo di questo procedimento nel capitolo 6 sul modulo CLIO.

[1]FQN (Full Qualified Name): un nome di classe preceduto dal nome completo del package, e.g.: java.util.Array

Stream	Dato Stream
	Dato
Dato	Primitivo
	Oggetto
	Array
	Stringa
Primitivo	(tipo)valore
Stringa	NuovaStringa
	RiferimentoStringa
NuovaStringa	INSTANCE (int)handle (utf8)stringa
RiferimentoStringa	REFERENCE (int)handle
Classe	NuovoTipo
	RiferimentoTipo
NuovoTipo	NEWCLASS (long)suid (utf8)class_name
RiferimentoTipo	REFCLASS (long)suid
Oggetto	NuovaIstanza
	RiferimentoIstanza
NuovaIstanza	INSTANCE (int)handle Classe Istanza
RiferimentoIstanza	REFERENCE (int)handle
Istanza	(byte[])istanza
Array	NuovoArray
	RiferimentoArray
NuovoArray	INSTANCE (int)handle Classe (int)length (array_type)instance
RiferimentoArray	REFERENCE (int)handle
Stringa	(utf8)stringa

Tabella 4.6: Protocollo di serializzazione: produzioni della grammatica.

NEWCLASS	Nuova definizione di classe
REFCLASS	Riferimento ad una classe già definita
INSTANCE	Nuova istanza di un oggetto
REFERENCE	Riferimento ad un oggetto già definito
(tipo)	Valore di un tipo primitivo di Java
(array_type)	Istanza di un tipo Array
(byte[])	Sequenza di byte
(int)	Intero
(long)	Intero lungo
(utf8)	Stringa codificata in formato utf8

Tabella 4.7: Protocollo di serializzazione: simboli terminali della grammatica

4.4.1 Ottimizzazioni Applicate

A questo livello, le ottimizzazioni sono orientate a diminuire il tempo di esecuzione della serializzazione / deserializzazione. Grazie all'aggiunta di serializzatori alle classi `Transportable` che evitano di fare l'introspezione, e all'uso di un protocollo leggero di serializzazione. che non rallenta la trasmissione dei dati.

4.5 Livello 4: Exec

Il package Exec si occupa della logica di gestione della invocazione remota. Esso è diviso in due parti. Dal lato client la gestione della invocazione è gestito dalla classe `Manager`, e dal lato server dalla classe `Dispatcher`. Entrambe utilizzano il pattern: *Listener*, implementato dall'interfaccia: `CallgramListener`. Che descriveremo di seguito.

4.5.1 Implementazione del Pattern Listener

La trasmissione di un *Callgram* genera eventi. Ad esempio al termine della trasmissione si genereranno due eventi: il mittente riceverà la notifica che è terminata la trasmisione, il ricevente che è arrivato un nuovo *Callgram*.

Una classe che prevede notifiche di eventi correlati ai *Callgram* implementa l'interfaccia `CallgramListener`. La mostriamo nel listato 4.1. Essa prevede l'implementazione di un solo metodo `execCall()` che ha come argomento il `Callgram` che ha generato l'evento.

```
package run.session;

public interface CallgramListener
{
  public void execCall( Callgram c );
}
```

Listato 4.1: CallgramListener

4.5.2 Lato Client: Classe `Manager`

La classe `Manager` gestisce la logica della chiamata remota dal lato client. Essa rende disponibile un solo metodo: `execRemoteCall()`, definito come statico per permettere una maggiore facilità di utilizzo. Si osservi che: nel caso che piú classi contemporaneamente effettuassero una invocazione remota si potrebbe generare una race-condition. Per evitare questa spiacevole evenienza, la gestione della chiamata è delegata ad un oggetto della classe `Pusher`, In modo che ogni istanza gestisce una sola invocazione.La classe `Pusher` offre due metodi: (1) `execRemoteCall()` per gestire la logica di chiamata remota, e (2) `execCall()` per le notifica degli eventi relativi al *Callgram* inviato.

4.5.3 Lato Server: Classe `Dispatcher`

La classe `Dispatcher` gestisce la logica della chiamata remota dal lato server. Esso fornisce un solo metodo: `getDispatcher()` che restituisce un thread `CallThread`, dal pool `CallThreadPool`. L'istanza della classe `CallThread` si occupa della gestione della chiamata. Essa è collegata ai livelli inferiore tramite il metodo: `execCall()` con cui riceverà notifica degli eventi collegati ai *Callgram*, e ai livelli superiore mediante il metodo `execLocalCall()`. Il metodo `execCall()` riceverà due notifiche: la prima all'arrivo della richiesta nel `Callgram`, la seconda all'avvenuto spedizione del `Callgram` con la risposta. Tra le due notifiche effet-

tua una invocazione, mediante `execLocalCall()`, dello Skeleton per eseguire la chiamata locale.

4.5.4 Ottimizzazioni Applicate

Il `Dispatcher` smista le richieste che gli arrivano eseguendo le chiamate mediante degli oggetti attivi `CallThread`. Questi sono prelevati da un pool: `CallThreadPool`. Dal lato client la richiesta viene gestita da un oggetto `Pusher`, che è un oggetto passivo in quanto riceve il controllo dal thread dell'applicazione. Quindi per la sua semplicità abbiamo deciso di non utilizzare per esso un pool.

4.6 Livello 5: Reference

Il livello Reference si occupa di referenziare un servizio[2]. Una discussione approfindita della problematica di referenziare in maniera univoca un servizio su una rete e della soluzione utilizzata rimandiamo al paragrafo 5.1 nel capitolo 5 sul modulo RIO, che è direttamente interessato al discorso. Per adesso diamo solo alcune definizioni.

Un servizio è definito in maniera elementare dalla coppia (nome, firma) del metodo che lo implementa. Dati incapsulati in una istanza della classe `Service`,

[2]Ricordiamo che un pool di servizi è implementato da una classe, mentre un singolo servizio da un metodo

a cui diamo nome di *riferimento semplice*. Nell'applicazione server un pool di servizi è referenziato con una istanza della classe `LocalServicesReference`, a cui diamo il nome di *riferimento locale*. Dal lato client, un servizio è referenziato mediante una istanza della classe `RemoteServiceReference`, cui diamo il nome di *riferimento remoto*.

4.7 Livello 6: Stub

Lo stub è la classe che piú a stretto contatto interagisce con l'applicazione Client. In primo luogo bisogna distinguere la classe `Stub`, e la classe che implementa lo stub.

Con il termine `Stub` indichiamo la classe: `run.stub_skeleton.Stub`, con il termine *stub* indichiamo una operazione svolta da una classe: cioé di permettere una invocazione remota.

La classe stub, viene generata a run-time dal modulo CLIO. Contiene tutti i metodi, quindi tutti i servizi, del pool di servizi, ma solo nominalmente. In realtà per ogni chiamata locale del client, essa inoltre la richiesta in remoto. Ad esempio, per un pool implementato mediante la classe `Aegis`, viene generata al volo la classe Aegis_Stub. La principale funzione di ciascuno di questi metodi è il Marshalling degli argomenti della chiamata, cioé la loro serializzazione in successione. In pratica svolge rispetto al Marshalling, la stessa funzione realizzata

dal metodo `writeObject()` per la serializzazione di un oggetto.

Al ritorno dalla chiamata remota, svolge la funzione duale: effettua l'Unmarshalling dei risultati. In Java una funzione può restituire un solo risultato quindi si tratta di deserializzare un solo oggetto. Qui rispetto all'Unmarshalling svolge la stessa funzione di `readObject()` rispetto alla deserializzazione.

La classe stub estende la classe `Stub` che fornisce il riferimento `RegistryServicesReference` al pool di servizi remoto. Per una discussione su questa classe rimandiamo al capitolo su RIO al paragrafo 5.1 a pagina 62.

4.8 Livello 6: Skeleton

Lo skeleton è la classe che piú a stretto contatto interagisce con l'applicazione Server. In primo luogo bisogna distinguere la classe `Skeleton`, e la classe che implementa lo skeleton.

Con il termine `Skeleton` indichiamo la classe: `run.stub_skeleton.Skeleton`, con il termine *skeleton* indichiamo una operazione svolta da una classe: cioé di permettere la ricezione di invocazioni remote.

La classe skeleton, viene generata a run-time dal modulo CLIO. Ad esempio, per un pool implementato mediante la classe `Aegis` viene generata al volo la classe `Aegis_Skeleton`. Contiene per ogni servizio del pool, un metodo che implementa la logica di Unmarshalling degli argomenti della chiamata, e il Marshalling della

risposta. In pratica svolge rispetto all'Unmarshalling, la stessa funzione realizzata dal metodo `readObject()` per la deserializzazione di un oggetto.

Al ritorno dalla chiamata, svolge la funzione duale: effettua il Marshalling dei risultati. In Java una funzione può restituire un solo risultato quindi si tratta di serializzare un solo oggetto. Qui rispetto al Marshalling svolge la stessa funzione di `writeObject()` rispetto alla serializzazione. La classe skeleton estende la classe `Skeleton` che fornisce il riferimento `Services` all'istanza della classe che implementa il pool di servizi. Per una discussione su questa classe rimandiamo al paragrafo 4.9.

I metodi della classe skeleton hanno formato: nomemetodo_numero. Per permettere il riferimento da remoto. Sull'argomento torneremo nel paragrafo 5.1.1 a pagina 65.

4.9 Livello 7: Server

Una classe per poter definire un pool di servizi, deve implementare una interfaccia di servizi remoti. Una interfaccia è definisce un pool di servizi remoti se estende l'interfaccia `Services`, che mostriamo nel listato 4.2. Ogni metodo dell'interfaccia di servizi remoti definisce un servizio remoto, ed ha come unico vincolo quello di dover dichiare nella lista di eccezione lanciabili l'eccezione `run.RemoteException`.

```
package run;

public interface Services
{

}
```

Listato 4.2: Services

4.10 Livello 7: Client

Un'applicazione client accede al pool di servizi remoto, mediante l'interfaccia di servizi. Dal punto di vista del programmatore, egli potrà invocare i metodi remoti allo stesso modo dei servizi locali. Dietro l'interfaccia di servizio vi è lo stub che inoltra le richieste in remoto.

4.11 Scenario d'Uso

Per conclude la nostra esposizione del modulo RUN, mostreremo i passi di una invocazione remota. Divideremo l'esposizione in due fasi: invio dei parametri e ritorno del risultato. Ci aiuteremo con un diagramma UML: il Collaboration Diagram, che mostra le interazioni tra le classi in termini di messaggi.

4.11.1 Invocazione di un Metodo Remoto

Osservando il diagramma di figura 4.11.1 a pagina 53, descriveremo i passi necessari per l'invio di una richiesta ad un metodo remota. Il flusso di controllo inizia dal client, che è posto nel diagramma nell'angolo in alto a destra, attraversa i di-

versi livelli del sistema fino a raggiungere l'applicazione server, posta nell'angolo in alto a sinistra. Il flusso di controllo attraversa il grafico in senso orario.

Lato Client Invio della richiesta.

1. L'applicazione Client interroga il registry sul servizio richiesto inviando un messaggio `lookup` alla classe `Registry`.
2. Invocazione di un servizio remoto mediante un riferimento di tipo `Services`. Il riferimento punta ad una classe stub, che si occuperà della chiamata remota.
3. Lo stub effettua il Marshalling degli argomenti della chiamata mediante l'uso di una classe `Container`.
4. Si usano le primitive messe a disposizione della classe interna `ContainerOutputStream`, per serializzare gli argomenti.
5. Fine del Marshalling, ed inizializzazione di un Callgram con il `Container` e l'indirizzo del server.
6. Tutte le chiamate remote del client sono gestite da una classe `Manager`, mediante l'invio di un messaggio `execRemoteCall()`.
7. La classe `Manager` crea un oggetto `Pusher` per gestire l'invio della richiesta.

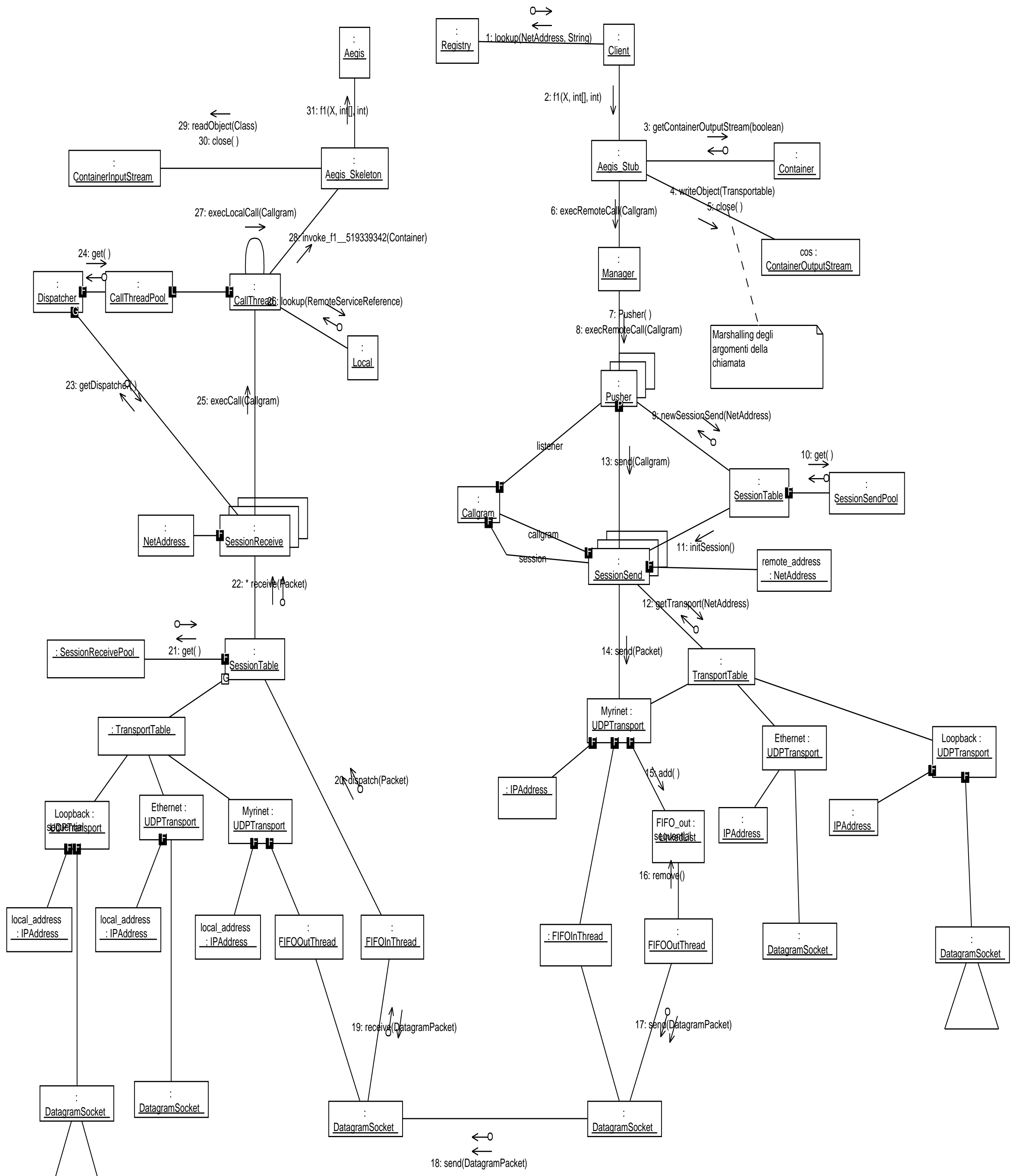

Figura 4.5: Invocazione di un metodo remoto (UML).

8. La chiamata remota è delegata all'istanza di `Pusher` mediante un messaggio `execRemoteCall()`.
9. Creazione di una nuova sessione per l'invio del `Callgram`.
10. La sessione `SessionSend` viene prelevata da un pool `SessionSendPool`.
11. Inizializzazione della sessione.
12. `SessionSend` tra i sui campi ha un riferimento al `Transport` per l'invio dei pacchetti al server. L'istanza di `Transport` si occupa di gestire la trasmissione / ritrasmissione dei pacchetti. L'istanza concreta di `Transport` nel nostro caso è un `UDPTransport`, che gestisce due code FIFO per l'invio ricezione dei pacchetti. Con questo passo termina l'inizializzazione della sessione.
13. Inizio della trasmissione con l'invio del `Callgram`.
14. Invio dei pacchetti contenenti segmenti dati del `Callgram`.
15. Accodamento dei pacchetti nella coda FIFO per la trasmissione.
16. I pacchetti da inviare sono prelevati dall'oggetto attivo `FIFOOutThread`
17. Al livello piú basso vi è l'istanza di `DatagramSocket` che si occupa dell'invio / ricezione dei datagrammi UDP su rete.
18. Trasmissione dei pacchetti sulla rete.

Lato Server Ricezione della richiesta e sua elaborazione.

