UNIVERSITÀ DEGLI STUDI DI FIRENZE
Facoltà di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Informatica

Definizione e realizzazione di miglioramenti alle metodologie per l'analisi di sistemi distribuiti

Candidata
Giuseppina Bastone

Relatore
Prof. Andrea Bondavalli

Correlatore
Dott. Lorenzo Falai

Anno Accademico 2004/2005

Prefazione

Lo sviluppo dei sistemi distribuiti ha portato alla sempre più massiccia realizzazione di applicazioni distribuite e quindi alla definizione di metodi che permettono una valutazione delle proprietà astratte e logiche di questi algoritmi distribuiti prima, e una valutazione quantitativa dopo. A sostegno di tali tecniche di valutazione è stato sviluppato il framework Neko che, insieme al pacchetto NekoStat, consente una valutazione completa degli algoritmi distribuiti.
Uno dei principali motivi che ha portato alla diffusione del framework Neko è la sua implementazione in Java, che lo rende estremamente indipendente da qualsiasi piattaforma hardware e da qualsiasi sistema operativo. Questo comporta, tuttavia, che qualsiasi algoritmo distribuito, scritto in un linguaggio di programmazione diverso da Java, sia tradotto in Java prima di poter essere valutato, con un conseguente aumento dei costi di valutazione, in termini di tempo e di denaro, inoltre la "traduzione" di un algoritmo, da un linguaggio a un altro, non garantisce che il comportamento dell'algoritmo tradotto sia identico all'originale. Nasce dunque la necessità di permettere l'utilizzo di Neko e NekoStat per la valutazione di algoritmi distribuiti scritti in linguaggi di programmazione diversi da Java, ad esempio il C o il C++. Lo scopo del mio lavoro è, dunque, rendere possibile l'integrazione di elementi scritti in C o C++ all'interno del framework Neko; tale integrazione è stata possibile grazie all'uso di elementi software di integrazione quali il *glue code* e l'*adattamento dei componenti*.

La tesi è composta da due parti; nella prima, dal primo al terzo capitolo, elenco le motivazioni del mio lavoro e introduco i sistemi distribuiti e gli strumenti utilizzati per la valutazione di algoritmi distibuiti, nella seconda parte, dal quarto al sesto capitolo, fornisco una descrizione dei concetti necessari a comprendere i termini e gli strumenti da me utilizzati, il lavoro da me svolto e un esempio che ne evidenzi l'uso e i vantaggi. In particolare:

nel primo capitolo introduco brevemente il problema della valutazione di algoritmi distribuiti e fornisco le motivazioni del mio lavoro;

nel secondo capitolo fornisco una definizione di sistema distribuito, i vantaggi e gli svantaggi dovuti al loro uso e ulteriori informazioni su tali sistemi, nonchè le tecniche utilizzate per la loro analisi quantitativa;

nel terzo capitolo presento il framework Neko e descrivo il pacchetto NekoStat che permette la valutazione quantitativa degli algoritmi distribuiti;

nel quarto capitolo, oltre a introdurre i concetti necessari a comprendere il mio lavoro, illustro le problematiche che occorre affrontare in queste situazioni;

nel quinto capitolo presento il lavoro vero e proprio e le estensioni da apportare a Neko e NekoStat per la sua realizzazione;

nel sesto capitolo, l'estensione da me sviluppata viene utilizzata per la valutazione di un meccanismo per la message freshness detection il cui algoritmo è scritto in C.

Infine, nell'**Appendice A**, fornisco un manuale d'uso della mia estensione specificando gli strumenti necessari per il suo utilizzo.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Ogni settore della nostra vita, oggigiorno, vede una presenza massiccia dei sistemi informatici; questa diffusione ha fatto sì che ad essi venissero affidate notevoli risorse, umane ed economiche, con una conseguente richiesta di affidabilità che giustifichi un tale investimento. Per soddisfare questa richiesta, gli sviluppatori cercano, sempre più, di fornire sistemi sui quali gli utenti devono poter porre fiducia. Una proprietà fondamentale dei sistemi informatici che rappresenta una misura di quanta fiducia possiamo riporre, in maniera giustificata, sul servizio fornito da un sistema è la *dependability* ([2, 3]). La dependability può essere anche espressa come l'abilità di un sistema di soddisfare specificati servizi ([4, 5]); il servizio di un sistema è classificato come *servizio proprio* se aderisce alle specifiche funzionali stabilite, come *servizio improprio* altrimenti. Nel caso in cui il sistema fornisce un servizio proprio si dice che esso si trova in uno stato corretto, altrimenti si trova in uno stato non corretto. Una transizione da uno stato di servizio proprio ad uno di servizio improprio rappresenta un fallimento.
Per capire quali sono i modi sistematici per costruire sistemi dependable è necessario definire quali sono gli *impedimenti* alla dependability, i *mezzi* per ottenerla e gli *attributi* della dependability ([2, 5, 6]). Gli impedimenti sono le cause potenziali di comportamenti non previsti, i

mezzi sono le tecniche che permettono di ottenere comportamenti corretti nonostante il verificarsi degli impedimenti e gli attributi ci permettono di esprimere e verificare il livello di dependability richiesto.

Impedimenti alla dependability

Gli impedimenti alla dependability possono essere di tre tipi: *guasti*, *errori* e *fallimenti*. Un *fallimento* è una transizione da un servizio corretto a un servizio non corretto. I possibili fallimenti di un sistema possono essere classificati in base alle possibili conseguenze del fallimento sul sistema e sull'ambiente esterno; una prima classificazione dei fallimenti prevede di distinguere i fallimento *benigni* da quelli *catastrofici*.
Il processo che porta a un fallimento ha origine da una causa interna o esterna al sistema chiamata *guasto*; un guasto è detto attivo quando produce un errore, latente altrimenti. L'*errore* è quella parte del sistema che può causare un successivo fallimento.
La classificazione dei guasti può essere fatta in base a diversi punti di vista, una prima distinzione può essere fatta in base alla natura del guasto: un guasto può essere *intenzionale* o *accidentale*, *malizioso* o *non malizioso*. Un'altra distinzione può essere fatta in base all'origine fenomenologica del guasto:

- *guasto fisico*: generato da cause fisiche (hardware);
- *guasto di design*: indotto durante la fase di progettazione del sistema;
- *guasto di interazione*: guasto che si verifica all'interfaccia fra i componenti del sistema o all'interfaccia con il mondo esterno.

I guasti di maggiore rilevanza per i sistemi distribuiti sono i *guasti di interazione*; nei sistemi distribuiti, infatti, i fallimenti sono spesso originati da guasti nell'interazione dei componenti (distribuiti) del sistema. Tra i guasti di interazione, in questo tipo di sistema, i guasti più rilevanti sono i guasti *transienti*, che possono influenzare negativamente la

comunicazione (per esempio una scarica elettromagnetica che produce un rumore sul segnale che trasporta un messaggio in rete), e i guasti *intermittenti*, che colpiscono principalmente i programmi concorrenti (per esempio il deadlock).

Mezzi per ottenere la dependability

Rappresentano le tecniche utilizzate per rendere un sistema dependable, esse sono:

- **Fault removal.** La rimozione dei guasti è una tecnica che si basa sul rilevamento dei guasti e la successiva rimozione, prima della loro attivazione. Con questa tecnica, per esempio, si tenta di individuare e rimuovere bug software, o hardware con difetti. La rimozione dei guasti è utilizzata sia nella fase di sviluppo che nella fase operativa del sistema.

  Durante la fase di sviluppo la rimozione dei guasti consiste di tre operazioni: la *verifica*, la *diagnosi* e la *correzione*. Compito della verifica è di controllare se il sistema aderisce alle proprietà specificate, in caso contrario vengono eseguite la diagnosi e la correzione dei guasti che hanno portato al fallimento della verifica. Le tecniche di verifica si distinguono essenzialmente in tecniche di *verifica statica* e tecniche di *verifica dinamica*. La verifica statica si basa sull'analisi del sistema senza l'esecuzione reale dello stesso; esempi di tecniche di questo tipo sono il model-checking e l'analisi statica del codice. La verifica dinamica si basa, invece, sull'osservazione del sistema in esecuzione; tecniche di questo tipo sono ad esempio il testing e i metodi di fault injection, attraverso i quali si inseriscono guasti o errori nei test per osservare le reazioni del sistema.

  La rimozione dei guasti durante la fase operativa è detta *corrective maintenance* e può assumere le seguenti forme: *curative maintenance* mirata alla rimozione dei guasti che hanno prodotto uno o

più errori e *preventive maintenance* che ha lo scopo di rimuovere i guasti prima che possano produrre errori.

- **Fault prevention.** Le tecniche di prevenzione dei guasti, come dice il nome stesso, si occupano di prevenire le cause dei possibili errori, eliminando le condizioni che rendono l'occorrenza dei guasti più probabile; questo scopo si ottiene attraverso tecniche di controllo della qualità impiegate durante la progettazione e fabbricazione di hardware e software, tra esse citiamo: per il software, la programmazione strutturata, l'informazione nascosta e la modularizzazione, e per l'hardware, rigorose regole di progettazione.

- **Fault tolerance.** Le tecniche viste sopra non permettono di eliminare totalmente i possibili guasti del sistema, per questo motivo si utilizzano spesso metodi per la tolleranza ai guasti. La tolleranza ai guasti consente al sistema di rimanere nello stato di servizio proprio nonostante la presenza di guasti attivi; essa è implementata generalmente da due fasi: il *rilevamento dell'errore* e il *ripristino del sistema*. Nella fase di ripristino si trasforma lo stato erroneo del sistema in uno stato senza errori rilevati e senza guasti che possano essere riattivati; il ripristino viene effettuato attraverso la *gestione dell'errore* e la *gestione del guasto*.
  Con la gestione dell'errore si tenta l'eliminazione degli errori dello stato del sistema, le tecniche principali sono: il *rollback*, ossia si ripristina uno stato del sistema salvato precedentemente (un checkpoint) privo di errori; la *compensazione*, consiste nel ricavare uno stato privo di errori dallo stato erroneo, se quest'ultimo però ha abbastanza informazioni ridondanti da poter eliminare gli errori; il *rollforward*, ossia si calcola, a partire dallo stato corrente erroneo, uno stato successivo del sistema, senza errori rilevati.
  Dopo la gestione dell'errore si può passare alle tecniche per la gestione del guasto; lo scopo di queste tecniche è di disattivare i

guasti in modo che non possano essere attivati di nuovo.