19. L'istanza di `DatagramSocket` dal lato server riceve in maniera asincrona i datagrammi UDP. Esso è gestita da due thread: `FIFOOutThread`, `FIFOInThread`.

20. `UDPTransport` non esamina il contenuto dei pacchetti, ma li smista mediante la `SessionTable`, alla sessione a cui sono destinati.

21. `SessionTable` analizza il tipo di pacchetto ed il numero di sessione. Il primo pacchetto ricevuto sarà del tipo `INIT`. A cui `SessionTable` reagisce inizializzando una nuova sessione `SessionReceive` dal pool `SessionReceivePool`. Da quel momento tutti i pacchetti con quell'identificativo di sessione verranno smistati a quella sessione.

22. `SessionReceive` sul server e `SessionSend` sul client comunicano mediante il protocollo GO-BACK-N. Questo fino al trasferimento dell'intero Callgram. Si noti che sia SessionSend che SessionReceive sono oggetti attivi ma anche multi-thread. Vi è il thread principale codificato nel metodo `run()` che si occupa di gestire il protocollo di ritrasmissione. Ma l'istanza è percorsa da un secondo thread: quello di UDPTransport che si occupa della gestione degli arrivi dei pacchetti. Il punto di ingresso di questo secondo thread è il metodo `receive`.

23. `SessionReceive` notifica la classe `Dispather` del ricevimento di un Callgram.

24. Il `Dispatcher` smista la richiesta ad un oggetto attivo `CallThread` che si occupa della gestione locale della chiamata.
25. Avvio dell'esecuzione locale del metodo. Il thread di SessionReceive cessa la sua attività con questo passaggio di consegne.
26. Il riferimento allo skeleton viene recuperato con un accesso al registry.
27. Esecuzione locale della chiamata.
28. La chiamata viene effettuata mediante la classe skeleton.
29. Unmarshalling degli argomenti della chiamata.
30. Fine Unmarshalling.
31. Invio di un messaggio all'oggeto che implementa il pool di servizi. Il server esegue le operazioni connesse al servizio.

4.11.2 Ritorno del risultato da una Metodo Remoto

Osservando il diagramma in figura 4.11.2 a pagina 58, descriveremo i passi per il ritorno del risultato di una invocazione remota. Il flusso di controllo inizia dal server, che è posto nel diagramma nell'angolo in alto a sinistra, attraversa i diversi livelli del sistema fino a raggiungere l'applicazione client, posta nell'angolo in alto a destra. Il flusso di controllo attraversa il grafico in senso antiorario. Si noti che questo procedimento avviene anche nel caso in cui il metodo che implementa

il servizio restituisce un `void`. Infatti una chiamata remota è una operazione sincrona, ed al ritorno dal metodo, il client deve essere sicuro che l'esecuzione remota del servizio sia terminata.

Lato Server Elaborazione della richiesta, ed invio del risultato.

1. Il Server elabora la richiesta e restituisce il risultato allo skeleton.
2. Lo skeleton effettua il Marshalling del risultato,
3. costruisce un Callgram con la risposta.
4. Restituzione del risultato al `CallThread`.
5. Il `CallThread` richiede una nuova sessione alla `SessionTable`,
6. questa viene prelevata dal pool di sessioni `SessionSendPool`.
7. Inizializzazione della sessione,
8. con il `Transport` usato per la trasmissione.
9. L'invio del Callgram sulla sessione pone il CallThread in sleep in attesa del completamento della trasmissione, che gli verrà notificato mediante il pattern *Listener*.
10. `SessionSend` si occupa della gestione delle trasmisione / ritrasmisioni dei pacchetti.
11. `UDPTransport` riceve i singoli pacchetti, e li accoda nella nella lista FIFO dei pacchetti in uscita.

Figura 4.6: Ritorno del risultato da un metodo remoto (UML).

12. Prelevamento del prossimo pacchetto da spedire.

13. La trasmisione dei pacchetti avviene mediante datagrammi UDP.

14. Il datagramma UDP è inviato sulla rete.

Lato Client Ricezione della risposta.

14. L'istanza di `DatagramSocket` dal lato cliente riceve i pacchetti.

15. Viene letto il datagramma UDP trasmesso.

16. `UDPTransport` non esamina il contenuto dei pacchetti, ma li smista mediante la `SessionTable`, alla sessione cui sono destinati.

17. Il primo pacchetto che contiene una risposta è del tipo `RET` che ha come effetto inizializzare una nuova sessione `SessionReceive` dal pool `SessionReceivePool`.

18. La `SessionReceive` contiene un riferimento alla vecchia `SessionSend` con cui era stata mandata la richiesta. In modo da sapere a chi notificare gli eventi relativi al Callgram.

19. `SessionTable` analizza il tipo di pacchetto ed il numero di sessione, e lo smista alla sessione. A questo punto `SessionReceive` sul client `SessionSend` sul server comunicano mediante il protocollo GO-BACK-N, fino al ricevimento dell'intero Callgram.

20. Una volta che il Callgram è stato ricostruito rappresenta la ricezione della risposta da remota. `SessionReceive` notifica dell'evento l'istanza di `Pusher`, che aveva in precedenza inviato la richiesta. Il riferimento a questa istanza è stato ricavato dalla precedente `SessionSend`.

21. Restituzione del risultato al `Manager`.

22. Restituzione della richiesta alla classe stub.

23. Preparazione della fase di Unmarshalling del risultato.

24. Lettura del risultato.

25. Fine Unmarshalling.

26. Il client riceve la risposta della chiamata remota.

4.12 Le Proprietà del Package `run.session`

Il package fornisce una "interfaccia" che possiamo formalizzare per i servizi offerti. Questi servizi possono essere espressi in termini di sequenze di ingresso (*input(P)*), di sequenze di uscita (*output(P)*) e di sequenze ammissibili (*seq(P)*). Come mostrato in [AW97]. In primo luogo detto M ogni elemento di questi insiemi. M rappresenta un messaggio trasmesso tra due nodi. M contiene i gli argomenti dei parametri di una chiamata remota. Oppure M contiene la risposta di una chiamata remota. In ogni caso sono dati codificati (dati serializzati).

Definiamo come:

input(P) Sequenze di possibili messaggi M, che rappresentano parametri di una chiamata remota.

output(P) Sequenze di possibili messaggi M' che rappresentano il risultato di una chiamata remota.

seq(P) Esiste una funzione φ che mappa uno a uno i messaggi da *input(P)* a *output(P)* che gode delle seguenti proprietà:

Integrità I messaggi sono ricevuti senza corruzione.

Nessun duplicato I messaggi non sono ricevuti piú di una volta.

Liveness Ogni messaggio è ricevuto una sola volta.

Capitolo 5

RIO: Cluster Registry

Il modulo RIO interpreta il ruolo di cluster registry. La funzione di registry è un servizio che permette alle applicazioni di individuare il nodo su cui è installato un servizio messo a disposizione del cluster. Ma è anche un meccanismo interno alle applicazioni client e server, che permette alla prima di individuare il servizio remoto, e alla seconda di individuare l'istanza locale a cui passare la richiesta remota.

5.1 Riferimento di un Servizio

Invocare un servizio significa poterlo referenziare, cioé avere un modo per individuarlo in maniera univoca.

Per referenziare una applicazione abbiamo fornito la classe astratta `NetAddress`

(4.2.3) che fornisce l'astrazione di un indirizzo di rete. L'unica classe che, per adesso, estende `NetAddress` è `IPAddress`. Questa incapsula indirizzo IP e numero di porta di un host. Queste informazioni bastano ad individuare una applicazione. Ma una applicazione puó istanziare piú pool di servizi. Ed inoltre un pool per definizione, puó contenere piú di un servizio.

Un pool di servizi è referenziato da un nome con cui è stato registrato. Ogni servizio all'interno del pool è individuato da una coppia: (nome, firma) del metodo che lo implementa. Questi dati sono stati incapsulati in una classe `Service` che chiameremo *riferimento semplice*. In quanto di per se non basta a referenziare un servizio.

Sono due le classi che possono effettuare questa operazione: `RemoteServiceReference` e `LocalServicesReference`, cui già abbiamo fatto cenno nel capitolo 4. A queste ve ne aggiungeremo un'altra: `RegistryServicesReference`. Queste classi verranno indicate con il termine *riferimento* perchè permettono di referenziare in varie forme un servizio.

Per non creare confusione vorremmo subito chiarire la differenza che esiste tra i *riferimenti* che sono istanze delle classi qui esposte, ed i riferimenti Java che sono il meccanismo usato dal linguaggio per indirizzare un oggetto. Per *riferimento*, d'ora in poi intenderemo sempre l'istanza di classe, e quando si vorrà parlare invece del tipo di Java vi sarà fatto esplicito cenno chiamandolo *riferimento Java.*

Descriveremo scopo ed uso di ogni riferimento aiutandoci con la tabella 5.1.

Nome	**Contiene**				**Usato da**
	Ind. Host	Nome pool	Lista Servizi	Rif. Semplice	
`RemoteServiceReference`	√	√		√	Client
`LocalServicesReferences`		√	√		Server
`RegistryServicesReference`	√	√	√		Registry

Tabella 5.1: Elenco dei riferimenti e loro uso nel sistema.

La classe `RemoteServiceReference` individua: (1) una applicazione, e all'interno di questa (2) un pool di servizi, ed all'interno del pool (3) un singolo servizio.

L'applicazione è individuata mediante una istanza della superclasse `NetAddress`. Il pool è individuato mediante la stringa del nome con cui è registrato. Il servizio è individuato con un riferimento semplice. Questa classe è utilizzata dal lato client per referenziare un servizio remoto. D'ora in poi la indicheremo con l'espressione *riferimento remoto*

La classe `LocalServicesReference` individua: (1) un pool di servizi ed elenca (2) tutti i servizi disponibili all'interno del pool. L'applicazione è individuata implicitamente come quella che contiene la struttura dati, il pool è descritto con il nome con cui si è registrato, la lista di servizi mediante un array riferimenti semplici. Questa classe è usata dal lato server per referenziare tutti i servizi messi a disposizione. D'ora in poi la indicheremo con l'espressione *riferimento locale.*

In aggiunta vi è la classe `RegistryServicesReference` che individua: (1) una applicazione, (2) un pool di servizi, (3) tutti i servizi disponibili all'interno del pool. Questi dati sono memorizzati con la terna: (1) `NetAddress`, (2) nome del servizio, (3) array di riferimenti semplici. Questa classe è usata dal registry per individuare in maniera univoca un pool di servizi all'interno del cluster. D'ora in poi la chiameremo per brevità: *riferimento registry*.

Come si puó notare i dati memorizzati in `RegistryServiceReference` sono un superinsieme di quelli presenti negli altri due tipi. In effetti, oggetti delle prime due classi sono istanziate in maniera diretta o indiretta da un riferimento del terzo tipo. Essa rappresenta l'unità di elaborazione che è alla base di questo modulo.

5.1.1 Riferimento di un Servizio ed Overloading di Java

Una richiesta da remoto contiene tra i suoi dati un riferimento remoto, quindi nome del pool, e riferimento semplice del servizio cioé nome e firma del metodo che lo implementa. Non vi possono essere due pool con lo stesso nome. Ma vi possono essere due servizi con lo stesso nome se la firma li distingue.

Nella nostra architettura per alleggerire il protocollo di serializzazione, il Marshalling e l'Unmarshalling dei dati sono effettuati dallo stub e dallo skeleton.

Lo stub implementa nominalmente tutti i servizi del pool, ma solo per ef-

fettuare il Marshalling degli argomenti. Al momento della chiamata remota se vi sono due servizi con lo stesso nome, ma con firme diverse, c'è il meccanismo dell'overloading di Java per distinguerli.

Ma i metodi della classe skeleton hanno tutti come argomento una istanza della classe `Container`, per permetterli di effettuare l'Unmarshalling. Ma a questo punto sorge il problema di come poter distinguere, al momento dell'invocazione dal lato server, due metodi con stesso nome ma con diverse firme, se tutti i metodi devono necessariamente avere lo stesso argomento `Container`.

Per risolvere il problema i metodi della classe skeleton hanno un *nome esteso*. In cui sono codificati il nome del metodo originario e la sua lista di argomenti. La codifica è la seguente: "nomeservizio_numero". Dove nomeservizio è il nome del metodo originario, mentre con numero è il valore hash calcolato sulla stringa della firma del metodo originario. Nel caso in cui il valore è negativo il segno meno viene convertito nell'underscore: '_'.

Quando arriva una richiesta da remoto, l'invocazione è effettuato sul metodo con il nome esteso e da questo, dopo l'Unmarshalling degli argomenti, sul nome originario. L'Unmarshalling ha prodotto oggetti con un loro tipo. Ci penserà a questo punto il meccanismo di overloading di Java a distinguere i metodi, usando i tipi degli argomenti della chiamata.

5.2 Architettura

Il modulo è diviso per le sue funzionalità in due parti: un registry locale all'applicazione[1] ed un registry globale al cluster[2]. Le funzioni del registry locale sono private, e presenti in ogni applicazione sia dal lato client che dal lato server, ed hanno lo scopo di poter referenziare internamente all'applicazione un servizio. Le funzioni del registry globale al cluster sono condivise, sono quelle che associano riferimenti a nomi, ed hanno validità sull'intero cluster.

5.3 Funzioni Locali

La parte locale del registry svolge all'interno delle applicazioni operazioni che trattano di riferimenti locali e Java. Quindi operazioni che influiscono solo su strutture dati interne e non sono visibili all'esterno. In particolare, una prima categoria di operazioni tratta della generazione di istanze di classi stub e skeleton. Una seconda categoria di operazioni gestisce una tabella di riferimenti locale ai pool di servizi ospitati dall'applicazione.

L'istanziazione di un oggetto dalla classe stub / skeleton è una funzione coesa

[1] Con il termine locale ci riferiamo al campo di validità dei riferimenti che tratta. Quindi riferimenti del tipo locale o Java

[2] Con il termine globale, intendiamo operazioni che trattano di riferimenti che sono validi sull'intero cluster. Quindi del tipo remoto o registry.

con la generazione delle classi stub / skeleton, operazione svolta dal modulo CLIO, e la tratteremo in dettaglio nel capitolo 6.

La registrazione di un pool di servizi è effettuata associando un nome di pool ad un riferimento locale. L'operazione di *lookup* viene effettuata all'arrivo di una richiesta da remoto. Questa contiene nel suo riferimento remoto il nome del pool e il riferimento semplice del servizio richiesto. L'operazione di *lookup* allora non fa altro che convertire il riferimento da remoto a locale. Il riferimento locale contiene il riferimento Java allo skeleton del servizio. A questo punto sarà facile risalire all'oggetto che implementa il pool di servizi.

Tutte le suddette operazioni sono state implementate nella classe `Local`.

5.3.1 Classe `Local`

Le operazioni locali di RIO sono implementate nella classe `Local`. Fornisce quattro operazioni: due per la gestione di una tabella di riferimenti locali, e due per la generazione di istanze di classi *stub* e *skeleton*. La scelta di inserire questi ultimi due metodi qui è stata motivata dal fatto che le classi stub e skeleton ereditano dalle superclassi `Stub` e `Skeleton`[3] i campi per i riferimenti ai servizi. In particolare la classe `Skeleton` contiene un riferimento Java all'oggetto che implementa il pool. La classe `Stub` contiene un riferimento remoto al pool di servizi.

[3] Per una descrizione delle differenze tra classi *stub*, *skeleton* e le superclassi `Stub` e `Skeleton` rimandiamo alla descrizione del paragrafo 4.7 a pagina 48 e del paragrafo 4.8 a pagina 49

Aiutandoci con la figura 5.1, descrivere le operazioni svolte da questa classe suddividendole per lato Server e Client.

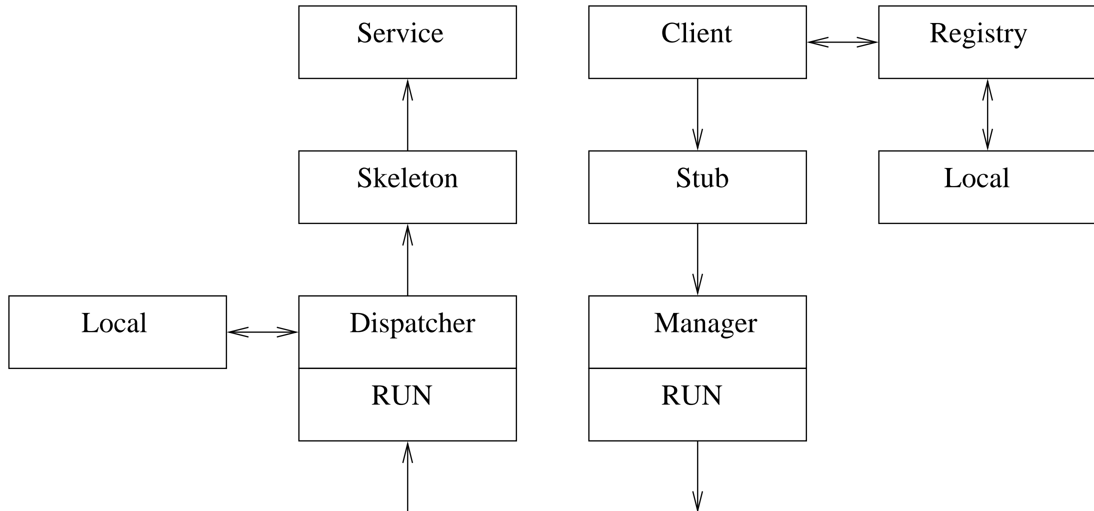

Figura 5.1: Funzioni di RIO locali in ogni applicazione.

Lato Server

La parte locale sul server contiene l'associazione tra il nome del servizio e l'istanza della classe `LocalServicesReference`. E fornisce due primitive: (1) `register()` e (2) `lookup()`, rispettivamente per registrare un riferimento e recuperarlo.

Registrare localmente il servizio significa associare il nome del pool di servizi al riferimento locale. Il riferimento locale tra i suoi campi ha il riferimento Java allo skeleton. La classe skeleton eredita dalla superclasse `Skeleton` il riferimento al pool di servizi.

In questo modo quando verrà interrogato il registry, restituirà il riferimento locale, da questo seguendo i collegamenti sarà possibile risalire fino all'istanza del pool di servizi.

Lato Client

L'unico metodo offerto dalla classe `Local` al client è l'instanziazione della classe stub. La superclasse `Stub` della classe stub, tra i suoi campi, ha un riferimento registry, che permette di instradare le richiesta a remoto. Instanziare la classe stub all'interno della classe `Local` permette di inizializzare in maniera trasparente questo campo.

5.4 Funzioni Globali

Le operazioni globali di RIO sono implementate come un pool di servizi. Quasi del tutto identico, per architettura, a quelli che puó implementare l'utente. Il pool è registrato sotto il nome: *"registry"*. Ed offre due servizi: (1) registrer e (2) lookup; che, come suggeriscono i nomi, registrano un pool di servizio e ne restituiscono un riferimento. Come qualunque alto pool, necessita per il suo funzionamento di una classe stub e di una skeleton, queste sono state implementate, rispettivamente, con le classi `Out` ed `In`. Inoltre per facilitarne l'utilizzo dalle applicazioni create dall'utente è stata fornita la classe `Registry` che fornisce due

facilities: `register()` e `lookup()` che permettono all'utente di elidere i dettagli della chiamata remota ai servizi "register" e "lookup". L'interazione tra queste classi è mostrata in figura 5.2.

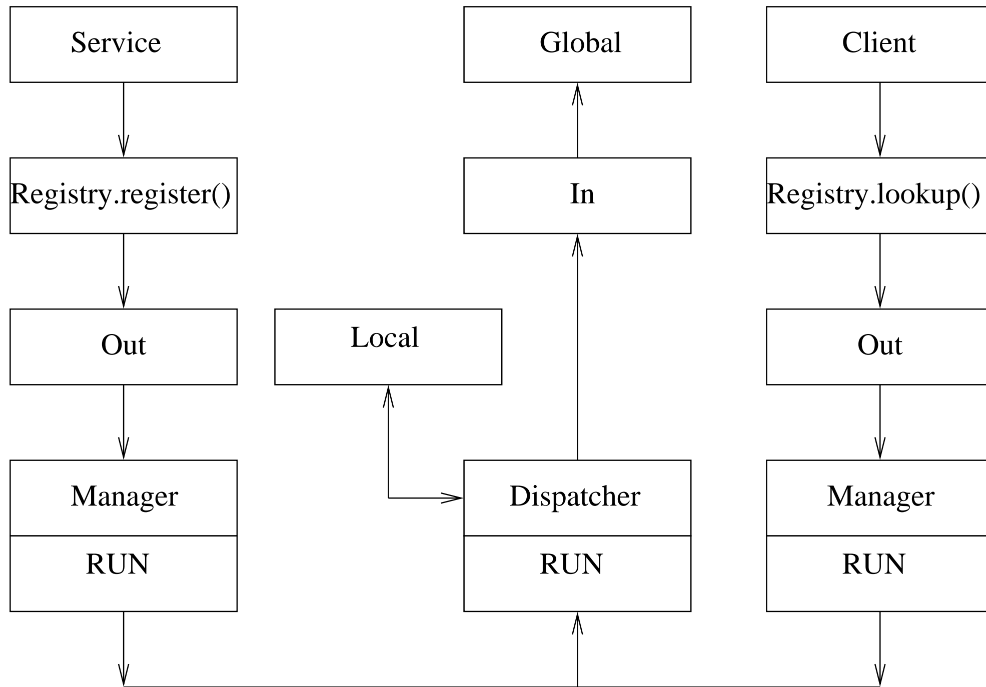

Figura 5.2: Funzioni di RIO globali al cluster.

5.4.1 Lato Server

L'applicazione server istanzia un oggetto dalla classe che implementa il pool di servizi. Lo registra invocando il metodo `register()` della classe `Registry`. Questa effettua due registrazioni. La prima in locale, in cui registra il riferimento locale nella classe `Local`. La seconda in remoto invocando il servizio remoto "register"

del servizio "registry". Tale invocazione remota è fatta in maniera trasparente dal metodo `register()` della classe `Registry`.

5.4.2 Lato Client

L'applicazione Client necessita, per operare, del riferimento Java alla classe *stub*. Per ottenerlo invoca il metodo `lookup()` della classe `Registry`. Questa effettua due operazioni. In primo luogo una invocazione remota al servizio "lookup" del pool "registry". Che gli restituisce un riferimento registry al servizio richiesto. Come seconda operazione istanzia una classe stub, inizializzandolo con il riferimento registry al pool. Infatti questa eredita dalla superclasse `Stub` un campo di tipo `RegistryServicesReference`. Ad ogni invocazione remota il riferimento registry (che descrive l'intero pool) verrà convertito nel riferimento remoto che descrive il singolo servizio, mediante il metodo di conversione: `toRemoteServiceReference()`.

5.4.3 Lato Registry

Le operazioni di Registry sono implementate mediante un pool di servizi. I servizi che offre riguardano riferimenti che hanno validità sull'intero cluster, quindi del tipo registry. Il servizio di Registry è codificato nella nostra architettura nelle classi `Global`, `In`, `Out`. Il pool ha nome prefissato "registry" ed offre due servizi:

"register" e "lookup", implementati con i metodi `register()` e `lookup()`. Il pool viene registrato alla classe `Local` in modo da essere rintracciabile per le richieste da remoto.

Per come è strutturata la nostra architettura, una classe che implementa un pool di servizi necessita di classi *stub* e *skeleton*, queste sono implementate dalle classi `Out` e `In`. La prima implementa nominalmente i metodi di `Global`, ma in realtà instrada le richieste a remoto. La seconda riceve le richieste da remoto ed invoca i metodi di `Global`

La funzioni svolte da `Global` sono in realtà molto semplici:

- `register()` associa un nome di pool (quindi una istanza di `String`) ad un riferimento registry (quindi una istanza di `RegistryServicesReference`).
- `lookup()` dato il nome di un pool restituisce il riferimento registry associato.

L'unica eccezione rispetto ad un pool definito dall'utente è che l'implementazione dei metodi delle classi stub e skeleton sono `static`. Questo per evitare di instanziare un oggetto *dummy* la cui unica funzione è permetterci di accedere ai metodi. Ma risulta una soluzione logica in quanto se i servizi del registry sono *unici* sull'intero cluster, lo sono ancora di piú all'interno della singola applicazione.

L'idea di implementare il Registry mediante un pool di servizi ci ha permesso

di evitare di implementare un protocollo di livello applicazione tra le applicazioni Client / Server per interrogare il Registry. E ci ha fornito una soluzione elegante che ben si innesta nell'architettura del sistema.

5.4.4 Trasferimento di un Riferimento Registry

Come descritto in precedenza un riferimento registry necessita di essere trasferito dal nodo dove è in esecuzione l'applicazione client al nodo dove risiede il registry. Questo perché costituisce uno dei parametri del servizio "register", ed il risultato del servizio "lookup". Viene utilizzata la nostra architettura di trasferimento di oggetti, e quindi come descriveremo nel paragrafo 6.3.5 la rispettiva classe deve implementare l'interfaccia `Transportable`, ma se cosí fosse dovrebbe verrebbe modificata a run-time da CLIO per renderla `FastTransportable`. Ma sempre nel paragrafo 6.3.5 spieghiamo che una classe può implementare l'interfaccia `FastTransportable` ed evitare di essere modificata, ma al costo di dover definire i metodi, campi e costruttori necessari.

Cosí per la classe `RegistryServicesReference`, abbiamo fornito i serializzatori / deserializzatori: `readObject(), writeObject()`, il metodo di supporto: `sizeOf()`, il costruttore di default `init()`, ed il campo `SUID`.

Questa scelta è nata in fase di programmazione, in quanto l'assenza del modulo CLIO non permetteva di sperimentare altrimenti l'architettura. Ma a regime con-

tinua ad avere senso. In quanto viene alleggerita la fase di esecuzione, non dovendo modificare, anche solo per una volta, la classe `RegistryServicesReference`.

Una ultima considerazione per l'interfaccia `FastTransportable`. L'implementazione dell'interfaccia non ha senso per gli altri due tipi di riferimenti: locali e remoto. Il riferimento locale per definizione ha validità solo all'interno dell'applicazione e non viene trasmessa ad altri nodi. Mentre il riferimento remoto effettivamente viene trasferito da una applicazione all'altra. Ma i suoi campi (nome pool e riferimento semplice al servizio) sono codificati nel protocollo di trasporto, e precisamente nel pacchetto `INIT`, come mostrato nella tabella 4.2, a pagina 34.

5.5 Scenario d'Uso

Mostreremo adesso due scenari d'uso del modulo RIO: la registrazione e il lookup di un pool di servizi. Ci aiuteremo con due diagrammi di UML: Collaboration Diagram. Da questi diagrammi abbiamo omesso quasi tutte le classi di RUN. Per non appesantire troppo il grafico. Quello che è interessante notare, è come i servizi di registry siano stati implementati come servizi "register" e "lookup" del pool "registry", ed implementati dalla classe `Global`.

5.5.1 Registrazione di un Servizio

Il collaboration diagram di figura 5.3 ci mostra la fase di registrazione di un pool di servizi.

Descrivermo passo passo le operazioni rappresentate.

1. Il servizio invia la richiesta di registrazione alla classe `Registry`.
2. Creazione di un riferimento registry.
3. Conversione del riferimento registry in riferimento locale, e sua registrazione nella classe `Local`
4. Invio del riferimento registry a remoto. La richiesta di registrazione viene inviata allo classe stub `Out`.
5. La classe stub `Out` invia la richiesta a remoto, eseguendo una `execRemoteCall()` sulla classe `Manager`
6. La chiamata remota, segue l'iter attraverso i layer di RUN Il messaggio di chiamata remota giunge ad un `CallThread` che la eseguirà nel suo thread di esecuzione.
7. Il messaggio contiene la richiesta per il pool "registry". Una interrogazione alla classe `Local` permette di ricavare il riferimento locale e da questo il riferimento Java alla skeleton.

8. Viene invocato il metodo register sulla classe skeleton `In`.

9. La classe skeleton decodifica la richiesta ed invoca il metodo register della classe `Global`.

5.5.2 *Lookup* di un Servizio

Il collaboration diagram di figura 5.4 ci mostra la fase di lookup di un pool di servizi.

Descriviamo di seguito i passi che descrive.

1. Il client chiede al `Registry` un riferimento al servizio remoto.

2. Invio della richiesta allo stub `Out`.

3. Invio della richiesta a remoto mediante la classe `Manager`, che qui rappresenta l'intero modulo RUN.

4. Arrivo della chiamata remota alla classe `CallThread`.

5. Ricerca del pool "registry" nella classe `Local`.

6. Invocazione del servizio lookup sullo skeleton `In`.

7. Chiamata al servizio "lookup" del pool "registry".

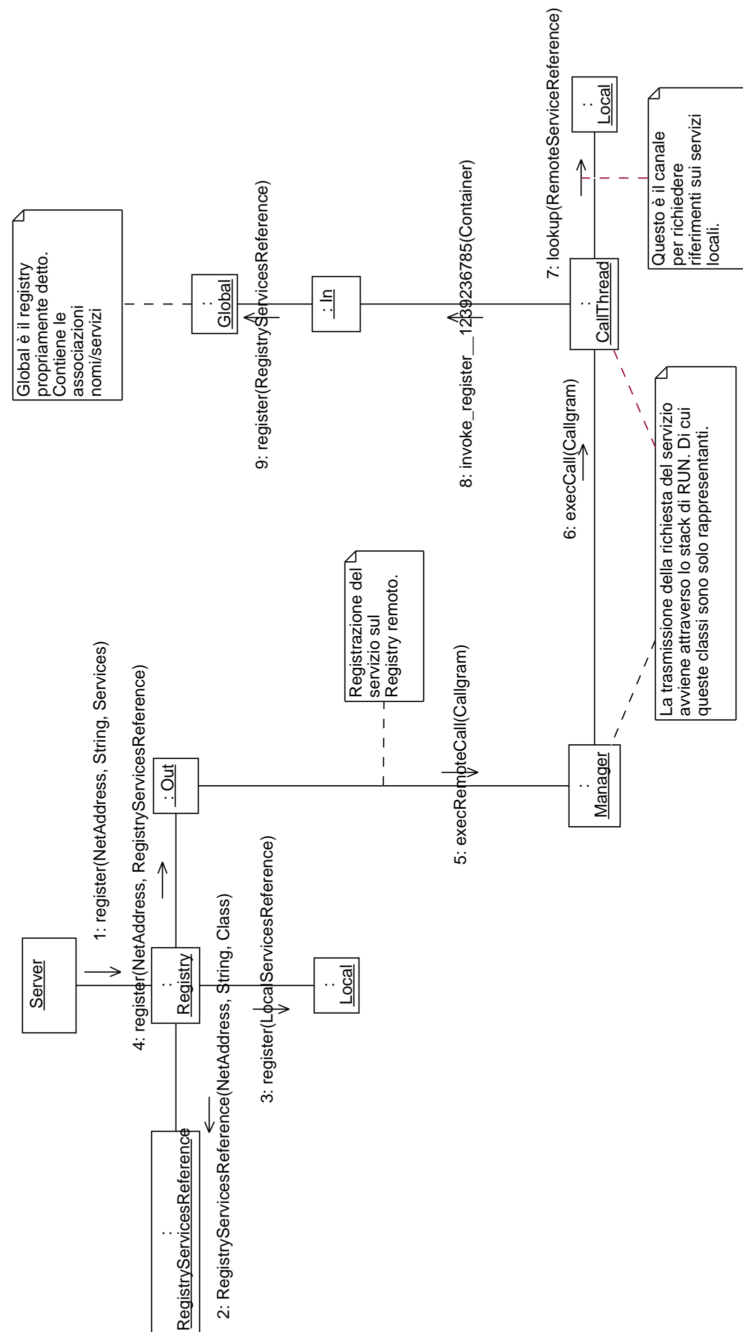

Figura 5.3: Registrazione di un pool di servizi (UML).