- **Fault forecasting.** Qualunche tecnica o combinazione di tecniche utilizzata difficilmente può portare alla costruzione di un sistema totalmente libero da guasti; per questo esistono le tecniche di previsione dei guasti. La previsione è condotta tramite valutazioni sul comportamento del sistema rispetto all'occorrenza o all'attivazione del guasto stesso. Possono esserci valutazioni *qualitative* e *quantitative*; nel primo caso, si cerca di identificare, classificare e ragruppare i modi di fallimento o la combinazione di eventi che portano al fallimento. Nel caso di valutazione quantitativa o *probabilistica*, si cerca di valutare in termini probabilistici i valori per i quali alcune misure di dependability sono soddisfatte.

Gli attributi della dependability

Gli attributi permettono di definire delle misure per rappresentare il grado di dependability del sistema. Il concetto di dependability comprende i seguenti attributi di base:

- **availability**: prontezza nel fornire un servizio corretto;
- **reliability**: continuità nel fornire un servizio corretto;
- **safety**: assenza di catastrofiche conseguenze sugli utenti e sull'ambiente;
- **confidentiality**: assenza di divulgazioni delle informazioni non autorizzate;
- **integrity**: assenza di alterazioni improprie dello stato del sistema;
- **maintenability**: capacità di subire riparazioni e modifiche.

In generale, per ogni sistema saranno solo alcune le misure di interesse ed i singoli attributi potranno essere più o meno importanti in base alle

funzioni che il sistema dovrà svolgere; tuttavia, l'availability è sempre richiesta, seppur con grado diverso, mentre gli altri attributi possono interessare o meno. Oltre agli attributi appena citati se ne possono definire altri, come combinazioni o specificazioni dei precedenti. La **security** ad esempio indica la simultanea presenza di availability, confidentiality ed integrity; una sua definizione potrebbe essere: assenza di accessi non autorizzati.

Altre proprietà fondamentali, oltre la dependability, che caratterizzano i sistemi informatici sono la *performance*, il *costo* e la *performability*. Con il termine performance ci riferiamo alle misure che rappresentano come un sistema fornisce un servizio, in termini di efficacia (per esempio il ritardo o il throughput) e di efficienza (per esempio l'utilizzo delle risorse); con il termine performability identifichiami un attributo che quantifica la capacità del sistema di fornire un servizio degradato in seguito a fallimenti, la performability è quindi una metrica combinata di performance e dependability.
I concetti di performance, dependability e performability vengono inoltre utilizzati come concetti base per definire sia la specifica che il rispetto della specifica da parte del sistema.
É importante, dunque, poter valutare se un sistema rispetta la specifica definita in modo soddisfacente, attraverso un *processo di validazione*. Uno degli aspetti necessari alla validazione è la capacità di poter valutare caratteristiche sia quantitative che qualitative del sistema in esame.
Vediamo, brevemente, quali sono gli approcci utilizzati per la valutazione di performance, dependability e performability di algoritmi distribuiti, nel Capitolo 2 vedremo in dettaglio tali tecniche di valutazione e la loro classificazione. L'*approccio analitico* permette la valutazione delle prestazioni dell'algoritmo basandosi su un modello parametrico dell'ambiente di esecuzione; l'*approccio simulativo* permette di simulare l'algoritmo in un ambiente di esecuzione; l'*approccio basato su misure*

permette di eseguire l'algoritmo in un ambiente reale, in parallelo con un meccanismo di monitoraggio che raccolga i dati dai quali estrarre le misure o le proprietà di interesse. Molto spesso è consigliato confrontare i risultati ottenuti da due approcci diversi per permettere di aumentare la confidenza e l'accuratezza dei risultati ottenuti. É quindi vantaggioso avere uno strumento come *Neko* (Capitolo 3), utilizzabile per l'analisi di algoritmi distribuiti attraverso simulazione e misure; in una piattaforma simile, un'unica implementazione di un algoritmo può essere utilizzata sia per simulare il comportamento dell'algoritmo che per l'esecuzione su di una rete reale: in questo modo, oltre ad avere risultati più affidabili, si hanno tempi di sviluppo e testing per un algoritmo inferiori rispetto a quelli ottenuti da un approccio tradizionale.

1.1 Motivazioni

Spesso, affinchè un sistema sia valutato, è necessario costruire un modello del sistema, ovvero una descrizione semplificata della realtà; è ovvio che più un modello si avvicina alla realtà dell'oggetto considerato più i risultati ottenuti sono affidabili.
Nel caso del framework Neko, è possibile simulare ed eseguire algoritmi distribuiti scritti in Java, un qualsiasi algoritmo scritto in un diverso linguaggio di programmazione dovrà essere tradotto in Java prima di poter essere valutato; in pratica, quello che si fa è costruire un modello dell'algoritmo distribuito il più accurato possibile. Nonostante sia richiesta la massima accuratezza, niente e nessuno garantisce che il modello proposto si comporta esattamente come il sistema originale.
Negli ultimi anni il linguaggio di programmazione Java sta vivendo una grande diffusione, giustificata dal fatto che

- Java è il primo linguaggio che permette un'indipendenza dal sistema operativo e dall'hardware, consentendo la migrazione del

codice da un computer all'altro alla sola condizione che il calcolatore sia dotato della Java Virtual Machine[1];

- nel caso di download del codice da remoto, il caricamento in memoria fatto per classi permette di limitare la quantità di codice da trasmettere sulla rete, in quanto si possono utilizzare le classi di sistema presenti sul calcolatore di destinazione;

- il caricamento parziale di codice permette di rendere maggiormente dinamici lo sviluppo e la successiva evoluzione del sistema.

Nonostante i vantaggi derivanti dall'uso di Java capita spesso di incontrare algoritmi distribuiti scritti in linguaggi di programmazione diversi, magari perchè per la soluzione di uno specifico problema è consigliabile usare un diverso linguaggio; tra i linguaggi maggiormente utilizzati risultano il C e il C++. Basti pensare ai sistemi critici e real-time: per questi tipi di sistemi il linguaggio Java è inutilizzabile.
Ancora, l'eliminazione della traduzione da altri linguaggi di programmazione a Java, oltre a garantire risultati più affidabili dal processo di valutazione, permette di ottenere una riduzione dei costi sia in termini di tempo che di denaro. Riassumendo, le motivazioni che presento a sostegno del mio lavoro sono:

- maggiore accuratezza e affidabilità dei risultati ottenuti dalla valutazione, attraverso il framework Neko, di algoritmi distribuiti originariamente scritti in C o C++;

- necessità di valutare algoritmi distribuiti, scritti in C o C++, di cui si vogliono avere "garanzie" sul loro funzionamento e che, nonostante la massiccia diffusione del linguaggio Java, rappresentano, ancora oggi, una parte consistente degli algoritmi in commercio;

- riduzione dei costi di valutazione.

[1]Modulo che provvede al caricamento e all'esecuzione delle classi java.

Capitolo 2

Sistemi distribuiti e analisi quantitativa

Per ottenere sistemi affidabili sono state sviluppate diverse tecniche di analisi, progettazione e realizzazione di sistemi fault-tolerant, cioè di sistemi in grado di gestire situazioni critiche legate alla presenza di guasti. Uno degli strumenti maggiormente utilizzati per la realizzazione di tali sistemi è l'uso della ridondanza: se un guasto compromette la funzionalità di una replica di uno degli elementi di un sistema di elaborazione, grazie alla replicazione può essere comunque garantita una continuità nel servizio offerto. Perchè tuttavia la ridondanza sia utile occorre che i guasti si presentino in maniera indipendente sulle varie repliche. Questo è uno dei principali motivi che ha portato allo sviluppo dei sistemi distribuiti [7].
Nel seguito del presente capitolo, oltre alla definizione di sistema distribuito, ho evidenziato gli ulteriori vantaggi derivanti dall'uso di tali sistemi, nonchè la classificazione dei modelli di sistemi distribuiti e le tecniche per la loro analisi quantitativa.

2.1 Definizione di sistema distribuito

Un sistema distribuito è un insieme di processi che comunicano tra loro per mezzo di diverse reti di comunicazione, come bus ad alta velocità o linee telefoniche. I processori che compongono un sistema distribuito variano per dimensioni e funzioni: i sistemi distribuiti possono infatti comprendere piccoli microprocessori, stazioni di lavoro, minicomputer e grandi computer general purpose[1].

In un sistema di questo tipo i processori non condividono la memoria o un clock, ognuno di essi ha la propria memoria; questo è ciò che distingue i sistemi distribuiti dai sistemi multiprocessore (ulteriore sistema utilizzato per distribuire il calcolo tra diversi processori fisici). I processori di un multiprocessore, detti saldamente accoppiati, condividono la memoria e un clock, e la comunicazione avviene normalmente attraverso la memoria condivisa.

Un'ulteriore distinzione va fatta tra sistema distribuito, multiprocessore e multicomputer. I multicomputer sono sistemi composti da processori fortemente accoppiati, con risorse di base separate; la comunicazione fra questi processori avviene attraverso appositi bus o piccole reti molto veloci. I processori di un sistema distribuito sono debolmente accoppiati: la comunicazione avviene attraverso una rete inaffidabile, con ritardi di trasmissione estremamente variabili e velocità e banda moderate.

Il processore di un sistema distribuito considera *remoti* gli altri processori del sistema e le rispettive risorse, considera *locali* le proprie risorse. A seconda del contesto in cui vengono citati, i processori vengono chiamati in modi diversi: *siti*, *nodi*, *computer*, *macchine*, *host* e così via.

Con il termine *risorse* vengono indicate sia le risorse hardware che quelle software; l'accesso a queste risorse è controllato dal sistema operativo.

Fondamentalmente esistono due schemi complementari che garantiscono

[1]Un computer general purpose è un computer progettato per poter svolgere, potenzialmente, con l'opportuno software, qualsiasi compito realizzabile da una macchina.

questa condivisione delle risorse:

- **Sistemi operativi di rete**: gli utenti sono a conoscenza delle macchine presenti e devono accedere a queste risorse effettuando un login nella macchina remota appropriata, oppure trasferendo i dati dalla macchina remota alle loro macchine.

- **Sistemi operativi distribuiti**: gli utenti non hanno bisogno di essere a conoscenza delle numerose macchine presenti; essi accedono alle risorse remote nello stesso modo in cui accedono a quelle locali.

2.2 Vantaggi e svantaggi

Motivi principali che ci inducono a utilizzare i sistemi distribuiti([8, 9]):

1. **Condivisione delle risorse.** Se siti diversi, con risorse diverse, sono collegati tra loro, allora l'utente di un sito ha la possibilità di utilizzare le risorse disponibili su un altro sito. La condivisione avviene solitamente attraverso l'uso di tecniche per l'accesso remoto a siti centrali.
   Questa è stata la prima motivazione che ha portato alla nascita dei sistemi distribuiti, ed è ancora oggi uno degli obiettivi di gran parte di questi sistemi.

2. **Accelerazione dei calcoli**. Se un particolare calcolo può essere suddiviso in più sottocalcoli eseguibili concorrentemente, disponendo di un sistema distribuito è possibile distribuire la computazione tra diversi siti per un'esecuzione concorrente.
   Inoltre, se un sito è sovraccarico di job, alcuni di essi possono essere spostati su altri siti con carico inferiore; lo spostamento dei job è chiamato *condivisione di carico*.

3. **Affidabilità**. Questa è uno dei tipici vantaggi che si attribuisce ad un sistema distribuito. Ovvero la sua capacità di sopravvivere a un guasto di una sua entità grazie alle sua ridondanza intrinseca. In pratica, se un sito di un sistema distribuito si guasta, i restanti possono potenzialmente continuare a lavorare.
   Perchè 'potenzialmente'?
   Il motivo è il seguente:

   - se il sistema è formato da un certo numero di computer general purpose, il guasto di uno di essi non influisce sugli altri;
   - se invece il sistema è formato da un certo numero di piccole macchine, ciascuna delle quali è responsabile di una funzione cruciale per il sistema, allora un piccolo guasto può interrompere il funzionamento di tutto il sistema.

   In generale, se il sistema è ridondante -sia a livello hardware che a livello software- può continuare a funzionare anche in seguito ad alcuni guasti.

4. **Comunicazione**. Gli utenti di diversi siti, collegati per mezzo di una rete di comunicazione, possono scambiarsi informazioni. A un livello basso, tra i sistemi vengono scambiati *messaggi* in modo simile a quello dei messaggi in un singolo computer.

5. **Performance e disponibilità del servizio**. Nessun server ovviamente può offrire performance infinite nè può rimanere attivo in maniera continuativa, è per questo motivo che sono state ideate strategie che permettono l'utilizzo di server *logici*. Essi possono essere, poi, dinamicamente mappati su server *fisici*, in modo da garantire una degradazione graduale del servizio offerto. Per ottenere ciò si utilizzano tecniche di bilanciamento del carico, che permettono l'incremento della performance e della disponibilità

del sistema.
La performance e la disponibilità possono essere aumentate anche utilizzando la distribuzione geografica dei diversi server.

6. **Sistemi scalabili**. L'utilizzo di architetture modulari nella costruzione di un sistema distribuito, permette la realizzazione di sistemi scalabili.

A fronte dei vantaggi abbiamo anche una serie di *svantaggi* di cui dobbiamo tener conto e a cui si sta rispondendo con la nascita di nuove tecnologie che cercano di mitigare tali svantaggi:

1. **Produzione di software**. Fino a pochi anni fa, uno dei più seri problemi riguardo lo sviluppo dei sistemi distribuiti era dato dalla mancanza di strumenti software per la realizzazione di applicazioni che potessero mostrare al mercato l'importanza di tali sistemi. Condizione ormai superata. Il cambiamento è stato segnato da tre passaggi importanti:

   - la nascita e lo sviluppo di metodologie, linguaggi di analisi e design e di strumenti orientati agli oggetti che hanno svolto un ruolo fondamentale nel permettere di costruire sistemi informativi sempre di maggior complessità, fino ad arrivare ai moderni software distribuiti, caratterizzati dai più alti standard di scalabilità e sicurezza;
   - l'affermarsi del linguaggio JAVA che permette l'indipendenza del prodotto software dall'hardware e dal sistema operativo;
   - la diffusione di architetture e specifiche per la creazione, la distribuzione e l'uso di oggetti software distribuiti in una rete; tra esse ricordiamo CORBA, *Common Object Request Broker Architecture*: permette, a programmi in differenti locazioni e sviluppati da programmatori diversi, di comunicare in rete attraverso un'interfaccia. CORBA è stato sviluppato da un

consorzio attraverso l'Object Management Group (OMG) ed è stato approvato dall'organizzazione per gli standard ISO come l'architettura standard per gli oggetti distribuiti; il nucleo centrale di CORBA è l'Object Request Broker (ORB).
ORB intercetta le richieste effettuate dal software client di tipo CORBA, sia che risieda sulla stessa macchina che in rete, quindi si incarica di trovare l'oggetto che può soddisfare la richiesta, gli passa i parametri e ne riceve i risultati che quindi passa al software client. Il software client è del tutto all'oscuro di quale oggetto software abbia risposto alla sua richiesta, nè della sua locazione, nè importa l'uniformità dei sistemi operativi e del processore, nè del linguaggio di programmazione utilizzato. É così possibile utilizzare computer con sistemi operativi diversi e con architettura diversa nella stessa rete, senza alcuna controindicazione nè inconveniente, conservando la completa compatibilità del software e dei dati.

2. **Sicurezza**. Nel momento stesso in cui l'informazione viene decentralizzata e c'è un flusso di informazioni sulla rete emerge il problema della sicurezza.
   Nei sistemi centralizzati, la sicurezza era per lo più fisica, legata cioè alla sicurezza dei dati locali. Oltre a questa sicurezza, ora, dobbiamo affrontare problematiche relative alla *sicurezza della comunicazione*, sicurezza rispetto ad accessi indesiderati ai dati attraverso la rete, protezione da programmi che vengono scaricati dalla rete e mandati in esecuzione locale.

2.3 I sistemi distribuiti di oggi

Un sistema distribuito è un insieme di processi che comunicano tra loro per mezzo di diverse reti di comunicazione. Un particolare sistema di-

stribuito è il *sistema multimediale distribuito*[2], un sistema che permette agli utenti di *condividere*, *comunicare* e *processare* una grande varietà di informazioni, integrate tra di loro. In genere, con il termine *multimediale*, si intende la possibilità di generare e percepire messaggi contenenti informazioni di natura diversa e di complessità elevata, paragonabile a quella che si ha nella comunicazione tra esseri umani. Un'informazione multimediale è costituita da un insieme di informazioni fra di loro eterogenee, che hanno diverse rappresentazioni dei dati e, addirittura, coinvolgono nella percezione i diversi sensi dell'uomo, attualmente la vista e l'udito.
Oggi, con lo sviluppo delle reti a larga banda sono state concepite tantissime applicazioni multimedali in campo industriale, amministrativo, scientifico e domestico; fra queste:

- comunicazione interpersonale e messaggistica avanzata con videoconferenza e videotelefonia, "high mail"[3];
- assemblaggio, accesso, calcolo e distribuzione dell'informazione (televendita e giornali personalizzati);
- pubblicazione, editoria e stampa decentralizzata, produzione di filmati e pubblicità;
- insegnamento a distanza e teledidattica;
- collaborazione fra esperti e telerichiesta di pareri specialistici (telemedicina);
- monitoraggio a distanza, telesorveglianza, controllo, diagnosi, telemantenimento e riparazione di robot;
- progettazione distribuita e cooperativa;

[2]Un sistema multimediale, in linea di principio, potrebbe essere *stand alone*, cioè un calcolatore isolato, anche se attrezzato con equipaggiamento multimediale.

[3]Posta elettronica con la possibilità di includere nel messaggio informazioni vocali e video.

- attività di divertimento e di piacere (libri animati, film interattivi ad alta definizione e tridimensionali).

Inoltre, la rapida evoluzione della rete e dei computer, insieme alla crescita esponenziale dei servizi e delle informazioni reperibili su Internet, ci ha portato, e ci porterà sempre più, al punto in cui centinaia di milioni di persone hanno un veloce accesso a una straordinaria quantità di informazioni dai posti di lavoro, dalle scuole e dalle abitazioni personali, attraverso cellulari, piccoli computer come i PDA (personal digital assistants),.., da ovunque e in qualsiasi momento ([10]).
Le nuove tecnologie dei sistemi distribuiti possono essere incorporate in una nuova emergente area, chiamata *Ubiquitous Computing* ([11]), essa è, in qualche senso, la proiezione del fenomeno Internet e del fenomeno di proliferazione dei telefoni cellulari.
La tecnologia Ubiquitous Computing e la diffusione della tecnologia wireless costituiscono oggi la nuova area di interesse dei sistemi distribuiti; grazie al loro sviluppo, un nuovo telefono cellulare ha un potere computazionale equivalente a quello di un PC del 1998.