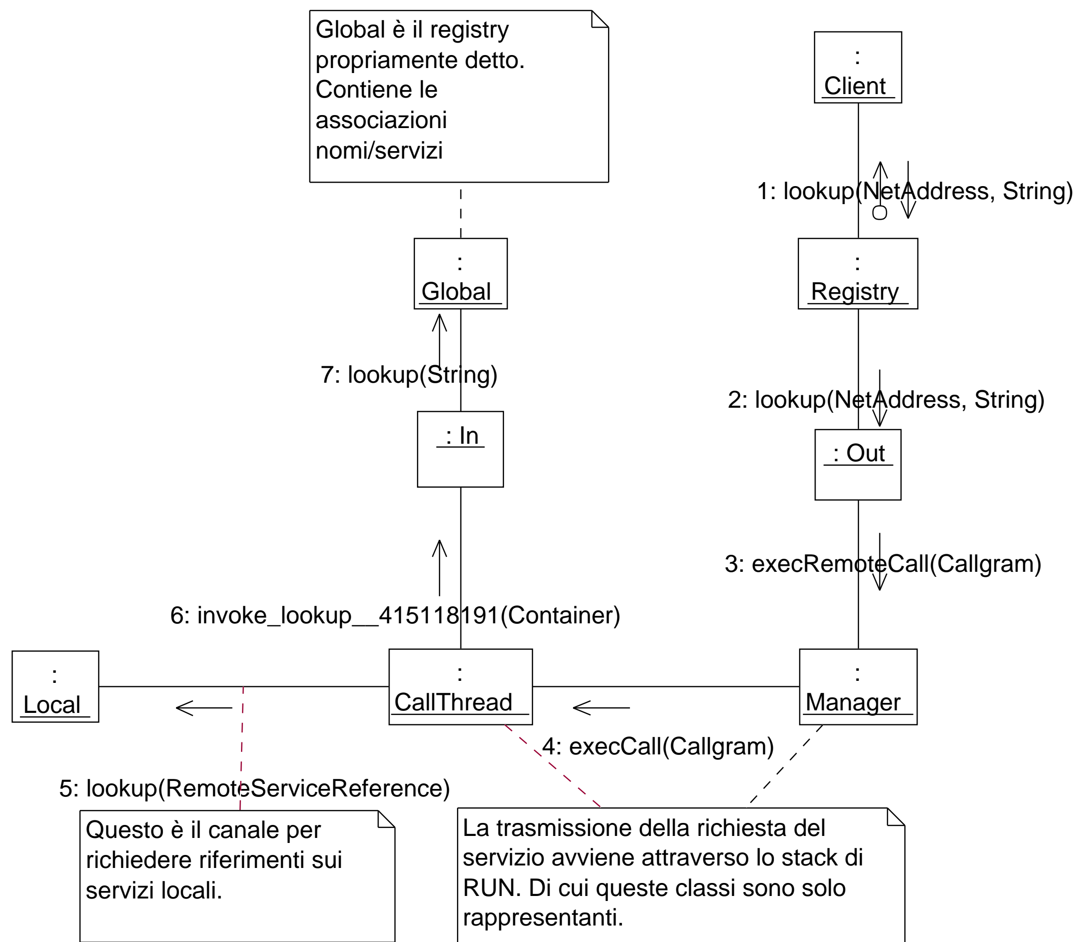

Figura 5.4: Ricerca di un servizio (UML).

Capitolo 6

CLIO: Cluster ClassLoader

Il modulo CLIO è composto da due sotto-moduli: il primo svolge le funzioni di ClassLoader, il secondo svolge le funzioni di generatore di classi. I due sotto-moduli sono fortementi accoppiati, ciò motiva la scelta di inserirli nello stesso package.

6.1 Classi Attive, Reattive e Passive

Distinguiamo in primo luogo le classi di Java in base al loro ruolo nell'architettura di comunicazione. In base a questa distinzione abbiamo: classi *attive*, classi *reattive* e classi *passive*.

Una classe è **attiva** se definisce uno stub, uno skeleton oppure implementa l'argomento di una chiamata remota. Una classe si definisce **reattiva** se im-

plementa una interfaccia di servizi remoti, oppure puó essere argomento di una chiamata remota. Tutte le altre sono dette **passive**.

6.1.1 Classi Attive

Le classi *attive* si definiscono tali perchè svolgono un ruolo nell'architettura di comunicazione. Le classi stub e skeleton sono attive, perché si occupano del Marshalling / Unmarshalling degli argomenti delle chiamate. Cosí come le classi che implementano il tipo di un argomento di una chiamata remota hanno un ruolo attivo. Perché si occupano di serializzare / deserializzare proprie istanze sullo stream di comunicazione tramite i serializzatori.

Riassumendo, le classi attive sono:

- Classi che definiscono uno stub. Sono riconosciute dal nome che ha suffisso "`_Stub`".
- Classi che definiscono uno skeleton. Sono riconosciute dal nome che ha suffisso "`_Skeleton`".
- Classi che definiscono un argomento oppure un risultato di una chiamata remota. Sono riconosciute in quanto implementano l'interfaccia `FastTransportable`.

Ad ogni classe attiva corrisponde una classe reattiva.

6.1.2 Classi Reattive

Le classi "reattive" si definiscono tali in quanto il nostro sistema *reagisce* ad ogni loro modifica, generando la relativa classe attiva. Le classi reattive sono:

- Classi che definiscono un pool di servizi remoti. Sono riconosciute in quanto implementano l'interfaccia `Services`.
- Classi le cui istanze possono essere argomenti di una chiamata remota. Sono riconosciute in quanto implementano l'interfaccia `Transportable`.

La classi che implementano gli argomenti di una chiamata remota appaiono sia nella categoria attive che reattive, ma con una differenza. Esse appaiono nella categoria reattive perchè *possono* essere argomento di una chiamata remota. Appaiono nella categoria attive in quanto *sono* utilizzate come argomento di una chiamata remota. Appaiono nella prima categoria perchè implementano l'interfaccia `Transportable`. Appaiono nella seconda in quanto implementano l'interfaccia `FastTransportable`.

6.1.3 Classi Passive

Le classi passive non hanno nessun ruolo nella nostra architettura E sono tutte le altre classi che non sono attive o reattive.

6.2 ClusterClassLoader

ClusterClassLoader è il ClassLoader definito per la nostra architettura. Un ClassLoader di Java ha la funzione di caricare nella Java Virtual Machine (d'ora in poi JVM) una classe. Esempio di ClassLoader è lo `URLClassLoader` che permette di caricare un classe da un insieme di URL. ClusterClassLoader (d'ora in poi CCL) esegue un procedimento simile permettondo il caricamento di una classe sia dalle directory specificate nel CLASSPATH, che dal Repository definito dall'utente. In piú CCL interagisce con il modulo di generazioni di classi (Generator), per la generazione in automatico delle classi attive.

6.2.1 Delegation Model di Java

Come descritto in [New00] il delegation model di Java è il sistema introdotto con la versione `1.2`, che definisce il modello di ricerca di una classe.

Vi sono tre istanze di ClassLoader che hanno particolare rilievo nella JVM:

- BootClassLoader: si occupa di caricare le classi base del Java Runtime Environment (JRE). Sono memorizzate nell'archivio `rt.jar` posto nella directory `jre/lib`.
- ExtClassLoader: si occupa di caricare le classi che definiscono estensione

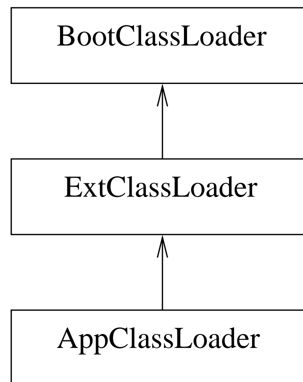

Figura 6.1: Gerarchia di AppClassLoader

standard del sistema. Sono le classi memorizzate nella directory `jre/lib/classes/`, oppure contenute in archivi posti nella directory `jre/lib/ext/`.

- AppClassLoader: si occupa di caricare le classi memorizzate nelle directory specificate nel CLASSPATH.

Tutti i ClassLoader sono legati gerarchicamente, grazie ad un campo che specifica per ognuno chi è il relativo padre.

Un ClassLoader quando riceve la richiesta di caricamento di una classe, cioè il messaggio `findClass()`, delega il caricamento al padre, e solo se questo fallisce tenta di caricarla.

Questa politica fa in modo che per un ClassLoader, che ha definito come padre l'AppClassLoader (fig. 6.1), l'ordine di ricerca sia: (1) BootClassLoader, (2) ExtClassLoader, (3) AppClassLoader, prima che possa provare lui a caricare

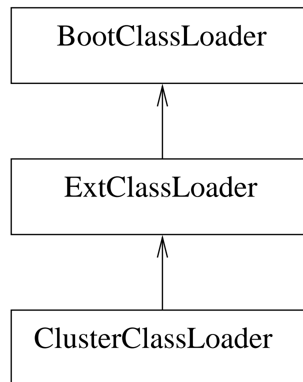

Figura 6.2: Gerarchia di ClusterClassLoader

la classe. Questa tecnica evita di caricare più di una volta la stessa definizione di classe. Infatti classi base e comuni del sistema, come ad esempio `String` oppure `Applet` verranno caricate una sola volta da un solo ClassLoader, che le carica da locazioni ben definite. Mentre classi definite dall'utente possono essere poste dovunque nel sistema, e l'istanza specifica del ClassLoader si occuperà del caricamento. Si noti che la stessa definizione di classe caricata da due ClassLoader diversi rappresentano a run-time due diversi tipi ([Con00]).

La gerarchia per il nostro ClusterClassLoader è semplice: definisce come padre l'ExtClassLoader, bypassando l'AppClassLoader (fig. 6.2). In questo modo le classi che definiscono le API di base, e le classi che definiscono estensioni del sistema saranno caricate senza modifiche. Mentre sarà reso possibile la modifica delle classi dell'applicazione.

Ma per fare questo CCL dovrà assumere quei compiti che prima erano del-

l'AppClassLoader. Cioè la ricerca dalle directory specificate nel CLASSPATH.

Per fare questo l'esecuzione di una applicazione Java sarà fatta da shell con una riga di comando del tipo:

```
# java clio.Main nome_classe
```

In modo che tutte le richieste per le classi dell'applicazione vengano filtrate dal modulo CLIO. La classe clio.Main verrà caricata dall'AppClassLoader ma quelle dell'applicazione dal ClusterClassLoader.

6.2.2 Percorso di Ricerca di ClusterClassLoader

CCL definisce il percorso di ricerca nel seguente modo:

1. User Repository, associato per default alla directory `~/repository/`
2. Directory poste nel CLASSPATH.

Lo User Repository è utilizzato per memorizzare le classi generate. In modo da non ripetere, se non in caso necessario, la loro generazione.

6.3 Generator: Generatore di Classi

Il modulo Generator è l'altra metà di CLIO. Si occupa di generare a run-time le classi necessarie per il funzionamento dell'architettura di comunicazione. La

generazione delle classi è effettuata mediante una tecnica detta di ByteCode Engineering.

6.3.1 ByteCode Engineering

Il ByteCode nella piattaforma di Java, riveste il ruolo di ciò che nei tradizionali sistemi nativi è il linguaggio macchina. Definisce cioé un linguaggio a basso livello che la macchina (virtuale) può eseguire direttamente. Il ByteCode si trova comunemente nei files con estensione ".class", che d'ora in poi chiameremo *file di definizione di una classe.*

Come definito in [Dah98a] l'ingegneria del ByteCode fornisce quelle metodologie che permettono di modificare una definizione di classe alterandone il ByteCode. Questo avviene prima che la definizione di classe diventi un tipo a run-time, cioé prima delle operazione di definizione e linking ([LY99]).

La possibilità di poter modificare una classe, mette a disposizione del programmatore tutta una serie di strumenti che prima gli erano precluse, se non disponendo dei sorgenti. Analisi dinamica delle risorse occupate da una classe, riflessione statica, sono solo alcuni degli esempi che troviamo in [Dah98a].

6.3.2 BCEL: ByteCode Engineering Library

Come descritto in [Dah98b] le funzioni offerte dalla libreria si possono raggruppare in tre categorie: (1) analisi di una classe, (2) class repository, (3) generazione di un classe.

I dati contenuti in un file di definizione di classe sono rappresentati in una istanza di `JavaClass`. Che fornisce tutta una serie di metodi per analizzare una classe. Si noti che tutte le informazione ottenibile da una istanza di `JavaClass` sono orientate alla singola definizione della classe, senza tener conto dell'ereditarietà. Quindi si potranno avere informazioni sui campi definiti all'interno della classe, ma non su quelli che sono stati ereditati da una superclasse, se non prima caricando il relativo file di definizione, ed istanziato una `JavaClass` che la rappresenta.

Il class repository fornisce uno strumento per caricare dal CLASSPATH una definzione di classe, e ottenere da questo la relativa istanza di `JavaClass`.

Infine lo strumento piú potente è rappresentato dalla possibilità di modificare i dati contenuti in una istanza di `JavaClass`, e da questa modifica generare un nuova definizione di classe e quindi un nuovo tipo. Modificare una classe, comprende anche la possibilità di partire da una definizione di classe vuota. Permettendo quindi di generare del tutto un nuovo tipo.

Riassumendo:

1. Abbiamo usato le funzionalità di ClassRepository per caricare in memoria una definizione di classe, e fornire quindi le funzionalità che prima erano assegnate all'AppClassLoader.

2. Abbiamo usato le funzionalità di analisi di una classe per verificare se questa rientra nella categoria di classe reattiva.

3. Abbiamo usato le funzionalità di modifica di una classe per generare le classi attive.

6.3.3 Generazione di una Classe

Prima di tutto bisogna precisare il significato di espressioni come *modificare una classe* e *generare una classe*. Per *modificare una classe* intendiamo sia partire da una definizione completa di una classe, che da una definizione vuota. In entrambi i casi (e non solo quindi nel secondo caso) l'effetto è *generare una nuova definizione di classe*, e da questa un nuovo tipo.

Il processo di generazione parte sempre con la richiesta (in maniera diretta o indiretta) da parte dell'applicazione di una classe attiva. La richiesta è diretta quando l'applicazione usa la classe richiesta. La richiesta è indiretta se la classe definisce il tipo di uno dei campi della classe richiesta dall'applicazione. In ogni caso, il sistema analizza la classe reattiva associata e produce la classe attiva.

6.3.4 Motivazioni

In questo paragrafo cerceremo di spiegare i motivi che ci hanno spinto a costruire il meccanismo di distinzione tra classi reattive e passive, e cercheremo di descrivere i vincoli che ci hanno portati nello scegliere quando generare le classi attive. Utilizzeremo le classi che implementano i tipi di una chiamata remota, in quanto operano in tutte le fasi dell'invocazione remota.

I parametri di una chiamata remota passano, all'atto della chiamata, attraverso la fase di Marshalling e di Unmarshalling con l'ausilio di una coppia di veloci routines di serializzazione / deserializzazione. Questa coppia rappresenta, una volta generata, la "bufferizzazione" della fase di introspezione della classe. Il primo vincolo è di implementare queste routines, come metodi, all'interno della relativa classe, in modo da accedere ai suoi campi protetti e privati.

Sorge il problema, peró, di come determinare se una classe è argomento di una chiamata remota. Ricordo che, per il discorso sull'ereditarietà, non basterebbe, se mai fosse possibile, analizzare le classi che implementano i servizi remoti. In generale bisogna decidere *come* determinare se una classe è reattiva, e *quando* fare la generazione della classe attiva. Partiamo dal discorso sul *quando* per poi arrivare al *come*.

Vi sono quattro momenti utili, nel ciclo di vita del software, per generare una classe attiva: (1) il programmatore all'atto della definizione della classe (quindi

in fase di editing), (2) da un post-compilatore (quindi in fase di compilazione), (3) al caricamento della classe (cioé in fase di linking, che in Java è svolto in parte dal ClassLoader a run-time), (4) al momento della chiamata remota (a run-time). L'ordine nell'elenco rispecchia il ciclo di vita del software.

In primo luogo bisogna dire che ritardare questa operazione risulta vantaggioso, perchè si evita di fare un lavoro inutile, nel senso di non strettamente necessario, per l'esecuzione del programma. Ad esempio si potrebbero aggiungere dei serializzatori ad una classe, solo perché è reattiva, ma senza che questa diventi attiva per la comunicazione.

Nella nostra analisi partiamo dal primo caso, che è completamente a carico del programmatore: egli prende il compito di sapere quali classi saranno oggetto di una chiamata remota, e di aggiungere a mano i serializzatori. Si osservi che questo, insieme all'ultimo caso, sono le uniche eventualità che non necessitano di una interfaccia `Transportable`. Nelle altre infatti c'è bisogno di un "segnaposto" per indicare che una classe può essere usata in una chiamata remota.

Nel secondo caso, in fase di compilazione, un post-compilatore potrebbe verificare se una classe implementa `Transportable` e generare la relativa classe `FastTransportable`. Ma implicherebbe avere a disposizione preventivamente tutte le classi necessarie per l'esecuzione del programma, ed allungare i tempi di compilazione. In ogni caso, comunque, si svolgerebbe un lavoro non strettamente

necessario per l'esecuzione dell'applicazione, in quanto non è detto che tutte le classi `Transportable` siano poi istanziate ed effettivamente trasmesse.

Saltiamo il terzo e passiamo direttamente all'ultimo caso. Esso rappresenta l'ideale, perché ci farebbe svolgere il lavoro di modifica di una classe solo se è necessario, cioé all'atto di una chiamata remota quando sono noti i tipi dei suoi parametri. Ma una volta che una classe è stata caricata nella JVM, ed ha subito la fase di linking, essa ha creato un legame strettissmo con le altre classi dell'applicazione, ed è impossibile modificarla, anche solo una per aggiungervi dei metodi. Questo legame è così forte che se carichiamo di nuovo la definizione della classe, anche senza attuare nessuna modifica, questa rappresenta un nuovo tipo, che rende impossibile assegnare una istanza dell'uno all'altra.

A questo punto, l'unico momento che abbiamo a disposizione, partendo dalla destra nel nostro ipotetico asse dei tempi, è il terzo caso: al caricamento della classe nella JVM. La logica da implementatate è la seguente: se una classe implementa l'interfaccia `Transportable` definisce una classe reattiva, e si genera in automatico la relativa classe attiva che implementa l'interfaccia `FastTransportable`. Svolgiamo un lavoro inutile? Puó essere, infatti non è detto che una istanza di quella classe sia effettivamente trasmessa, ma rispetto al caso due (post-compilazione) siamo almeno sicuri che la classe è istanziata.