2.4 Classificazione dei modelli

Un modello di sistema è una descrizione semplificata della realtà ([7]). Un modello per un oggetto è una collezione di attributi e un insieme di regole che governano come questi attributi interagiscono. Un modello viene detto *accurato* quando si avvicina alla realtà dell'oggetto considerato, *trattabile* quando è usabile per l'analisi dell'oggetto. Un buon modello dovrebbe essere sia accurato che trattabile: gli attributi selezionati devono essere solo quelli che influenzano in modo significativo il fenomeno in esame.
I sistemi distribuiti possono essere modellati come un insieme di processi che interagiscono tramite uno scambio di messaggi, attraverso canali di comunicazione.

Esistono diversi modelli di sistemi distribuiti ed è possibile individuare dei parametri che differenziano i diversi modelli: la sincronia, i tipi di fallimento dei processi o delle comunicazioni, la topologia e il modello di comunicazione della rete e la natura deterministica o random dei processi.
Nel caso dei sistemi distribuiti la soluzione di un dato problema computazionale è dipendente dal modello di sistema scelto: minori sono le restrizioni imposte nel modello, più la soluzione sarà complessa da trovare ma allo stesso tempo più generale. In più, un determinato problema può essere risolvibile in un certo modello e non avere soluzioni in un altro. Si consideri ad esempio il problema del *consenso*, è l'astrazione di una vasta classe di problemi, in cui abbiamo un gruppo di processi che devono mettersi d'accordo su un valore, partendo dalle loro opinioni (locali e forse divergenti) sul valore stesso. É un problema fondamentale per i sistemi distribuiti, esempi di utilizzo sono: la scelta da parte di più database server sul commit/abort di una transazione, l'elezione di un leader in un gruppo, la scelta dell'ordinamento di un insieme di messaggi, la diagnosi dell'errore in sistemi ridondanti (basata sullo scambio dello stato di ogni replica del sistema per il rilevamento di stati computazionali erronei). Ebbene, Fisher, Lynch e Patterson, hanno dimostrato che il problema del consenso non ha soluzione per i sistemi distribuiti asincroni, in cui anche un solo processo può subire un fallimento per crash (risultato noto come *Teorema FLP*, la cui dimostrazione è data in [12]).

2.4.1 I sistemi sincroni e asincroni

La sincronia è un attributo comune ai processi e alle comunicazioni.
In un sistema sincrono abbiamo come ipotesi la conoscenza di un limite superiore sia per gli scostamenti delle velocità di esecuzione dei processi, sia per i ritardi di consegna dei messaggi, il che consente di ridurre il grado di non-determinismo del sistema e di facilitare la progettazione di

applicazioni distribuite. Più formalmente possiamo dire che un sistema è *sincrono* se soddisfa le seguenti proprietà:

1. esiste ed è noto un limite superiore δ sul ritardo di consegna dei messaggi;

2. ogni processo p ha un *clock locale*[4] C_p con un *drift rate*[5] $\rho \geq 0$ conosciuto e limitato rispetto al tempo reale. Quindi per tutti i p e per tutti gli istanti di tempo $t > t^{'}$, si ha

   $$(1+\rho)^{-1} \leq \frac{C_p(t)-C_p(t^{'})}{t-t^{'}} \leq (1+\rho)$$

   Dove $C_p(t)$ è la lettura del clock C_p all'istante di tempo reale t;

3. esistono limiti superiori conosciuti sul tempo richiesto da un processo per eseguire un passo elementare.

In un sistema sincrono possiamo quindi imporre dei timeout, che possono servire come meccanismo di rilevamento dei guasti.
Un sistema distribuito è *asincrono* se non esiste alcun limite sul ritardo nella consegna dei messaggi, sulla deviazione dei clock o sul tempo necessario per eseguire un passo computazionale. In pratica assumere che un sistema sia asincrono equivale ad una non assunzione.
L'interesse verso il modello asincrono è legato al fatto che le applicazioni realizzate su di esso sono più generali e trasportabili di quelle che considerano delle specifiche assunzioni temporali, inoltre i carichi di lavoro variabili o inaspettati sono causa di asincronia, per cui le assunzioni del modello sincrono diventano validi solo in modo probabilistico. Ulteriori ragioni che inducono all'utilizzo del modello asincrono sono: ha una

[4]Il clock locale è un clock fisico, spesso realizzato con oscillatori al quarzo, che può essere rappresentato da una funzione discreta monotona crescente C_p, che mappa il tempo reale t nel tempo del clock $C_p(t)$.

[5]Il drift rate rappresenta lo scostamento tra il clock locale e il tempo reale.

semantica più semplice; un protocollo pensato per girare in un sistema asincrono può essere usato per girare in qualsiasi sistema distribuito; un sistema distribuito realizzato attraverso l'infrastruttura di rete e sistemi operativi commerciali ad oggi più comuni (per esempio Internet e Windows XP/2000/NT) è riconducibile ad un modello asincrono, infatti, i protocolli UDP/IP e TCP/IP non sono in grado di limitare superiormente il tempo di consegna di un pacchetto e i sistemi operativi commerciali non sono anch'essi in grado di assicurare scostamenti limitati nelle velocità dei processi in esecuzione.
Sebbene il modello asincrono sia un modello molto attraente, molti problemi di base nell'ambito dei sistemi distribuiti non possono essere risolti in sistemi asincroni (per esempio il problema del consenso [12]). Per questo sono stati studiati modelli che si pongono tra quello sincrono e quello asincrono, tra essi il *modello parzialmente sincrono* (vedi Figura 2.1) e il *modello timed asynchronous*.
Nel modello parzialmente sincrono vengono fatte delle assunzioni di sincronia, come assunzioni sulla sincronia dei clock e sulla velocità dei processi, oppure tramite impostazioni di un limite superiore sul ritardo massimo di consegna di un messaggio. Questo limite è spesso inizialmente sconosciuto e viene valutato via via tramite meccanismi di timeout. Un utile applicazione di questi sistemi è quella al problema del consenso: sfruttando queste assunzioni temporali è possibile risolvere il problema del consenso ([13]).
Il modello timed asynchronous ([14]) prevede una serie di assunzioni sul comportamento dei processi, dei canali di comunicazione e dei clock locali; questo modello può essere caratterizzato attraverso le seguenti specifiche:

1. i servizi definiti nel sistema sono *temporizzati*, possiamo quindi associare dei timeout per i servizi stessi;

2. la comunicazione fra processi è realizzata attraverso un servizio

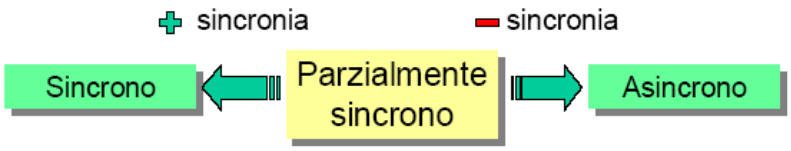

Figura 2.1: Relazione tra modello sincrono, asincrono e parzialmente sincrono

datagram non affidabile, soggetto a fallimento di tipo crash o timing;

3. i processi possono subire fallimento di tipo crash o timing;

4. il tasso di fallimenti delle comunicazioni e dei processi è limitato;

5. i processi hanno accesso a clock locali che si mantengono all'interno di un inviluppo lineare del tempo reale.

Le restrizioni appena elencate, anche se possono sembrare troppo specifiche, sono garantite da gran parte dei sistemi distribuiti attuali su LAN. La condizione numero 5 è garantita dai clock hardware dei sistemi reali; la condizione numero 1 può essere garantita anche su reti inaffidabili, attraverso l'imposizione di limiti temporali a livello più alto.
Un modello più recente è il modello *quasi sincrono* di Verissimo e Almeida ([15]), utilizzato per rappresentare sistemi real-time. In questi modelli abbiamo i failure detector per crash e i timing failure detector, una famiglia di failure detector specializzati nel rilevamento dei fallimenti di tipo timing.

2.4.2 I fallimenti

Un ulteriore parametro che consente la distinzione tra i vari sistemi è rappresentato dalla tipologia dei fallimenti cui possono essere soggetti sia i processi coinvolti che le comunicazioni.

Una prima distinzione dei fallimenti prevede la seguente classificazione ([7, 16]):

crash : il processo termina prematuramente e non esegue più alcuna azione, in pratica, interrompe l'esecuzione;

fallimento per omissione : si verifica se un processo omette una o più azioni, per esempio se non viene eseguito l'invio e/o la ricezione di messaggi;

fallimento di valore : viene fornito un valore diverso da quello previsto dalla sua specifica;

fallimento bizantino : il processo esibisce un comportamento qualsiasi, ad esempio cambia stato in modo arbitrario.

I modelli di fallimento sopra mensionati possono essere applicati sia a sistemi sincroni che asincroni. Tuttavia nel sistema sincrono esiste un ulteriore tipo di fallimento: il **timing failure**, vale a dire una violazione dei limiti temporali imposti sulla velocità di esecuzione o sulla deviazione del clock (si possono avere early o late timing failures in base al modo in cui i vincoli temporali non sono rispettati). Un processo soggetto ad un timing failure, dunque, è un processo che viola una delle assunzioni di sincronia; un simile processo piò guastarsi in uno o più dei seguenti modi:

1. commette un fallimento di omissione;
2. il suo clock locale eccede il bound specificato (clock failure);
3. viola il bound relativo al tempo richiesto per eseguire un passo elementare (performance failure).