6.3.5 Classi `Transportable`

Una classe che implementa l'interfaccia `Transportable`, può essere argomento di una chiamata remota sia come parametro, che come valore di ritorno. Il fatto che durante l'esecuzione del programma nessun servizio remoto ne faccia direttamente uso, dichiarando un argomento di quel tipo, non significa nulla. Grazie al dinamismo di Java, nel proseguo dell'esecuzione, vi puó essere un nuovo servizio che la usa direttamente. Ma la classe stessa può essere usata indirettamente se è sottotipo di un parametro formale di un servizio remoto.

La modifica consiste nell'aggiunta dei metodi, costruttori e campi che permettono ad una istanza di classe di essere trasferita da un nodo all'altro. Le modifiche sono le seguenti:

1. Viene aggiunta, tra le interfacce implementate, l'interfaccia `FastTransportable`.

2. Viene aggiunto un costruttore di default con modificatore d'accesso public. Se la classe già contiene un costruttore di default, ma non è public, allora viene modificato solo l'accesso.

3. Viene aggiunto un campo:

   - `public final static long SUID;`

   con il valore con l'algoritmo esposto nel paragrafo 6.4.

```
public class hype.X extends java.lang.Object implements run.Transportable {
    int a;
    int b;
    int ia[];
    public hype.X(int,int,int[]);
    public int get();
    public java.lang.String toString();
    public boolean equals(java.lang.Object);
    static {};
}
```

Listato 6.1: X una classe che implementa Transportable

4. vengono aggiunti i tre metodi:

   - `public void writeObject( Container.ContainerOutputStream cos ) throws SerializationException;`
   - `public void readObject( Container.ContainerInputStream cis ) throws SerializationException;`
   - `public int sizeOf();`

In primo luogo le modifiche vengono effettuate "solo se necessarie". Cioè solo se la classe non contiene già quell'attributo. Questo consente al programmatore di definire proprie versione di questi metodi. Le modifiche sono attuate dai quasi omonimi metodi della classe `RWSGenerator`. Di seguito descriviamo in dettaglio le modifiche attuate. Per aiutare la descrizione mostriamo nel listato 6.1 una classe prima delle modifiche attuate, e nel listato 6.2 dopo.

```java
package hype;

import run.*;
import run.serialization.SizeOf;
import run.serialization.Container;
import run.serialization.SerializationException;

public class X implements FastTransportable {
  int a;
  int b;
  int[] ia;

  public X(int _a, int _b, int[] _ia)
  {   a = _a;    b = _b;    ia = _ia;   }

  public String toString() {}
  public  boolean equals( Object o ) {}

  public X()
  {   super(); }

  public void writeObject( Container.ContainerOutputStream cos )
      throws SerializationException
  {
    cos.writeInt( a );
    cos.writeInt( b );
    cos.writeArray( ia );
    return;
  }

  public void readObject( Container.ContainerInputStream cis )
      throws SerializationException
  {
    a = cis.readInt( );
    b = cis.readInt( );
    ia = (int[]) cis.readArray();
  }

  public int sizeOf()
  {
    return SizeOf.INT+// a
      SizeOf.INT + //b
      SizeOf.array( ia ); //ia
  }

  final public static long SUID = clio.SUID.getSUID( X.class );
}
```

Listato 6.2: X una classe che implementa FastTransportable

Campo SUID

Il campo SUID contiene il valore hash della codifica di una classe. È calcolato in funzione della sua definizione e la rappresenta in maniera univoca. Il valore è calcolato con l'algoritmo esposto nel paragrafo 6.4. Se il campo è gia definito non viene modificato, questo permette di *congelare* la versione di una classe.

Il valore SUID è calcolato in funzione della definizione completa della classe ed anche modifiche minori, che non influenzano i campi ma solo i metodi, influenzano il valore calcolato. In linea di principio questo è un concetto valido. Infatti se cambia la definizione di un metodo è possibile che cambi anche il significato dei campi, ed in ogni caso le due definizioni rappresentano due tipi diversi. Questo rientra nel campo del possibile, ma è piú probabile il contrario: le due classi rappresentino lo stesso tipo. Il programmatore è libero di gestire questa eventualità assegnando staticamente un valore unico al campo SUID.

Costruttore di Default

La necessità di un costruttore di default pubblico nasce dall'osservazione che deserializzare implica istanziare un nuovo oggetto della classe. L'instanziazione viene effettuata esternamente alla classe, da qui la necessità di renderlo pubblico. Deve essere di default in quanto si occuperá la deserializzazione ad inizializzarne i campi. L'unica inizializzazione effettuata da questo costruttore è la chiamata

del costruttore di default della superclasse.

Serializzatore: `writeObject()`

Il processo di serializzazione da noi implementato si basa su una introspezione statica fatta a run-time. Cioè un analisi fatta una sola volta, a tempo di esecuzione, prima che vengano generate istanze dell'oggetto. Questo processo produce il metodo `writeObject()` ed il suo duale `readObject()`. Il metodo `writeObject()` si occupa di serializzare i campi della classe, invocando gli opportuni metodi della classe `Container`[1]. Se un campo è di un tipo primitivo (int, byte, ...) vi sono già codificati i metodi per la loro serializzazione. Se il campo rappresenta un tipo composto, verrà chiamato il metodo `writeObject()`. Che a sua volta chiama il serializzatore della classe. Anche se al momento non è definito, lo sará al tempo di caricamento della classe.

Deserializzatore: `readObject()`

Il processo di deserializzazione da noi implementato si basa su una introspezione statica fatta a run-time. Cioè un analisi fatta una sola volta, a tempo di esecuzione, prima che vengano generate istanze dell'oggetto. Questo processo produce il metodo `readObject()` ed il suo duale `writeObject()`. Il metodo

[1] In realtà i metodi di serializzazione sono stati raggruppati nella sottoclasse `Container.ContainerOutputStream`

`readObject()` si occupa di deserializzare i campi della classe, invocando gli opportuni metodi della classe `Container`[2]. Se un campo è di un tipo primitivo (int, byte, ...) vi sono già codificati i metodi per la loro deserializzazione. Se il campo rappresenta un tipo composto, verrà chiamato il metodo `readObject()`. Che a sua volta chiama il deserializzatore della classe. Anche se al momento non è definito, lo sará al tempo di caricamento della classe.

Dimensione di un Oggetto: `sizeOf()`

Conoscere la dimensione di un oggetto serve durante la serializzazione di un oggetto, per permettere l'allocazione di un frame di byte che lo possa contenere. Questo frame contiene una successione di bytes, che devono essere consecutivi in memoria per essere inviati sul canale di comunicazione.

Durante la fase di deserializzazione ció non è necessario in quanto l'allocazione degli oggetti è fatta nello heap, e non vi sono vincoli sull'ordine di memorizzazione.

Il procedimento di calcolo segue la stessa logica della fase di serializzazione. Viene analizzato ogni campo: se questo è primitivo la sua dimensione è nota e viene sommato al totale, se è un oggetto viene chiamato il relativo metodo `sizeOf()`. A run-time il calcolo è effettuato sommando il totale, già calcolato,

[2] In realtà i metodi di deserializzazione sono stati raggruppati nella sottoclasse `Container.ContainerInputStream`

```
package aegis;

import run.Services;
import run.RemoteException;

public interface Rs extends Services {

  int f1( X x, int[] ia, int i ) throws RemoteException;
  int[] f2( X x, X[] xa, int i ) throws RemoteException;
  X f3( int i ) throws RemoteException;
  Z f4( int n ) throws RemoteException;
  void f5( ) throws RemoteException;
  float f6( X x, long l, String s ) throws RemoteException;
  String f7( X x, int l, String s ) throws RemoteException;
  X f8( long l, String s, X x ) throws RemoteException;

}
```

Listato 6.3: Rs una interfaccia di servizi remoti

dei campi primitivi piú i risultati delle chiamate dei metodi `sizeOf()` sui campi che sono oggetti.

6.3.6 Classi stub

La classe stub ha la funzione di effettuare il Marshalling dei parametri della chiamata e l'Unmarshalling del risultato. La procedura di Marshalling produce un frame di byte, contenente i parametri serializzati. La fase di Unmarshalling produce il risultato della chiamata remota.

La prima operazione è svolta come serializzazione dei parametri della chiamata nell'ordine in cui appaiono.E come deserializzazione del risultato della chiamata remota, per quanto riguarda la seconda. Entrambe si basano sul tipo effettivo dei parametri per l'invocazione degli opportuni metodi della classe `Container`.

Mostriamo nel listato 6.4 lo stub relativo all'interfaccia di servizi remoti del

```
package aegis;

public class Aegis_Stub extends Stub implements Rs // Nota: Rs extends services
{
  public Aegis_Stub( RegistryServicesReference rsr )
  {    super( rsr );  }

  // primo servizio
  public int f1( X x, int[] ia, int i ) throws RemoteException
  {
    try
    {
      int f1_framesize =0;
      f1_framesize = SizeOf.object( (FastTransportable) x );
      f1_framesize += SizeOf.array(ia);
      f1_framesize += SizeOf.INT;

      Container container = new Container( f1_framesize );

      // true indica che serializzeremo anche oggetti quindi creare la reference_table
      Container.ContainerOutputStream cos = container.getContainerOutputStream( true );

      cos.writeObject( (FastTransportable) x );
      cos.writeArray( ia );
      cos.writeInt( i );
      cos.close();

      // migliorare qui per l'individuazione del servizio remoto.
      // firma aggiunta a "mano"
      RemoteServiceReference remote_services_reference =
          registry_services_reference /*campo ereditato*/
            .toRemoteServiceReference( "f1", "(Laegis/X;[II)I" );

      Callgram callgram = new Callgram( Callgram.CALL,
                    remote_services_reference, null, container );

      Callgram rcv_callgram = Manager.execRemoteCall( callgram );
      callgram.free();
      callgram = null;

      Container res = rcv_callgram.getContainer();

      // argomento false: il risultato e' un primitivo
      Container.ContainerInputStream cis = res.getContainerInputStream( false );

      int _f1 = cis.readInt();

      rcv_callgram.free();
      rcv_callgram = null;
      return _f1;
    }
    catch (ExecException exec_exception )
    { throw new RemoteException( exec_exception.getMessage() ); }
    catch (SerializationException serialization_exception)
    { throw new RemoteException(serialization_exception.getMessage() ); }
    catch (TransportException transport_exception)
    { throw new RemoteException( transport_exception.getMessage() ); }
  }

    public int[] f2( X x, X[] xa, int i ) throws RemoteException   { }

    public X f3( int i ) throws RemoteException                    { }

    public Z f4( int i ) throws RemoteException                    { }

    public void f5( ) throws RemoteException                       { }
}
```

Listato 6.4: Aegis_Stub

```java
package aegis;

public class Aegis_Skeleton extends Skeleton
{

  public Aegis_Skeleton( Services _services )
  {
    super( _services );
  }

  public Container invoke_f1__519339342( Container container )
        throws RemoteException, SerializationException
  {
    Container.ContainerInputStream cis = container.getContainerInputStream(true);

    X x = (X) cis.readObject( );
    int[] ia = (int[]) cis.readArray();
    int i = cis.readInt();

    Rs server_ref = (Rs) services;
    int invoke_res = server_ref.f1( x, ia, i ) ;

    Container res = new Container( SizeOf.INT );
    Container.ContainerOutputStream cos = res.getContainerOutputStream( false );
    cos.writeInt( invoke_res );
    cos.close();
    return res;
  }

  public Container invoke_f2_373712653( Container container )
         throws RemoteException, SerializationException  { }

  public Container invoke_f3__1108078295( Container container )
    throws RemoteException, SerializationException       { }

  public Container invoke_f4__1108078233( Container container )
     throws RemoteException, SerializationException      { }

  public Container invoke_f5_39797( Container container )
     throws RemoteException, SerializationException      { }
}
```

Listato 6.5: Aegis_Skeleton

listato 6.3. L'implementazione di questa procedura è nella classe `StubGenerator` mostrata nel listato B.21 dell'appendice B.

6.3.7 Classi skeleton

La classe skeleton ha la funzione di effettuare l'Unmarshalling dei parametri della chiamata e il Marshalling del risultato. La prima operazione viene svolta come deserializzazione, dal frame passato come argomento, dei parametri nell'ordine in cui appaiono nella firma del metodo.E come produzione di un frame di byte con il risultato serializzato, per quanto riguardo la seconda. Entrambe usano il tipo effettivo dei parametri della chiamata per chiamare i metodi di serializzazione / deserializzazione della classe `Container`.

Mostriamo nel listato 6.5 lo skeleton relativo all'interfaccia di servizi remoti del listato 6.3. Si noti il nome di ogni metodo. Per quello che è stato detto nel paragrafo 5.1.1 a pagina 65 il nome esteso del metodo `f1` è `invoke_f1__519339342` con la stringa `_519339342` che rappresenta la codifica hash della firma del metodo "`(Laegis.X;[II)I`". L'implementazione di questa procedura è nella classe `SkeletonGenerator` mostrata nel listato B.22 dell'appendice B

6.4 SUID: Stream Unique IDentifier

Lo Stream Unique Identifier (SUID d'ora in poi) è un intero a 64-bit che permette di identificare in maniera univoca ogni classe. Il valore è calcolato in funzione del nome della classe, ma anche della sua definizione. E permette cosí di distinguere tra due versioni della stessa classe.

Il valore è stato calcolato con la procedura definita in [SUN98], cioé come firma di uno stream di bytes che riflette la definizione della classe. La funzione Secure Hash Algorithm (SHA-1) del National Institute of Standards and Technology (NIST) è usato per computare una firma a 160-bit dello stream. I primi otto bytes della firma sono usati per ottenere i 64-bit necessari.

I valori che sono posti nello stream sono i seguenti:

1. Nome della classe, nella codifica UTF
2. Modificatori della classe, scritti come interi a 32-bit.
3. Il nome di ciascuno interfaccia implementata, ordinati e scritti usando la codifica UTF.
4. Per ciascun campo della classe (eccetto quelli statici, privati e transienti) ordinati per nome:
   (a) Nome del campo, nella codifica UTF.

   (b) Modificatore della classe, codificato come intero a 32-bit.

   (c) Descrittore (tipo) del campo, nella codifica UTF.

5. Se è definito un inizializzatore di classe, vengono scritti:

   (a) Nome del metodo: "⟨clinit⟩", nella codifica UTF.

   (b) Modificatore del motodo (static) scritto come intero a 32-bit.

   (c) Descrittore (firma) del metodo: "()V", nella codifica UTF.

6. Per ciascun costruttore, ordinati per firma:

   (a) Il nome del metodo "⟨init⟩", nella codifica UTF.

   (b) Modificatori del metodo, scritti come intero a 32-bit.

   (c) Descrittore (firma) del metodo, nella codifica UTF.

7. Per ciascun metodo non privato, ordinati per nome e firma:

   (a) Il nome del metodo, codificato in UTF.

   (b) I modificatori del metodo, scritti come intero a 32-bit.

   (c) Descrittore del metodo (firma), nella codifica UTF.

8. L'algoritmo SHA-1 è eseguito sullo stream. Produce 20 bytes.

9. Il valore, di tipo intero lungo, è ricavato dai primi 8 bytes della firma hash.

Il valore è calcolato dal metodo `computeSUID()` della classe `SUID`. C'è da osservare che il valore calcolato dal motodo `getSerialVersionUID()` della classe `ObjectStreamClass` è diverso. Nonostante entrambi siano calcolati con lo stesso algoritmo. Infatti i due metodi usano diverse librerie per l'introspezione della classe. La prima usa BCEL, la seconda l'API reflection della SUN. BCEL è orientata alla classe cosí come è definita in un file di definizione, quindi prima del linking. L'API reflection è orientata alla classe dopo il linking. La principale differenza di questo diverso approccio è che i campi ereditati dalla superclasse non sono tenuti in conto da BCEL, mentre lo sono nell'API reflection. Quindi i due algoritmi calcolano valori diversi, perchè operano su dati diversi.

6.5 Scenario d'Uso

Mostreremo adesso due scenari d'uso del modulo CLIO: la generazione di una classe stub, e la generazione di una classe `FastTransportable`. Ci aiuteremo con due Collaboration Diagram di UML.

6.5.1 Generazione di una Classe stub

Il Collaboration Diagram di figura 6.3 ci mostra la fase di generazione di una classe stub.

Descrivereremo passo-passo le operazioni svolte.

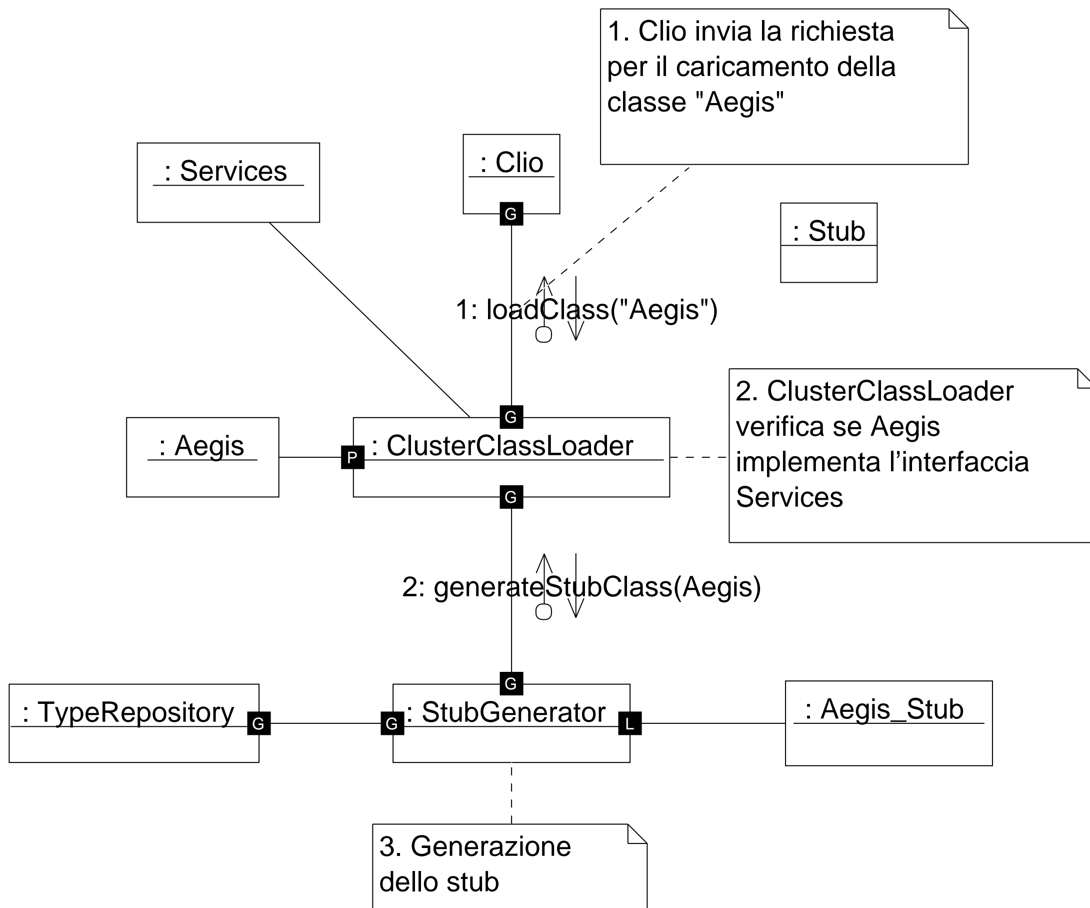

Figura 6.3: Generazione di una classe stub (UML).

1. L'applicazione richiede una classe stub di nome: `Aegis_Stub`.

2. ClusterClassLoader verifica: (1) se la classe è disponibile nel repository, (2) se è caricabile dal CLASSPATH. Se è giá disponibile verifica se necessita di essere aggiornata.

3. Nel caso debba essere generata, viene inviato il messaggio `generateStubClass()`

alla classe `StubGenerator`.

4. La classe viene memorizzata nel repository e restituita all'applicazione

6.5.2 Generazione di una Classe `FastTransportable`

Il collaboration diagram di figura 6.4 ci mostra la fase di generazione di una classe `FastTransportable`

Descrivereremo passo-passo le operazioni svolte.

1. L'applicazione richiede una classe.

2. La richiesta arriva a ClusterClassLoader che la carica, con il metodo `loadClass()`.

3. Si controlla se la classe implementa il tipo `Transportable`, ed in caso affermativo viene inviato il messaggio `modify()` alla classe `RWSGenerator`.

4. La classe viene modificata, alla fine della modifica viene computato il valore SUID.

5. La class cosí modificata viene memorizzata nel repository e restituita all'applicazione.

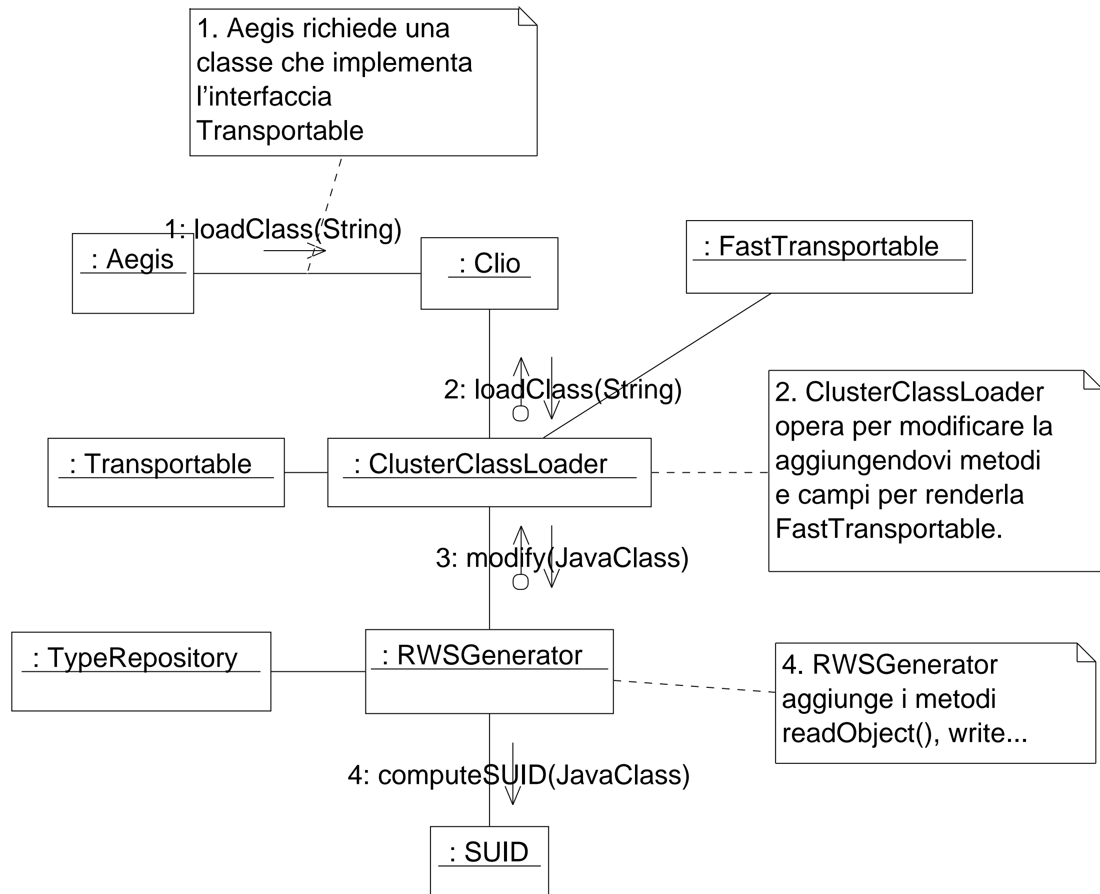

Figura 6.4: Generazione di una classe `FastTransportable` (UML)

Capitolo 7

Esempio di Applicazione

Mostreremo in questo capitolo un utilizzo dell'architettura da noi presentata. Descriveremo: (1) come definire una interfaccia di servizi remoti, (2) come implementare e (3) registrare un pool di servizi, ed infine (4) mostreremo le operazioni che deve svolgere un'applicazione Client per richiedere un servizio.

7.1 Definizione di un Pool di Servizi

Il primo passo è definire una interfaccia di servizi remoti. Una interfaccia definisce un pool di servizi remoti se estende `Services`. L'unico vincolo dell'interfaccia è di avere un modificatore di accesso pubblico. Ogni metodo dichiarato al suo interno rappresenta l'implementazione di un servizio, ed ha come unico vincolo quello di dichiare nella lista delle eccezioni lanciabili l'eccezione `run.RemoteExeception`.

```
package demo;

import run.Services;
import run.RemoteException;

public interface Aegis extends Services
{
  int f1( int[] ia, int i ) throws RemoteException;
  X f2( X[] xa, int i ) throws RemoteException;
}
```

Listato 7.1: Definizione di una interfaccia di servizi remoti

Mostriamo nel listato 7.1 un esempio di definizione di interfaccia di servizi remoti. Definiamo una interfaccia di nome `Aegis` che definisce un pool di due servizi, il numero ed il tipo dei parametri per adesso non è importante, si noti per adesso solo l'uso di un tipo `X` come parametro del metodo `f2()`.

7.2 Implementazione di un Pool di Servizi

Implementare un pool di servizi remoti significa definire una classe che implementa l'interfaccia di definizione dei servizi remoti.

Nel nostro caso definiamo una classe `AegisImpl` che implementa l'interfaccia `Aegis`, come mostrato nel listato 7.2. I metodi `f1(), f2()` sono pubblici e definiscono nella lista di eccezioni lanciabili `run.RemoteException` così come richiede l'interfaccia `Aegis`.

Le operazioni svolte dai due metodi sono simili e piuttosto semplici. I metodi restituiscono l'elemento di indice `i` dell'array passato come primo parametro: `ia` nel caso di `f1()` ed `xa` nel caso di `f2()`. Entrambi i metodi generano una eccezione

```
package demo;

import rio.registry.Registry;
import rio.registry.RegistryException;

import run.RemoteException;
import run.Services;
import run.Utility;

import rio.registry.*;

import run.transport.*;

public class AegisImpl implements Aegis
{
  public AegisImpl()  {  }

  public int f1( int[] ia, int i ) throws RemoteException
  {
    Debug.println( "f1 invoked, i:" + i );

    if (i<0 || i>=ia.length)
      throw new RemoteException("f1(): Indice fuori dai limiti consentiti");
    else
      return ia[i];
  }

  public X f2( X[] xa, int i ) throws RemoteException
  {
    Debug.println( "f2 invoked, i: " + i  );

    if (i<0 || i>=X.length)
      throw new RemoteException("f2(): Indice fuori dai limiti consentiti");
    else
      return X[i];
  }
}
```

Listato 7.2: Implementazione di una interfaccia di servizi remoti

```
package demo;

public class X implements Transportable {
    int a;
    int b;
    int ia[];

    public X(int _a,int _b ,int[] _ia)
    {
      a = _a; b = _b; ia = _a;
    }

    public int get() { return a; }
}
```

Listato 7.3: Implementazione di un argomento di una servizio remoto

remota se l'indice eccede la dimensione dell'array oppure è negativo.

Passiamo adesso alla definizione degli argomenti dei metodi. Il metodo `f1()` usa come argomenti solo tipi primitivi. Il metodo `f2()` dichiara di usare come parametro un array i cui componenti sono istanze della classe `X` e come valore di ritorno una istanza sempre della classe `X`. Un oggetto per poter essere trasmesso da una nodo all'altra deve essere istanza di una classe che implementare l'interfaccia `Transportable`. Tale interfaccia non pone obblighi per il programmatore. Rappresenta solo un segnaposto per CLIO per informarlo di aggiungervi i serializzatori, cosí come descritto nel capitolo 6. L'uso della classe `X` nel metodo `f2()` ha lo scopo di mostrare che istanze di `X` sono correttamente trasferite da un nodo all'altro in entrambi i versi della comunicazione. La definizione della classe `X` è nel listato 7.3.

Si noti che la classe ha modificatore di accesso `public` in quanto verrà istanziata da un package esterno a quello a cui appartiene la classe. In questo specifico

caso dal modulo CLIO durante la fase di deserializzazione. Ed l'unico vincolo per le classi che definiscono i tipi degli argomenti di una servizio remoto.

7.3 Registrazione di un Pool di Servizi

In primo luogo bisogna istanziare un pool di servizi remoti. La registrazione è svolta poi inviando il riferimento dell'istanza del pool allo host che ospita il servizio di registry. L'indirizzo dello host è memorizzato in un file di configurazione detto file delle proprietà.

Queste operazioni sono state implementate nella classe `AegisServer` mostrata nel listato 7.4.

7.4 Implementazione di una Applicazione Client

L'applicazione Client per richiedere un servizio remoto deve svolgere due operazioni: (1) effettuare il lookup del servizio inviando una richiesta al ClusterRegistry, e sul riferimento ricevuto (2) effettuare l'invocazione remota del servizio.

Descriveremo queste due fasi aiutandoci la classe `Client` mostrata nel listato 7.5.

```
package demo;

import java.net.*;

import rio.registry.Registry;
import rio.registry.RegistryException;

import run.RemoteException;
import run.Services;
import run.Utility;

import rio.registry.*;

import run.transport.*;

public class AegisServer
{
  public static void main(String[] args)
      throws RegistryException, RemoteException, UnknownHostException
  {
    NetAddress registry_address = null;
    AegisImpl server = new AegisImpl();
    String prop_registry_address = Utility.getStringProperty(
                                      "rio.registry.address", "10.0.2.1"
                                    );
    int prop_registry_port = Utility.getIntProperty(
                                      "rio.registry.port", 2002
                              );

    registry_address = new IPAddress(
        InetAddress.getByName(prop_registry_address),
        prop_registry_port
    );

    Registry.register( registry_address, "aegis", server );
  }
}
```

Listato 7.4: Istanziazione di un pool di servizi remoti

```
package demo;

import java.net.*;

import rio.registry.Registry;
import rio.registry.RegistryException;

import run.RemoteException;
import run.transport.IPAddress;

import run.session.SessionTable;
import run.Utility;

public class Client
{
  public static void main(String[] args)
          throws RegistryException, RemoteException, UnknownHostException
  {
    String prop_registry_address =
         run.Utility.getStringProperty( "rio.registry.address", "10.0.2.1" );
    int prop_registry_port =
         run.Utility.getIntProperty( "rio.registry.port", 2001 );

    IPAddress registry_address = null;

    registry_address = new IPAddress(
          InetAddress.getByName(prop_registry_address),
          prop_registry_port );


    Aegis remote_services = (Aegis) Registry.lookup( registry_address, "aegis" );

    int[] ia = new int[] { 6, 7, 8 };
    int i_res = remote_services.f1( ia, 0 );
    System.out..println( "f1(): "+i_res );

    X[] xa = new X[] {
                      new X( 1, 2, new int[] { 3, 4, 5} ),
                      new X( 6, 7, new int[] { 8, 9 } ),
                      new X( 10, 11, new int[] {13, 14, 15, 16 } )
                    };

    X x_res = f2( xa, 1 );
    System.out.println( ``f2(): ``+ x_res );
  }
}
```

Listato 7.5: Invocazione di un servizio remoto

7.4.1 Lookup di un Pool di Servizi

L'operazione di "lookup" è svolta invocando l'omonimo servizio dal pool "registry". L'intera operazione è trasparente grazie alla classe `Registry` ed al suo metodo `lookup()`. L'unica "difficoltà" è specificare l'indirizzo dello host che ospita il ClusterRegistry, questo dato è stato parametrizzato nel file delle proprietà a cui si accede mediante la facility: `getIntProperty()` della classe `Utility`. Per una descrizione del file delle proprietà rimandiamo all'appendice A.

7.4.2 Invocazione di un Servizio Remoto

Il risultato dell'operazione di `lookup()` è un riferimento Java ad un oggetto che implementa l'interfaccia di servizi remoti. Su questo riferimento verranno invocati i metodi per richiedere in maniera trasparente i servizi remoti.

7.5 Esecuzione dell'esempio

Come prima operazione bisogna configurare il file delle proprietà con gli indirizzi dei nodi che ospitano l'applicazione server, client ed il registry. Se non si desidera modificare il file è possibile codificare nel sorgente dell'applicazione tali informazioni. Per una descrizione del file delle proprietà rimandiamo all'appendice A.

I passi da eseguire sono poi i seguenti:

1. Sul nodo che ospiterà il registry avviare il ClusterRegistry con la riga di comando:

   ```
   # java rio.registry.Main
   ```

2. Sul nodo che ospiterà il pool di servizi, avviare il Server con la riga di comando:

   ```
   # java clio.Main demo.AegisServer
   ```

3. Infine sul nodo che ospiterà l'applicazione Client, eseguire il comando:

   ```
   # java clio.Main demo.Client
   ```

Il formato inusuale dei comandi per avviare l'applicazione Client e Server ha lo scopo di usare il modulo CLIO per effettuare la generazione on-the-fly della classi necessarie all'architettura di comunicazione. Per ulteriori spiegazioni rimandiamo al capitolo 6 sul modulo CLIO.

Capitolo 8

Futuri sviluppi

Ulteriori passi nello sviluppo dell'architettura si muovono in due direzioni. Da un lato verso il miglioramento delle prestazioni delle funzionalità già a disposizione, dall'altro con l'aggiunta di nuove funzionalità.

Migliorare sotto l'aspetto prestazionale, significa studiare due aspetti: il protocollo di comunicazione, e la fase di serializzazione.