I modelli di fallimento possono essere classificati in termini di severità; un modello A è più severo di un modello B se l'insieme di comportamenti di guasto permessi da B è un sottoinsieme proprio di quello costituito

dai comportamenti di guasto permessi da A. Quindi un algoritmo che tollera guasti di tipo A tollera anche guasti di tipo B.
I fallimenti per crash sono un sottoinsieme proprio dei fallimenti per omissione: un fallimento per crash si presenta quando, dopo la prima omissione nel rispondere, un componente omette sistematicamente di rispondere a tutti gli input successivi.
I fallimenti per omissione sono una sottoclasse propria dei fallimenti di timing: un fallimento per omissione può essere visto come il comportamento di un componente con un tempo di risposta infinito.
I fallimenti bizantini, infine, rappresentano una categoria di fallimenti a sè e la gestione di questi comporta un notevole aumento della complessità del sistema.
Come già detto, gli stessi tipi di fallimento interessano le comunicazioni del sistema; in particolare, la classificazione è la seguente:

crash : un canale smette di trasmettere messaggi; prima del crash si è comportato in modo corretto;

omissione : un canale omette in modo intermittente di trasportare i messaggi inviati attraverso di esso;

bizantino : un canale esibisce un qualsiasi comportamento.

Nel caso di sistemi sincroni, si hanno anche:

timing failure : un canale trasporta messaggi in modo più veloce o più lento rispetto alle sue specifiche.

2.4.3 Topologia della rete

La rete di comunicazione può essere modellata come un grafo, dove i nodi rappresentano i processi e gli archi i canali di comunicazione tra processi.
Questo modello si adatta sia a reti punto-punto che a reti broadcast.

Assunzioni più precise sulla topologia della rete possono essere fatte quando vengono considerati particolari problemi.
Spesso, comunque, l'interesse è rivolto alle reti in cui si utilizzano messaggi di tipo diverso:

- *messaggi unicast*: questo tipo di messaggi hanno una sorgente ed un'unica destinazione;
- *messaggi broadcast*: possono avere come destinazione tutti i processi della rete o di una particolare sottorete
- *messaggi multicast*: possono avere come destinazione insiemi qualunque di processi.

2.4.4 Processi deterministici e random

Un qualsiasi processo può essere modellato con un automa a stati finiti o infiniti e l'evoluzione del processo determina il passaggio da un stato all'altro. Il comportamento di un processo può essere *deterministico* o *random*:

- la relazione di transizione di stato di un processo deterministico determina in modo unico lo stato risultante dall'esecuzione di una computazione;
- in un processo random, al contrario, l'esecuzione di una computazione porta in uno stato che fa parte di un insieme di possibili stati e ad ogni possibile transizione è associata una certa probabilità.

La caratteristica dei processi di essere deterministici o meno influenza molto la possibilità di trovare o meno una soluzione a molti problemi: alcuni problemi non risolvibili con processi deterministici sono stati risolti attraverso l'utilizzo di processi con comportamento casuale; in [17] ad esempio, abbiamo una soluzione al problema del consenso per sistemi distribuiti asincroni, tramite l'uso di processi casuali.

2.5 Proprietà dei sistemi distribuiti

Le proprietà dei sistemi distribuiti sono di solito suddivise in due classi: proprietà di *safety* e proprietà di *liveness*.
Le proprietà di safety specificano la *correttezza* del sistema: cosiderando le varie situazioni in cui un sistema può trovarsi, gli stati del sistema, le proprietà di safety specificano quali sono gli stati che il sistema non dovrebbe mai raggiungere; questi stati corrispondono a situazioni pericolose sia per il sistema che per l'ambiente.
Le proprietà di liveness sono legate al *progresso* del sistema in esame: specificano proprietà che prima o poi si devono verificare.
Le proprietà di liveness e di safety si complementano nella specifica del sistema.
Un'altra classe di proprietà, utile nella specifica del comportamento di alcuni tipi di sistema, è la classe delle proprietà di *timeliness*, essa comprende tutte le proprietà che devono valere a un certo istante di tempo reale. Questo tipo di proprietà è utile nella specifica del comportamento dei sistemi *real-time*.

2.6 Tolleranza ai guasti nei sistemi distribuiti

Come accennato all'inizio di questo capitolo, uno dei motivi che ha portato allo sviluppo dei sistemi distribuiti è la possibilità di costruire sistemi tolleranti ai guasti grazi all'uso della modularità e della ridondana. I sistemi distribuiti, e quindi i sistemi distribuiti tolleranti ai guasti, sono costituiti di nodi, reti e componenti software; la tolleranza ai guasti nei sistemi distribuiti viene ottenuta attraverso l'uso di opportuni protocolli che governano le interazioni fra le componenti.
Un'architettura distribuita modulare permette di ottenere una *dependability incrementale*: attraverso la sostituzione delle componenti più

fragili (questo equivale a prevenire i guasti), incrementando il numero di repliche di ogni componente o rendendo le singole repliche resistenti a tipologie di guasti più severi (questo equivale a rendere il sistema più tollerante ai guasti). É fondamentale però ricordare che l'utilizzo di architetture semplici e principi di design rigorosi sono essenziali nella costruzione di sistemi distribuiti altamente dependable: la complessità è potenziale causa di guasti del sistema, maggiore è la complessità del sistema ideato più alta è la probabilità di non aver considerato tutte le possibili interazioni fra i componenti del sistema. La tolleranza ai guasti nasconde dunque un paradosso: per creare sistemi di questo tipo è spesso necessario aumentare la complessità di design del sistema stesso, complessità che può portare a una maggiore presenza di possibili guasti nel sistema.
In questo paragrafo presenterò alcuni protocolli fondamentali utilizzati nella costruzione di sistemi distribuiti tolleranti ai guasti.

2.6.1 Comunicazione tollerante ai guasti

Uno degli scenari più frequenti nei sistemi distribuiti è la comunicazione tra i vari processi attraverso l'invio di messaggi, con destinatari singoli o multipli. La comunicazione tra processi deve essere garantita nonostante l'occorrenza di fallimenti dei canali di comunicazione e dei processi che fanno parte del sistema. Vediamo i tipi principali di fallimento che occorre gestire in questi casi e i possibili comportamenti da adottare.

1. Per gestire i fallimenti per omissione possiamo utilizzare tecniche per il *mascheramento dell'errore* o il *recupero dell'errore*.
   Il mascheramento può avvenire o tramite ridondanza spaziale, ossia utilizzando più canali di comunicazione, o tramite ridondanza temporale, ripetendo più volte l'invio del messaggio. Questo tipo di approccio è utile nel caso in cui il grado di omissione[6] sia pic-

[6]Il grado di omissione rappresenta il massimo numero di omissioni successive.

colo, e quindi la possibilità di effettuare un veloce ripristino della situazione di fallimento compensa la perdita di banda disponibile. Il recupero dell'errore, invece, è basato sull'uso di messaggi di *acknowledgement* (*ack*) e di timeout. Se si decide di usare l'ack positivo, il messaggio di ack verrà spedito alla ricezione di ogni messaggio, se il mittente del messaggio iniziale non riceve l'ack entro un certo timeout questo provvederà a rinviare il messaggio. Se si utilizza l'ack negativo (o *nack*), il destinatario invia un nack al momento del rilevamento della perdita del messaggio, chiedendo al mittente il rinvio del messaggio.
Le tecniche per il recupero dell'errore, rispetto a quelle per il mascheramento dell'errore, offrono un utilizzo migliore delle risorse disponibili: la ritrasmissione del messaggio avviene soltanto nel caso in cui un'omissione è realmente avvenuta.

2. Tollerare guasti di valore significa garantire che se il messaggio è stato corrotto durante la trasmissione, questo venga rilevato come tale. Per raggiungere questo scopo si utilizzano meccanismi di *firma* o di *checksum*, basati su apposite funzioni matematiche che permettono il rilevamento delle modifiche sul messaggio. Se il messaggio viene riconosciuto come corrotto, può essere eliminato: trasformiamo così il guasto di valore in un guasto di omissione. Ci possiamo però trovare in una situazione simile: un mittente produce l'errore di valore prima del calcolo del checksum, in questo caso il messaggio passa il controllo del checksum pur essendo errato. Per ovviare a situazioni simili è necessario disporre di ridondanza spaziale, ossia avere diverse sorgenti che devono calcolare lo stesso valore logico.

3. Un'ulteriore situazione in cui ci potremmo trovare è il fallimento del mittente. Nel caso di comunicazione punto-a-punto, il problema si trasforma in un problema di *failure detection* (affronteremo

più avanti questo argomento); infatti, il rilevamento del fallimento del mittente consente al destinatario di non rimanere bloccato in attesa di un messaggio. Se invece la comunicazione è di tipo broadcast o multicast, il fallimento del mittente potrebbe lasciare il sistema in uno stato inconsistente, dipende dall'affidabilità della rete. É possibile distinguere tre livelli di affidabilità nelle comunicazioni multicast (Figura 3):

Multicast inaffidabile : è la forma più semplice da realizzare ma anche la più debole; non si effettua nessun tentativo di superare i fallimenti possibili dei collegamenti. Esempio di multicast inaffidabile è quello fornito dal protocollo UDP; non è opportuno usare un meccanismo di questo tipo per costruire applicazioni che permettono la tolleranza ai guasti, ma può essere utilizzato in applicazioni in cui interessa soltanto la bassa latenza, come applicazioni di videoconferenza o altre applicazioni multimediali, con client multipli, su Internet.

Multicast di tipo best-effort : la responsabilità della consegna del messaggio è affidata al mittente, è il mittente che deve effettuare delle operazioni (come la ripetizione dell'invio del messaggio) per garantire la consegna; se il mittente fallisce il protocollo non garantisce l'affidabilità.

Multicast affidabile : i partecipanti si coordinano in modo da garantire che il messaggio sia consegnato a tutti i destinatari corretti. Per ottenere un multicast affidabile si potrebbero utilizzare i meccanismi di mascheramento e recupero dell'errore. Nel primo caso, tutti i destinatari ritrasmettono il messaggio non appena lo ricevono; in questo modo, anche se il mittente fallisce, i destinatari mantengono una copia del messaggio che verrà trasmesso agli altri in caso di rilevamento del fallimento del mittente.