Sarebbe interessante studiare i risultati ottenibili ottimizzando l'attuale protocollo di comunicazione basato sul GO-BACK-N. Ma anche esplorare altri protocolli non basati sullo Sliding Window ma sul Window Flow ([ASU86]).

Per quanto riguarda la fase di serializzazione sarebbe interessante studiare l'ottimizzazione dei serializzatori.

Funzionalità che adesso mancano ma che sarebbero di notevole interesse, sono

l'aggiunta del supporto a nuove librerie di comununicazione. Quindi non solo l'uso del protocollo UDP ma, ad esempio, il protocollo VIA ([VIA97]). In quest'ottica il sistema è già pronto. Con scelte come l'astrazione di un indirizzo di un nodo (`NetAddress`). E la scelta di serializzare tutti i dati da trasmettere, per produrre un frame di byte consecutivi in memoria. Questo rende la fase di serializzazione indipendente dal trasporto vero e proprio, ma soprattutto soddisfa le necessità del protocollo VIA che necessita di un buffer consecutivo in memoria per trasmettere il frame.

Infine, sarebbe utile l'implementazione di tecniche di time-out e ritrasmissione. Sono semplici da implementare e, in situazioni di possibile fallimento di nodi, sono molto utili per aumentare l'affidabilità del sistema.

Capitolo 9

Conclusioni

Ci eravamo posti l'obiettivo di realizzare una architettura di invocazione di metodi remoti che permettesse di raggiungere i seguenti risultati:

- Presentasse innovazioni rispetto al passato.
- Fosse orientata ad un cluster di computer.
- Fosse semplice da utilizzare per l'utente.
- Fosse efficiente.

Ed abbiamo raggiunto i seguenti obiettivi in quanto:

- È innovativa, in quanto introduce il concetto di modifica a run-time di una classe allo scopo di aggiungere i serializzatori.

- È orientata ad un sistema cluster, in quanto: (1) il registry memorizza l'associazione tra servizi e host per l'intera rete, (2) il protocollo di comunicazione si basa sulla relativa affidabilità delle comunicazioni locali, e (3) il protocollo di serializzazione si basa sul presupposto che le definizioni delle classi siano tutte disponibili, e nella corretta versione, sia al client che al server.

- È semplice, in quanto l'utente non è costretto ad allungare il ciclo di vita del software con una fase di post-compilazione.

- È efficiente, in quanto secondo test preliminare ClusterRMI risulta essere piú efficiente di RMI nella fase di introspezione, e lo eguaglia nella fase di serializzazione e trasmissione. Benchmark piú precisi sono in fase di allestimento.

Appendice A

Proprietà di Sistema

A.1 File delle Proprietà

Il file delle proprietà ha la funzione di poter configurare alcuni parametri di ClusterRMI. Di questi, alcuni regolano meccanismi interni del sistema, altri sono di maggior interesse dell'utente in quanto specificano gli indirizzi di rete dell'applicazione Client, Server e del ClusterRegistry.

Un esempio di file delle proprietà è mostrato nel listato A.1.

Il file è posto nella directory di lavoro dell'applicazione. Quindi, in presenza di uno scenario d'uso client/server, di questo file vanno create due copie. Ognuno di essi va modificato per configurare un diverso indirizzo di rete da usare per il trasporto, mediante le proprietà run.transport.*. Infine per entrambi va configu-

```
run.session.sessionsendpool.poolsize=4
run.session.sessionreceivepool.poolsize=4
run.session.sessionsend.windowsize=16
run.exec.callthread.poolsize=2

run.transport.udptransport.ethernet.windowsize=16
run.transport.udptransport.loopback.windowsize=16
run.transport.udptransport.myrinet.windowsize=16

run.transport.udptransport.loopback.port=0
run.transport.udptransport.loopback.address=127.0.0.1
run.transport.udptransport.ethernet.port=0
run.transport.udptransport.ethernet.address=10.0.2.6
run.transport.udptransport.myrinet.port=0
run.transport.udptransport.myrinet.address=10.1.4.6

run.transport.packetpool.poolsize=16

rio.registry.address=10.1.4.1
rio.registry.port=2013
```

Listato A.1: File delle Proprietà di Sistema

rato l'indirizzo di rete del ClusterRegistry, mediante le proprietà rio.registry.*. Si noti che, se non in casi specifici, si suppone di usare lo stesso valore per entrambi.

A.2 Elenco delle Proprietà di Sistema

Nella tabella A.1 elenchiamo tutte le proprietà di sistema utilizzate da Cluster-RMI.

Nome	Usato da	Default	Descrizione	Nota
run. session. sessionsendpool. poolsize	run. session. SessionSendPool	4	Dimensione del pool di SessionSend	
run. session. sessionreceivepool. poolsize	run. session. SessionReceivePool	4	Dimensione del pool di SessionReceive	
run. session. sessionsend. windowsize	run. session. SessionSend	16	Dimensione della finestra di pacchetti	Questo valore ha precedenza sull'equivalente al livello trasporto
run. exec. callthread. poolsize	run. exec. CallThreadPool	4	Dimensione del pool di CallThread	
run. transport. udptransport. ethernet. windowsize	run. transport. TransportTable	16	Dimensione della finestra di pacchetti per il trasporto UDP associato all'interfaccia Ethernet	Il valore di default è pari alla massima dimensione del buffer dei Socket di Berkeley
run. transport. udptransport. loopback. windowsize	run. transport. TransportTable	16	Dimensione della finestra di pacchetti per il trasporto UDP dell'interfaccia di loopback	Il valore di default è pari alla massima dimensione del buffer dei Socket di Berkeley
run. transport. udptransport. myrinet. windowsize	run. transport. TransportTable	16	Dimensione della finestra di pacchetti per il trasporto UDP associato all'interfaccia Myrinet	Il valore di default è pari alla massima dimensione del buffer dei Socket di Berkeley
run. transport. udptransport. loopback. port	run. transport. TransportTable	0	Numero di porta associato all'interfaccia di loopback	Il valore 0 indica al sistema di scegliere il primo disponibile
run. transport. udptransport. loopback. address	run. transport. TransportTable	127.0.0.1	Indirizzo di rete associato all'interfaccia di loopback	
run. transport. udptransport. ethernet. port	run. transport. TransportTable	0	Numero di porta associato all'interfaccia Ethernet	Il valore 0 indica al sistema di scegliere il primo disponibile
run. transport. udptransport. ethernet. address	run. transport. TransportTable	10.0.2.6	Indirizzo di rete associato all'interfaccia Ethernet	Il valore di default è relativo alla rete del laboratorio
run. transport. udptransport. myrinet. port	run. transport. TransportTable	0	Numero di porta associato all'interfaccia Myrinet	Il valore 0 indica al sistema di scegliere il primo disponibile
run. transport. udptransport. myrinet. address	run. transport. TransportTable	10.1.4.6	Indirizzo di rete associato all'interfaccia Myrinet	Il valore di default è relativo alla rete del laboratorio.
run. transport. packetpool. poolsize	run. transport. Transport	16	Dimensione della finestra di pacchetti per il trasporto UDP associato all'interfaccia Myrinet	Il valore di default è pari alla massima dimensione del buffer dei Socket di Berkeley
rio. registry. address	Applicazione Client/Server	10.1.4.1	Indirizzo dell'host che ospita il Registry	Il valore di default è relativo alla rete del laboratorio.
rio. registry. port	Applicazione Client/Server	2013	Numero di porta associato al Registry	

Tabella A.1: Elenco delle proprietà di sistema

Appendice B

Listati

B.1 Package: run.transport

Listato B.1: Packet

```
Compiled from Packet.java
public class run.transport.Packet extends java.lang.Object {
    protected static final int START_OF_OPCODE;
    protected static final int START_OF_SESSION;
    protected static final int START_OF_TOTAL_NUMBER;
    protected static final int START_OF_PACKET_NUMBER;
    protected static final int START_OF_DATA;
    protected static final int START_OF_CALLGRAM_SIZE;
    protected static final int START_OF_SERVICES_NAME;
    protected static final int START_OF_RETURN_VALUE;
    protected static final int START_OF_REF_SESSION_ID;
    protected static final int SIZE_OF_INIT_HEADER;
    protected static final int SIZE_OF_RET_HEADER;
    protected static final int SIZE_OF_DATA_HEADER;
    protected static final int SIZE_OF_DISC;
    protected static final int SIZE_OF_ACKI;
    protected static final int SIZE_OF_ACKD;
    protected static final int SIZE_OF_RN;
    protected byte data[];
    protected int packet_size;
    protected int max_data_size;
    protected run.transport.NetAddress remote_address;
    byte opcode;
    int session_id;
    int total_number;
    int number;
    int data_size;
    int callgram_size;
    java.lang.String services_name;
```

```
    java.lang.String service_name;
    java.lang.String service_signature;
    int return_value;
    int ref_session_id;
    run.transport.PacketPool packet_pool;
    run.transport.Transport transport;
    public run.transport.Packet(run.transport.PacketPool,int);
    public run.transport.Packet(int);
    public run.transport.Packet();
    public void free();
    private final run.transport.Packet set(byte, run.transport.NetAddress, int, int, int, java.l
    public final void setOpcode(byte);
    public final run.transport.Packet setInit(run.transport.NetAddress, int, int, int, java.lang
    public final run.transport.Packet setRet(run.transport.NetAddress, int, int, int, int, int);
    public final run.transport.Packet setData(run.transport.NetAddress, int, int, int, byte[], i
    public final run.transport.Packet setDisc(run.transport.NetAddress, int);
    public final run.transport.Packet setAckI(run.transport.NetAddress, int);
    public final run.transport.Packet setAckD(run.transport.NetAddress, int);
    public final run.transport.Packet setAckR(run.transport.NetAddress, int);
    public final run.transport.Packet setRN(run.transport.NetAddress, int, int, int);
    public final run.transport.Packet setNop(run.transport.NetAddress);
    final void setTransport(run.transport.Transport);
    public final run.transport.PacketPool getPacketPool();
    public final run.transport.Transport getTransport();
    public final byte getOpcode();
    public final int getSessionId();
    public final int getTotalNumber();
    public final int getRN();
    public final int getSN();
    public final int getPacketNumber();
    public final byte getData()[];
    public final int getCallgramSize();
    public final java.lang.String getServicesName();
    public final java.lang.String getServiceName();
    public final java.lang.String getServiceSignature();
    public final run.transport.NetAddress getRemoteAddress();
    public final int getMaxDataSize();
    public final int getDataSize();
    public final int getPacketSize();
    public final int getReturnValue();
    public final int getRefSessionId();
    public boolean equals(java.lang.Object);
    public java.lang.String toString();
    public run.transport.Packet assign(run.transport.Packet);
    public static void main(java.lang.String[]) throws run.serialization.SerializationException;
    private boolean selftest() throws run.serialization.SerializationException;
    private boolean selftest_init();
    private boolean selftest_data() throws run.serialization.SerializationException;
    private boolean selftest_rn();
    private boolean selftest_ret();
    static {};
}
```

Listato B.2: Transport

```
Compiled from Transport.java
public abstract class run.transport.Transport extends java.lang.Thread {
    public static final int DEFAULT_WINDOW_SIZE;
    protected byte type_of_transport;
    protected run.transport.NetAddress local_address;
    protected int packet_size;
```

```
    protected int window_size;
    protected run.transport.PacketPool packet_pool;
    public run.transport.Transport(byte,run.transport.NetAddress,int,int);
    public final int getWindowSize();
    public final run.transport.NetAddress getLocalAddress();
    public final byte getType();
    public final int getPacketSize();
    public final int getMaxDataSize();
    public abstract void send(run.transport.Packet);
    public abstract void run();
    public java.lang.String toString();
    static {};
}
```

Listato B.3: NetAddress

```
Compiled from NetAddress.java
public abstract class run.transport.NetAddress extends java.lang.Object implements run.FastTrans
    public static final byte IP_ADDRESS;
    public static final byte VIA_ADDRESS;
    public static final long SUID;
    public run.transport.NetAddress();
    public abstract byte getType();
    public abstract byte getByteArray()[];
    public boolean equals(run.transport.NetAddress);
    public abstract boolean isReachable(run.transport.NetAddress);
    public abstract void writeObject(run.serialization.Container$ContainerOutputStream) throws r
    public abstract void readObject(run.serialization.Container$ContainerInputStream) throws run
    public abstract int sizeOf();
    static {};
}
```

Listato B.4: UDPTransport

```
Compiled from UDPTransport.java
public final class run.transport.UDPTransport extends run.transport.Transport {
    public static final int ETHERNET_PACKET_SIZE;
    public static final int ETHERNET_WINDOW_SIZE;
    public static final int ETHERNET_PORT;
    public static final int LOOPBACK_PACKET_SIZE;
    public static final int LOOPBACK_WINDOW_SIZE;
    public static final int LOOPBACK_PORT;
    public static final int MYRINET_PACKET_SIZE;
    public static final int MYRINET_WINDOW_SIZE;
    public static final int MYRINET_PORT;
    private static int TIME_OUT;
    protected static int SOCKET_BUFFER_SIZE;
    protected static int DATAGRAM_BUFFER_SIZE;
    protected java.util.LinkedList FIFO_out;
    protected java.net.DatagramSocket socket;
    protected java.lang.ThreadGroup fifo_threadg;
    protected java.lang.Thread fifo_in_thread;
    protected java.lang.Thread fifo_out_thread;
    private int debug_total_packet_sent;
    private int debug_total_packet_received;
    public run.transport.UDPTransport(run.transport.IPAddress,int,int,int) throws run.transport.
    public void send(run.transport.Packet);
    public void setFromDatagram(java.net.DatagramPacket, run.transport.Packet, run.serialization
    public void setToDatagram(java.net.DatagramPacket, run.transport.Packet, run.serialization.C
    public void run();
    public java.lang.String toString();
```

```
    static int access$0(run.transport.UDPTransport);
    static int access$2(run.transport.UDPTransport);
    static int access$1();
    static {};
    private class run.transport.UDPTransport. FIFOOutThread extends java.lang.Thread
    /* ACC_SUPER bit NOT set */
{
        private final run.transport.UDPTransport this$0;
        run.transport.UDPTransport.FIFOOutThread(run.transport.UDPTransport,java.lang.ThreadGrou
        public void run();
    }
    private class run.transport.UDPTransport. FIFOInThread extends java.lang.Thread
    /* ACC_SUPER bit NOT set */
{
        private final run.transport.UDPTransport this$0;
        run.transport.UDPTransport.FIFOInThread(run.transport.UDPTransport,java.lang.ThreadGroup
        public void run();
    }
}
```

B.2 Package: run.session

```
Compiled from Callgram.java
public class run.session.Callgram extends java.lang.Object {
    public static final byte NOP;
    public static final byte CALL;
    public static final byte RETURN;
    public static final byte REMOTE_EXCEPTION;
    public static final byte DGC;
    public static final byte ERROR;
    protected byte operation;
    protected int return_value;
    protected boolean little_endian;
    private run.reference.RemoteServiceReference service_reference;
    protected run.serialization.Container container;
    protected run.session.Session session;
    protected run.session.CallgramListener listener;
    public static final boolean isLittleEndian();
    public run.session.Callgram(byte,run.reference.RemoteServiceReference,run.session.CallgramL
    public run.session.Callgram(byte,int,run.reference.RemoteServiceReference,run.session.Sessio
    public run.session.Callgram(byte,int,run.session.Session,run.session.CallgramListener,run.se
    public final void freeSession();
    public void free();
    public java.lang.String toString();
    public boolean equals(java.lang.Object);
    public final byte getOperation();
    public final run.transport.NetAddress getRemoteAddress();
    public final run.reference.RemoteServiceReference getRemoteServiceReference();
    public final void setRemoteServiceReference(run.reference.RemoteServiceReference);
    public final void setOperation(byte);
    public final void setListener(run.session.CallgramListener);
    public final run.session.CallgramListener getListener();
    public final run.serialization.Container getContainer();
    public final run.session.Session getSession();
    public final void setSession(run.session.Session);
    public final int getReturnValue();
    static {};
}
```