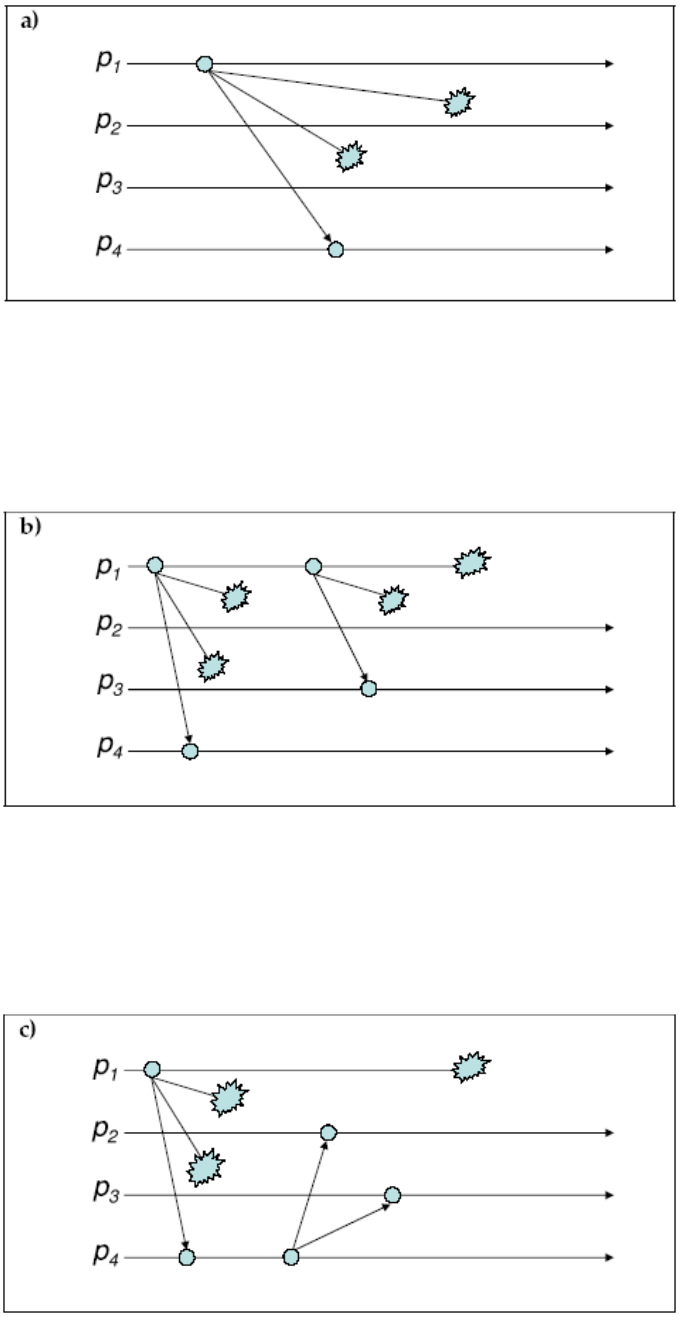

Figura 2.2: Tipi di multicast: a) inaffidabile, b) best-effort, c) affidabile

Perchè le tecniche di tolleranza ai guasti viste in questo paragrafo funzionino, spesso è necessario garantire proprietà di ordinamento nella consegna dei messaggi. I metodi principali di ordinamento dei messaggi sono:

Ordinamento FIFO : i messaggi spediti dallo stesso sender sono consegnati a tutti i destinatari nello stesso ordine.

Ordinamento causale : questo ordinamento è definito a partire dalla relazione *avvenuto prima* ([8, 18]); diciamo che un evento *a precede* b e scriviamo $a \rightarrow b$, quando:

- a e b sono eventi locali allo stesso processo e a avviene prima di b;
- a è l'evento relativo alla spedizione di un messaggio e b è l'evento relativo alla ricezione dello stesso messaggio;
- a e b appartengono alla chiusura transitiva della relazione definita con le due regole precedenti.

Se per due eventi, a e b, non vale nè $a \rightarrow b$, nè $b \rightarrow a$, gli eventi si dicono *concorrenti*.
Dunque, l'ordinamento causale nella consegna dei messaggi vale se per ogni coppia di messaggi m_1 e m_2, spediti rispettivamente da p e q allo stesso destinatario r, si verifica che:

$$send_p(m_1) \rightarrow send_q(m_2) \Rightarrow deliver_r(m_1) \rightarrow deliver_r(m_2)$$

Ordinamento totale : ogni coppia di messaggi, spediti da ogni coppia di partecipanti, sono consegnati nello stesso ordine a tutti i destinatari.

Garantire richieste di ordinamento più stringenti è sempre più difficile, e la difficoltà aumenta ancora considerando classi di fallimenti più severe.

2.6.2 La failure detection

Uno degli approcci fondamentali nella costruzione di sistemi dependable è il *rilevamento degli errori* e il successivo *ripristino*. Per costruire sistemi tolleranti ai guasti, è importante riuscire a rilevare un fallimento di un componente del sistema; un errore del sistema, infatti, si presenta a partire da un fallimento che si è verificato in qualche componente. La *failure detection* ([19]) permette sia di rilevare, scollegare e riparare il componente guasto che di aumentare la performance del sistema stesso: se un componente è guasto, i tentativi di comunicare con esso portano soltanto ad un inutile spreco di risorse.
Un failure detection è come un oracolo distribuito che mantiene costantemente una visione, più o meno corretta, dello stato dei componenti del sistema. In genere, ogni processo p_i ha accesso ad un modulo locale FD_i che monitora gli altri processi nel sistema e mantiene l'insieme di quelli che *sospetta* essere guasti. Ogni processo periodicamente consulta il modulo locale per avere informazioni sullo stato degli altri processi del sistema; inoltre, i moduli FD si scambiano informazioni sui processi sospettati. La realizzazione di questi moduli si basa spesso su meccanismi di heartbeat per decidere se sospettare o meno un certo processo. In genere i moduli FD sono *inaffidabili*, possono cioè sospettare processi attivi o non sospettare processi guasti; perciò, l'informazione fornita da un modulo di failure detector FD_i può non essere corretta, ad esempio viene sospettato un processo non guasto, e può essere inconsistente, ad esempio FD_i sospetta al tempo t un processo p_k mentre, nello stesso istante di tempo t, un altro modulo FD_j non sospetta p_k.
Un failure detector è caratterizzato, a livello astratto, da due proprietà:

1. proprietà di *completezza*: i processi guasti nel sistema sono sospettati;

2. proprietà di *accuratezza*: i processi corretti non sono sospettati.

Secondo la classificazione proposta in [19], possiamo avere due livelli di completezza:

- **Strong Completeness**: ogni processo guasto alla fine è sempre sospettato da *tutti* i processi corretti.
- **Weak Completeness**: ogni processo guasto alla fine è sempre sospettato da *qualche* processo corretto.

Per ciò che riguarda l'accuratezza, abbiamo quattro livelli:

- **Strong Accuracy**: *nessun* processo corretto è mai sospettato da un altro processo corretto.
- **Weak Accuracy**: *qualche* processo corretto non è mai sospettato da un altro processo corretto.
- **Eventual Strong Accuracy**: esiste un tempo t dopo il quale *nessun* processo corretto è mai sospettato da un altro processo corretto.
- **Eventual Weak Accuracy**: esiste un tempo t dopo il quale *qualche* processo corretto non è mai sospettato da un processo corretto.

	SA	WA	ESA	EWA
SC	P	S	$\diamond P$	$\diamond S$
WC	Q	W	$\diamond Q$	$\diamond W$

Tabella 2.1: Classi di failure detector

Le otto combinazioni (Tabella 2.1) di completezza e accuratezza definiscono altrettante classi di failure detector. In realtà queste otto classi di FD non sono legate da relazione "stretta" di maggiore potere

discriminatorio: è stata dimostrata ([19]) infatti, l'equivalenza fra Weak e Strong Completeness. In [20] viene inoltre dimostrato che $\diamond W$ è il failure detector più debole che permette di realizzare il consenso.
Una classe particolare di failure detector è il *timing failure detector*: un meccanismo per il rilevamento di fallimenti di tipo timing; vedremo nel Capitolo 6 le caratteristiche di questi failure detector.

2.7 Analisi dei sistemi distribuiti

Nel capitolo precedente, ho illustrato l'importanza del processo di validazione di un sistema, in questo paragrafo presento le principali tecniche utilizzate per effettuare un'analisi quantitativa di sistemi distribuiti.
Le tecniche utilizzate per la valutazione di dependability, performability e performance, possono essere classificate in: *tecniche modellistiche* e *tecniche basate su misure*.