Listato B.5: Callgram

Listato B.6: Session

```
Compiled from Session.java
abstract class run.session.Session extends java.lang.Thread {
    protected static final int TIME_OUT;
    protected static final run.transport.Packet nop_packet;
    protected run.transport.Transport transport;
    protected int session_id;
    protected int ref_session_id;
    protected int packet_total_number;
    protected run.session.Callgram callgram;
    protected int callgram_size;
    protected byte callgram_buffer[];
    protected java.util.ArrayList packet_list;
    protected run.transport.NetAddress remote_address;
    protected run.session.CallgramListener listener;
    protected int return_value;
    protected byte operation;
    volatile byte status;
    run.exec.ThreadPool session_pool;
```

```
    protected run.session.Session(java.lang.ThreadGroup,run.exec.ThreadPool,java.lang.String);
    void initSession(int, int, run.transport.Transport, run.transport.NetAddress, run.session.Ca
    protected synchronized int rand(int, int);
    public abstract void run();
    protected final run.transport.NetAddress getRemoteAddress();
    protected final void setRemoteAddress(run.transport.NetAddress);
    protected final void notifyListener();
    public final run.session.CallgramListener getListener();
    public final int getSessionId();
    public final int getRefSessionId();
    public final run.transport.Transport getTransport();
    protected final void setListener(run.session.CallgramListener);
    abstract run.transport.Packet receive(run.transport.Packet);
    abstract run.transport.Packet retrasmit(run.transport.Packet);
    protected abstract run.transport.Packet init(run.transport.Packet);
    protected abstract run.transport.Packet acki(run.transport.Packet);
    protected abstract run.transport.Packet disc(run.transport.Packet);
    protected abstract run.transport.Packet ackd(run.transport.Packet);
    protected abstract run.transport.Packet data(run.transport.Packet, int);
    protected abstract run.transport.Packet rn(run.transport.Packet);
    protected abstract run.transport.Packet ret(run.transport.Packet);
    protected abstract run.transport.Packet ackr(run.transport.Packet);
    public java.lang.String toString();
    static {};
}
```

Listato B.7: SessionSend

```
Compiled from SessionSend.java
public final class run.session.SessionSend extends run.session.Session {
    protected int window_size;
    protected run.transport.PacketPool window_pool;
    protected int rn;
    protected int sn_min;
    protected int sn_max;
    private int debug_roundtripdelay;
    private int debug_sleeptime;
    private int debug_sleep_counter;
    private int debug_send_counter;
    private int debug_sendtime;
    private int debug_sent_packet_counter;
    public run.session.SessionSend(java.lang.ThreadGroup,run.session.SessionSendPool);
    synchronized void initSession(int, int, run.transport.Transport, run.transport.NetAddress, r
    synchronized void initSession(int, int, run.transport.Transport, run.transport.NetAddress, r
    public synchronized void send(run.session.Callgram);
    protected final int set_sn_max();
    synchronized run.transport.Packet receive(run.transport.Packet);
    run.transport.Packet retrasmit(run.transport.Packet);
    public void run();
    final run.transport.Packet send_window(run.transport.Packet);
    final int send_window();
    protected run.transport.Packet init(run.transport.Packet);
    protected run.transport.Packet acki(run.transport.Packet);
    protected run.transport.Packet ret(run.transport.Packet);
    protected run.transport.Packet ackr(run.transport.Packet);
    protected run.transport.Packet disc(run.transport.Packet);
    protected run.transport.Packet ackd(run.transport.Packet);
    protected run.transport.Packet data(run.transport.Packet, int);
    protected run.transport.Packet rn(run.transport.Packet);
    public java.lang.String toString();
}
```

Listato B.8: SessionReceive

```
Compiled from SessionReceive.java
public final class run.session.SessionReceive extends run.session.Session {
    protected byte callgram_buffer[];
    protected java.lang.String services_name;
    protected java.lang.String service_name;
    protected java.lang.String service_signature;
    protected int rn;
    protected run.transport.PacketOne ctrl_packet;
    protected java.util.BitSet packets_map;
    private int debug_datapacket_dropped;
    private int debug_received;
    public run.session.SessionReceive(java.lang.ThreadGroup,run.session.SessionReceivePool);
    synchronized void initSession(int, int, run.transport.Transport, run.transport.NetAddress, r
    synchronized run.transport.Packet receive(run.transport.Packet);
    run.transport.Packet retrasmit(run.transport.Packet);
    protected void buildCallgram();
    public void run();
    int add(run.transport.Packet);
    protected run.transport.Packet init(run.transport.Packet);
    protected run.transport.Packet acki(run.transport.Packet);
    protected run.transport.Packet ret(run.transport.Packet);
    protected run.transport.Packet ackr(run.transport.Packet);
    protected run.transport.Packet rn(run.transport.Packet);
    protected run.transport.Packet data(run.transport.Packet);
    protected run.transport.Packet data(run.transport.Packet, int);
    protected run.transport.Packet disc(run.transport.Packet);
    protected run.transport.Packet ackd(run.transport.Packet);
    public java.lang.String toString();
}
```

B.3 Package: run.exec

Listato B.9: Manager

```
Compiled from Manager.java
public class run.exec.Manager extends java.lang.Object {
    static java.lang.Class class$run$exec$Manager;
    static java.lang.Class class$run$session$SessionTable;
    public run.exec.Manager();
    public static run.session.Callgram execRemoteCall(run.session.Callgram) throws run.transport
    public static void main(java.lang.String[]) throws java.lang.Exception;
    private static boolean selftest() throws java.lang.Exception;
    private static boolean selftest_callgram() throws java.lang.Exception;
    static java.lang.Class class$(java.lang.String);
}
```

Listato B.10: Pusher

```
Compiled from Pusher.java
public class run.exec.Pusher extends java.lang.Object implements run.session.CallgramListener {
    protected static long TIME_OUT;
    run.session.Callgram callgram_reply;
    run.exec.Pusher();
    void free();
    run.session.Callgram execRemoteCall(run.session.Callgram) throws run.serialization.Serializat
    public void execCall(run.session.Callgram);
    static {};
}
```

Listato B.11: Dispatcher

```
Compiled from Dispatcher.java
public class run.exec.Dispatcher extends java.lang.Object {
    private static run.exec.ThreadPool thread_pool;
    static run.session.CallgramListener dispatcher;
    static java.lang.Class class$run$exec$Dispatcher;
    public run.exec.Dispatcher();
    public static run.session.CallgramListener getDispatcher();
    public static java.lang.String getStatus();
    static java.lang.Class class$(java.lang.String);
    static run.exec.ThreadPool access$0();
    static {};
}
```

Listato B.12: CallThread

```
Compiled from CallThread.java
public class run.exec.CallThread extends java.lang.Thread implements run.session.CallgramListene
    protected static long TIME_OUT;
    private static int counter;
    run.exec.CallThreadPool thread_pool;
    private static final java.lang.Class Container_class;
    private run.session.Callgram callgram_event;
    static java.lang.Class class$run$serialization$Container;
    public run.exec.CallThread(java.lang.ThreadGroup,run.exec.CallThreadPool);
    public void execCall(run.session.Callgram);
    public void run();
    private run.session.Callgram execLocalCall(run.session.Callgram);
    static java.lang.Class class$(java.lang.String);
    static {};
}
```

B.4 Package: run.reference

Listato B.13: Service

```
Compiled from Service.java
public final class run.reference.Service extends java.lang.Object implements run.I
    java.lang.String name;
    java.lang.String signature;
    public static final long SUID;
    public run.reference.Service(java.lang.String,java.lang.String);
    public final java.lang.String getName();
    public final java.lang.String getSignature();
    public run.reference.Service();
    public final void writeObject(run.serialization.Container.ContainerOutputStrea
    public final void readObject(run.serialization.Container.ContainerInputStream)
    public final int sizeOf();
    public java.lang.String toString();
    static {};
}
```

Listato B.14: RemoteServiceReference

```
Compiled from RemoteServiceReference.java
public class run.reference.RemoteServiceReference extends java.lang.Object {
    private run.transport.NetAddress address;
    private java.lang.String services_name;
    run.reference.Service service;
    public run.reference.RemoteServiceReference(run.transport.NetAddress,java.lang.String,run.re
    public final java.lang.String getServiceName();
    public final java.lang.String getServiceSignature();
    public final java.lang.String getServiceNumericSignature();
    public final java.lang.String getServicesName();
    public final run.transport.NetAddress getAddress();
    public java.lang.String toString();
}
```

Listato B.15: RemoteServiceReference

```
Compiled from LocalServicesReference.java
public class run.reference.LocalServicesReference extends java.lang.Object {
    java.lang.String services_name;
    run.reference.Service service_list[];
    run.stub_skeleton.Skeleton skeleton;
    public run.reference.LocalServicesReference(java.lang.String,run.reference.Service[],run.stu
    public final java.lang.String getServicesName();
    public final run.stub_skeleton.Skeleton getSkeleton();
    public final void setSkeleton(run.stub_skeleton.Skeleton);
    public java.lang.String toString();
}
```

B.5 Package: run.stub_skeleton

Listato B.16: Stub

```
Compiled from Stub.java
public abstract class run.stub_skeleton.Stub extends java.lang.Object implements run.Services {
    public static final java.lang.String CLASSNAME_SUFFIX;
    protected rio.registry.RegistryServicesReference registry_services_reference;
    public run.stub_skeleton.Stub(rio.registry.RegistryServicesReference);
    public java.lang.String getClassname();
    public java.lang.String toString();
    static {};
}
```

Listato B.17: Skeleton

```
Compiled from Skeleton.java
public abstract class run.stub_skeleton.Skeleton extends java.lang.Object {
    public static final java.lang.String CLASSNAME_SUFFIX;
    protected run.Services services;
    public run.stub_skeleton.Skeleton(run.Services);
    public java.lang.String getClassname();
    public final run.Services getServices();
    public final void setServices(run.Services);
    static {};
}
```

B.6 Package: rio.registry

Listato B.18: Local

```
Compiled from Local.java
public class rio.registry.Local extends java.lang.Object implements run.Services {
    protected static java.util.Map local_servicetable;
    static java.lang.Class class$rio$registry$RegistryServicesReference;
    static java.lang.Class class$run$Services;
    public rio.registry.Local();
    public static void register(run.reference.LocalServicesReference) throws rio.registry.Regist
    public static run.reference.LocalServicesReference lookup(run.reference.RemoteServiceReferen
    static run.stub_skeleton.Stub newStub(rio.registry.RegistryServicesReference) throws rio.reg
    static run.stub_skeleton.Skeleton newSkeleton(rio.registry.RegistryServicesReference, run.Se
    static java.lang.Class class$(java.lang.String);
    static {};
}
```

Listato B.19: Global

```
Compiled from Global.java
public class rio.registry.Global extends java.lang.Object implements run.Services {
    static run.reference.Service register_service;
    static run.reference.Service lookup_service;
    static run.reference.Service global_services[];
    protected static java.util.Map global_servicetable;
    public rio.registry.Global();
    public static void register(rio.registry.RegistryServicesReference) throws run.RemoteExcepti
    public static rio.registry.RegistryServicesReference lookup(java.lang.String) throws run.Rer
    static {};
}
```

B.7 Package: clio

Listato B.20: ClusterClassLoader

```
Compiled from ClusterClassLoader.java
public class clio.ClusterClassLoader extends java.lang.ClassLoader {
    private static java.io.File repository_url;
    private static de.fub.bytecode.ClassPath repository_classpath;
    private static final clio.ClusterClassLoader cluster_class_loader;
    private final java.util.Map class_repository;
    public static final clio.ClusterClassLoader getClusterClassLoader();
    public clio.ClusterClassLoader();
    public java.lang.Class findClass(java.lang.String) throws java.lang.ClassNotFoundException;
    java.lang.Class javaClassToClass(de.fub.bytecode.classfile.JavaClass);
    de.fub.bytecode.classfile.JavaClass loadFromUserRepository(java.lang.String);
    private void storeIntoUserRepository(de.fub.bytecode.classfile.JavaClass);
    private de.fub.bytecode.classfile.JavaClass generateStubSkeleton(java.lang.String) throws ja
    public java.lang.String toString();
    static {};
}
```

Listato B.21: StubGenerator

```
Compiled from StubGenerator.java
final class clio.StubGenerator extends clio.Generator {
    private static final de.fub.bytecode.generic.Type init_arg_type[];
    clio.StubGenerator();
    static de.fub.bytecode.classfile.JavaClass generateStubClass(de.fub.bytecode.classfile.JavaC
    private static void addInit(de.fub.bytecode.generic.ClassGen, de.fub.bytecode.generic.Consta
    private static void addServiceStub(de.fub.bytecode.generic.ClassGen, de.fub.bytecode.generic
    static {};
}
```

Listato B.22: SkeletonGenerator

```
Compiled from SkeletonGenerator.java
final class clio.SkeletonGenerator extends clio.Generator {
    private static final de.fub.bytecode.generic.Type init_arg_type[];
    clio.SkeletonGenerator();
    static de.fub.bytecode.classfile.JavaClass generateSkeletonClass(de.fub.bytecode.classfile.J
    private static void addInit(de.fub.bytecode.generic.ClassGen, de.fub.bytecode.generic.Consta
    private static void addServiceSkeleton(de.fub.bytecode.generic.ClassGen, de.fub.bytecode.ger
    static {};
}
```

Listato B.23: RWSGenerator

```
Compiled from RWSGenerator.java
final class clio.RWSGenerator extends clio.Generator {
    static final java.lang.String init_signature;
    static final java.lang.String read_signature;
    static final java.lang.String write_signature;
    static final java.lang.String sizeof_signature;
    clio.RWSGenerator();
    static de.fub.bytecode.classfile.JavaClass modify(de.fub.bytecode.classfile.JavaClass);
    private static boolean checkAndSetInit(de.fub.bytecode.generic.ClassGen);
    private static boolean addInit(de.fub.bytecode.generic.ClassGen, de.fub.bytecode.generic.Cor
    private static void addReadObject(de.fub.bytecode.generic.ClassGen, de.fub.bytecode.generic.
    private static void addWriteObject(de.fub.bytecode.generic.ClassGen, de.fub.bytecode.generic
    private static void addSUID(de.fub.bytecode.generic.ClassGen, de.fub.bytecode.generic.Consta
    private static void addSizeOf(de.fub.bytecode.generic.ClassGen, de.fub.bytecode.generic.Cons
```

```
    private static void addSuperInvoke(de.fub.bytecode.generic.ClassGen, de.fub.bytecode.generic
    private static void addSuperSizeOf(de.fub.bytecode.generic.ClassGen, de.fub.bytecode.generic
    static {};
}
```

Listato B.24: SUID

```
Compiled from SUID.java
public class clio.SUID extends java.lang.Object {
    private static final clio.ClusterClassLoader ccl;
    private static java.util.Comparator compareByName;
    public clio.SUID();
    public static final long getSUID(java.lang.Class);
    private static long computeSUID(java.lang.Class);
    public static long computeSUID(java.lang.String);
    static long computeSUID(de.fub.bytecode.classfile.JavaClass);
    static {};
    private static class clio.SUID. CompareByName extends java.lang.Object implements java.util.
    /* ACC_SUPER bit NOT set */
{
        private clio.SUID.CompareByName();
        public int compare(java.lang.Object, java.lang.Object);
        clio.SUID.CompareByName(clio.SUID$$1);
    }
}
```

Bibliografia

[ASU86] Aho, Sethi, and Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Publishing Company, 1986.

[AW97] Hagit Attiya and Jennifer Welch. *Distributed Computing*. McGraw-Hill, 1997.

[BB99] Baker and Buyya. *Cluster Computing at a Glance: Architectures and Systems*, volume 1. Prentice Hall, NJ, USA, 1 edition, 1999.

[BG92] Dimitri Bertsekas and Robert Gallager. *Data Networks*. Prentice Hall, Upper Saddle River, NJ 07458, 2 edition, 1992.

[Con00] Java Developer Connection. Class Loaders as a Namespace Mechanism. Technical report, SUN, 2000.

[Dah98a] Markus Dahm. ByteCode Engineering. Technical report, Freie Universität Berlin, 1998.

[Dah98b] Markus Dahm. ByteCode Engineering with the BCEL API. Technical Report B-17-98, Freie Universität Berlin Institut für Informatik, 1998.

[Jav98] JavaGrande: Java Portal for Grande Application. http://www.javagrande.org, 1998.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Weshley, 2 edition, 1999.

[MNV+00] Jason Maassen, Rob Van Nieuwpoort, Ronald Veldema, Henri Bal, Thilo Kielmann, Ceriel Jacobs, and Rutger Hofman. Efficient Java RMI for Parallel Programming. Technical report, Department of Mathematics and Computer Science, Amsterdam, The Netherlands, 2000.

[New00] Ted Neward. Understanding Class.forName(). Technical report, A JavaGeeks White Paper, http://www.javageeks.com, 2000.

[PH99] Michael Philippsen and Bernard Haumacher. UKA Transport: Efficient Serialization in Java. In *Parallel And Distributed Processing*. International Workshop On Java For Parallel And Distributed Computing, 1999.

[SUN97a] SUN Microsystems Inc., http://java.sun.com. *Java Core Reflection*, 1997.

[SUN97b] SUN Microsystems Inc., http://java.sun.com /products /jdk /1.1 /docs /guide /rmi /index.html. *Java Remote Method Specification*, 1997.

[SUN98] SUN Microsystems Inc., http://java.sun.com /products /jdk /1.2 /docs /guide /serialization /index.html. *Java Object Serialization Specification*, 1998.

[VIA97] Virtual Interface Architecture. http://www.viarch.org, 1997.

[WC00] Matt Welsh and David Culler. Jaguar: Enabling Efficient Communication and I/O in Java. Technical report, Berkeley University, Department of Computer Science, http://www.cs.berkeley.edu/ mdw, 2000.