2.7.1 Tecniche modellistiche

Le tecniche che fanno parte di questa categoria possono essere distinte in *analitiche* e *tecniche simulative*, entrambe basate sulla costruzione di un modello del sistema e delle sue componenti. Il modello rappresenta quindi le nostre assunzioni sul comportamento del sistema da analizzare.
Nei modelli analitici le componenti del sistema sono rappresentate attraverso *variabili di stato* e *parametri*, le loro interazioni sono rappresentate attraverso *relazioni*. Nei modelli simulativi, invece, è necessario utilizzare un programma dedicato, il *simulatore*, che permette la rappresentazione dell'evoluzione temporale del sistema e che fornisce una stima delle misure di interesse, in pratica, in questi modelli, si *riproduce* il comportamento dinamico del sistema nel tempo.
I sistemi sotto analisi si differenziano in due classi principali: i *sistemi discreti* e i *sistemi continui*. Nei sistemi discreti, una o più quantità

del sistema cambiano istantaneamente, ma soltanto in istanti di tempo separati; un esempio di tali sistemi è la coda dei pacchetti di un computer in rete: lo stato del sistema cambia in istanti di tempo separati, a causa dell'uscita di un pacchetto dalla coda o dell'arrivo di un nuovo pacchetto. Nei sistemi continui, invece, le quantità possono cambiare con continuità nel tempo; un esempio di tali sistemi è un treno, che ha velocità e posizione rispetto ad una stazione che variano con continuità nel tempo. Non è strano che un sistema sia allo stesso tempo discreto e continuo, dipende infatti dalle quantità che si stanno analizzando e non dal sistema stesso.
Anche i modelli possono essere classificati, in particolare distinguiamo tra modelli *statici* e *dinamici*, *deterministici* e *stocastici*, *continui* e *discreti*.
Un modello statico permette la rappresentazione di un sistema in un preciso istante di tempo; in un modello dinamico, invece, rappresentiamo l'evoluzione temporale del sistema, ad esempio attraverso l'interazione tra i componenti.
Un modello è deterministico quando nessuno dei suoi componenti presenta comportamenti probabilistici; se uno o più componenti presentano un comportamento probabilistico, il modello si dice stocastico.
La distinzione tra modelli continui e discreti, infine, si basa sulla rappresentazione delle quantità di interesse (e non sulla quantità), ossia la distinzione si basa sul fatto di voler rappresentare la quantità come variabile con continuità o discretamente nel tempo.
I modelli più frequentemente utilizzati per l'analisi di sistemi informatici sono quelli dinamici, stocastici e a stati discreti.

Tecniche analitiche

Esistono varie tecniche di modellazione analitica e in generale si possono utilizzare combinazioni appropriate di queste tecniche per la specifica, la costruzione e la soluzione del modello. Distinguiamo comunque due

gruppi di metodi analitici:

- I *metodi combinatori*, più semplici e intuitivi nella fase di costruzione del modello e di soluzione, ma sono inadeguati nella rappresentazione delle dipendenze tra i diversi componenti del sistema.
  Tra i principali metodi combinatori, utilizzati per la costruzione di modelli di dependability e performability, ricordiamo i *fault trees* ed i *reliability block diagrams*.

- I *metodi basati su spazio degli stati* sono molto più potenti, ma più difficili da costruire e da risolvere. Tra questi metodi ricordiamo le *catene di Markov*, i *modelli a rete di code* e le *reti di Petri* con le sue estenzioni[7].
  Una catena di Markov è un processo Markov con uno spazio degli stati discreto; un processo di Markov è un processo random che può essere informalmente così definito: dato il valore, $X(t)$, di un processo di Markov X al tempo t, il comportamento futuro di X può essere completamente descritto in termini di $X(t)$. Le reti di Petri e le sue estensioni, sono state sviluppate per gestire e facilitare la generazione automatica e soluzione di sistemi stocastici Markoviani. I modelli a rete di coda permettono una rappresentazione più dettagliata dei sistemi composti da un insieme di risorse interconnesse.

Esistono appositi strumenti che supportano i formalismi per la definizione dei modelli descritti, essi permettono la costruzione e soluzione di questi modelli; gran parte di questi tool forniscono sia la possibilità di ricavare soluzioni analitiche del modello (imponendo delle condizioni sui formalismi utilizzati), sia di eseguire simulazioni del modello stesso. Tra gli strumenti maggiormente utilizzati ricordiamo *UltraSAN* ([21]), *DEEM* e *Mobius* ([22]).

[7]Reti di Petri *stocastiche*, *stocastiche generalizzate*, *deterministiche/stocastichee* e le *Stochastic Activity Networks*.

Tecniche simulative

Queste tecniche, principalmente utilizzate per predire performance e dependability di sistemi prima della loro implementazione, possono essere classificate in: tecniche basate su *emulazione*, tecniche *guidate da tracce* e tecniche *ad eventi discreti*.

- Le tecniche basate su emulazione permettono di ottenere risultati molto dettagliati sul comportamento del sistema oggetto di studio. Queste tecniche si basano su sistemi esistenti per rappresentare il comportamento di sistemi simili, che vengono quindi *emulati*. Un esempio di questa tecnica è l'utilizzo di un processore di una certa famiglia per emulare un altro processore.

- Le tecniche guidate da tracce utilizzano un *simulatore* e un *generatore di eventi*; il generatore produce una traccia di esecuzione che viene utilizzata come input del simulatore che, a sua volta, fornisce delle stime delle quantità di interesse.
  Il principale vantaggio di queste tecniche è l'accuratezza dei risultati, lo svantaggio è che, utilizzando tracce ottenute attraverso modelli dedicati, abbiamo input predeterminati e quindi poca varianza nei risultati ottenuti.

- Le tecniche di simulazione ad eventi discreti sono utilizzate per simulare il comportamento di sistemi di cui possiamo ottenere una rappresentazione sotto forma di modelli ad eventi discreti. Tali tecniche sono utilizzate per lo studio di modelli di reti di code e nella simulazione di reti di Petri. In una simulazione di questo tipo l'evoluzione del sistema nel tempo viene rappresentata attraverso il cambiamento delle variabili di stato del modello; l'evoluzione avviene attraverso la simulazione degli eventi, che modificano lo stato

Anche per queste tecniche esistono molti strumenti che aiutano nella preparazione di un simulatore ed esistono ambienti di simulazione specifici per singoli problemi, come il framework Neko (Capitolo 3), utilizzato per la simulazione e la prototipazione di algoritmi e sistemi distribuiti.
I motivi per cui scegliere la simulazione invece della modellazione analitica sono la flessibilità e la generalità dei modelli risolvibili tramite simulazione; d'altra parte però occorre tener presente gli svantaggi dovuti al costo e alla performance: simulazioni di sistemi su larga scala, con un livello elevato di dettagli, o di cui vogliamo ricavare stime sul verificarsi di eventi a bassa probabilità, richiedono costi di sviluppo e tempi di esecuzione spesso troppo elevati.

2.7.2 Tecniche basate su misure

Queste tecniche si basano sull'utilizzo di appositi moduli, software e hardware, chiamati *monitor*, per osservare l'attività del sistema sotto esame durante la sua esecuzione; in pratica, i monitor hanno il compito di acquisire dati, analizzarli e fornire in output i risultati ottenuti.
I vantaggi ottenuti dall'uso di tali tecniche sono molteplici, esse permettono la caratterizzazione qualitativa e quantitativa del carico di lavoro, delle performance e dell'utilizzo delle risorse.
In genere i monitor si suddividono in tre classi: monitor *software*, *hardware* e *ibridi*. I primi sono appositi processi in esecuzione in parallelo con il sistema, permettono il rilevamento di particolari stati del sistema stesso. I monitor hardware sono particolari dispositivi elettronici collegati al sistema, utilizzano il segnale di output del sistema per caratterizzare il fenomeno che si intende osservare. I monitor ibridi sono composti da più moduli, sia software che hardware, interconnessi.
Una distinzione interessante dei monitor può essere fatta in base alla modalità di attivazione:

attivazione al verificarsi di particolari eventi: in questo caso si parla di *tracciamento* e il monitor è detto *guidato dagli eventi*; il

verificarsi di un evento attiva il monitor, che cattura i dati sullo stato del sistema; lo svantaggio di questo metodo è la grande quantità di dati da gestire e il maggiore overhead. A causa di un overhead elevato possiamo ottenere risultati delle misure non accurati, per questo motivo la riduzione dell'overhead dei monitor è uno dei principali obiettivi nella loro fase di design. É quindi necessario in questo caso utilizzare delle procedure di filtraggio dei dati raccolti, la cui accuratezza è fondamentale per ottenere risultati significativi.

attivazione rispetto al tempo: in questo caso, il monitor viene riattivato allo scadere di ogni intervallo di tempo Δt e l'operazione di raccolta dei dati è detta *campionamento*; in questo caso lo svantaggio è legato alla difficoltà di definire un intervallo Δt appropriato: intervalli troppo brevi possono portare a troppi dati, molti dei quali poco significativi, mentre intervalli troppo ampi possono portare a risultati poco precisi o incompleti.

Capitolo 3

Neko e NekoStat

Neko è una piattaforma di comunicazione utilizzabile per l'analisi di algoritmi distribuiti attraverso simulazione e misure ([23, 24]); un'unica implementazione di un algoritmo può essere utilizzata sia per la simulazione che per l'esecuzione su una rete reale.
Per comprendere l'utilità del framework Neko basta ricordare che gli approcci utilizzabili per la valutazione della dependability, della performability e della performance di algoritmi distribuiti sono tre, quello *analitico*, quello *simulativo* e quello basato su *misure* e che, per aumentare la confidenza e l'accuratezza dei risultati ottenuti, è opportuno confrontare i risultati ricavati da almeno due di questi approcci; l'uso di Neko dunque, fornendo risultati ottenuti da un'analisi simulativa e da un'analisi basata su misure, permette di aumentare l'attendibilità dei risultati ottenuti.
Inoltre, grazie all'utilizzo della stessa implementazione per la simulazione e per l'esecuzione reale, si hanno tempi di sviluppo e testing per un algoritmo inferiori rispetto a quelli ottenuti da un approccio tradizionale che prevede due implementazioni diverse per lo stesso algoritmo, realizzate solitamente in tempi diversi dato che spesso si utilizzano linguaggi diversi per i due tipi di analisi, ad esempio SIMULA per la simulazione e C++ per l'implementazione finale dell'algoritmo.

Per la gestione delle simulazioni dei sistemi distribuiti Neko utilizza un simulatore ad eventi discreti, tuttavia, predisponendo le opportune librerie di connessione che permettono la comunicazione tra il simulatore e Neko, è possibile utilizzare anche altri simulatori, come ad esempio SimJava.
Neko è stato realizzato al *Distribuited Systems Lab* dell'istituto *EPFL* di Lausanne, da Peter Urban e Xavier Defago, con la collaborazione del Prof. Andre Shiper.
In questo capitolo presenterò una breve descrizione dell'architettura e dell'utilizzo del framework che servirà a comprendere meglio il mio lavoro di tesi.

3.1 Architettura di Neko

L'architettura di Neko è suddivisa in tre componenti principali (Figura 3.1): le *applicazioni* (realizzate attraverso una struttura a livelli multipli detti *Layer*), i *NekoProcess* e le *reti*. A livello di applicazione, un insieme di m processi, numerati da *1* a m, comunicano attraverso un'interfaccia per il passaggio di messaggi: un processo *sender* inserisce un messaggio in rete (attraverso la primitiva asincrona **send**) e la rete consegna il messaggio al processo ricevente (attraverso la primitiva **deliver**).
Per ciò che riguarda la piattaforma di comunicazione in Neko, ci sono diverse alternative:

- lo sviluppatore ha la possibilità di istanziare una rete già prevista da Neko;
- si possono creare ed istanziare nuove tipologie di reti non previste in Neko;
- è possibile permettere lo scambio di tipologie diverse di messaggi attraverso tipi diversi di connessioni utilizzando in parallelo diverse reti.

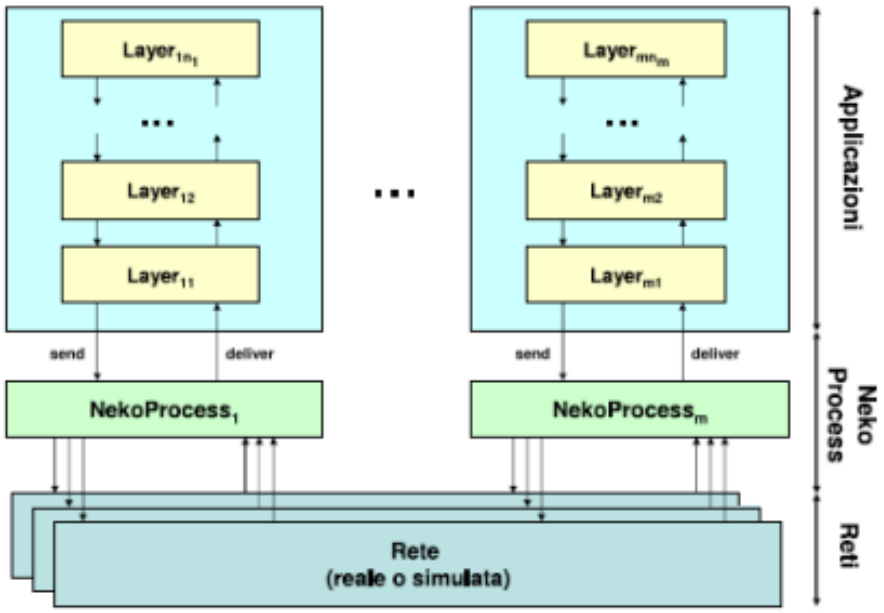

Figura 3.1: Architettura di Neko

Approfondiremo più avanti il discorso sulle reti.

3.1.1 I layer

I processi che compongono le applicazioni distribuite Neko sono realizzati attraverso *gerarchie di livelli*. I livelli comunicano attraverso scambi di messaggi che vengono passati da un livello più alto ad uno più basso attraverso la primitiva **send** e da un livello più basso ad uno più alto attraverso la primitiva **deliver**.
Distinguiamo i layer in *passivi* e *attivi*, rappresentati entrambi in Figura 3.2; i layer passivi permettono soltanto il trasporto di messaggi attraverso la gerarchia di livelli, con l'uso delle primitive **send** e **deliver**.
I layer attivi, al contrario, dispongono di un proprio thread di controllo. I messaggi ricevuti dal livello sottostante attraverso la primitiva **deliver** sono inseriti in una coda FIFO; durante l'esecuzione del thread di controllo del layer, si può eseguire la primitiva **receive** che preleva dalla

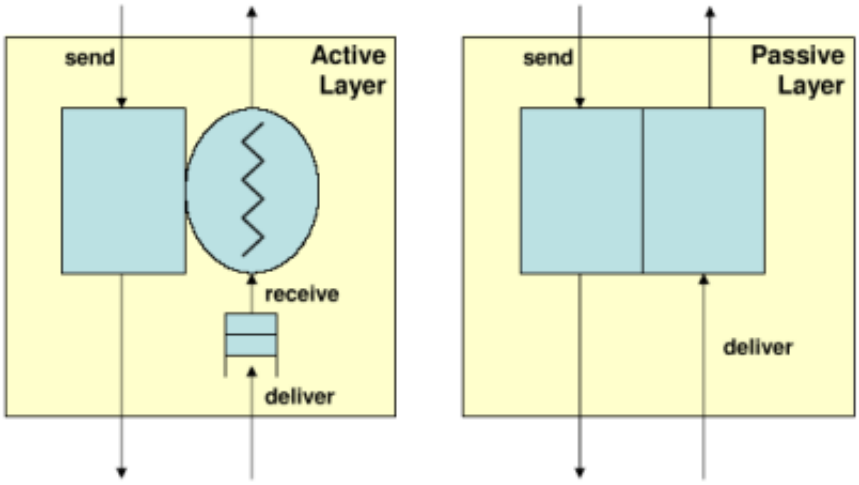

Figura 3.2: Layer attivi e passivi

coda un nuovo messaggio. La primitiva **receive** è bloccante: se non ci sono messaggi in coda il thread di controllo viene messo in attesa; esiste però una primitiva **receive** con timeout alla quale corrisponde una semantica non bloccante se si imposta un timeout nullo.
L'utilizzo della coda di messaggi FIFO nei layer attivi non è obbligatorio: un layer attivo può fornire un metodo **deliver** diverso da quello standard che può essere richiamato dai layer sottostanti, proprio come avviene nei layer passivi.
Abbiamo parlato di gerarchia di layer ma ciò non significa che è obbligatorio creare applicazioni con i processi composti come gerarchie di layer; i layer possono essere combinati in modi diversi e possono interagire attravero metodi diversi da quelli predefiniti.

3.1.2 I NekoProcess

Tutti i processi che compongono un'applicazione distribuita Neko hanno associato un oggetto NekoProcess che risiede tra la gerarchia di livelli dell'applicazione e le reti sottostanti. Vediamo i ruoli del NekoProcess:

- innanzitutto mantiene informazioni comuni utili ai layer sovrastanti, tipi di informazioni sono l'indirizzo di rete del computer che ospita il processo Neko oppure l'identificatore numerico associato al processo stesso; questo permette per esempio, la realizzazione di tecniche di programmazione come SPMD (Single Program Mltiple Data) ovvero lo stesso programma viene eseguito su diversi processi e, attraverso le informazioni ottenute dal NekoProcess, si esegue la parte di codice opportuna per il particolare processo.

- l'implementazione di servizi generali utili, come il log dei messaggi spediti e ricevuti da un processo;

- inoltre, nel caso di utilizzo di reti multiple, il NekoProcess raccoglie ed invia i messaggi nelle reti opportune.

3.1.3 Le reti

Il livello più basso nell'architettura di un'applicazione è formato dalle reti Neko, possiamo distinguere due tipologie principali di reti, quelle reali e quelle simulate. Diverse tipologie di reti sono predefinite in Neko ma è possibile definirne di nuove, sia per utilizzare reti reali proprietarie che per definire nuove tipologie di reti simulate.
Le reti reali sono costruite in Neko a partire dalle socket Java, oppure attraverso l'utilizzo di apposite librerie esterne per reti proprietarie. L'invio e la ricezione di un NekoMessage nelle reti reali avviene attraverso l'uso della serializzazione del linguaggio Java: la serializzazione permette di rappresentare il contenuto di un NekoMessage durante il tragitto dal processo mittente al destinatario.
Le reti reali predefinite in Neko sono le seguenti:

TCPNetwork: costruita sullo stack TCP/IP, garantisce un servizio di consegna dei messaggi affidabile. Utilizzando questa rete, all'avvio dell'applicazione viene stabilita, tra ogni coppia di processi, una connessione TCP.

UDPNetwork: costruita sullo stack UDP/IP, fornisce un servizio di consegna di messaggi inaffidabile.

MulticastNetwork: può essere utilizzata per la spedizione di datagrammi UDP di tipo multicast (multicast inaffidabile).

PMNetwork: utilizza la libreria PM per la comunicazione a bassa latenza su architetture cluster (bypassando lo stack TCP/IP, si veda [25]).

EnsembleNetwork: fornisce un servizio di multicast affidabile per IP tramite integrazione del framework di group communication Ensemble ([26]).

Come abbiamo anticipato, in Neko sono presenti alcune reti simulate predefinite, tra queste citiamo Ethernet, FDDI, CSMA-DCR e una rete con tempi di trasmissione distribuiti esponenzialmente, utile per il debugging di applicazioni distribuite: il comportamento meno deterministico rispetto alle reti reali esercita maggiormente l'algoritmo in molte differenti condizioni di utilizzo. L'integrazione di nuove reti simulate può essere realizzata facilmente: si tratta di definire un modello di comportamento della rete e rappresentare questo modello sotto forma di NekoNetwork, in cui si definisce il comportamento della rete attraverso l'implementazione dei metodi di **send** e **deliver**.
Ciò che viaggia nella rete sono messaggi, detti NekoMessage, essi possono essere di tipo *unicast* o *multicast*; ogni messaggio è caratterizzato da un contenuto (che può essere un qualunque oggetto Java) e da un *header*, che contiene le seguenti informazioni:

(Sorgente, Destinazione): informazioni necessarie per l'indirizzamento del messaggio, sotto forma di indirizzi sul processo mittente e sul processo destinatario; gli indirizzi sono espressi attraverso un identificatore intero associato al processo, ad esempio in Figura 3.2 sono i numeri da *1* a *m*.

Rete: se utilizziamo diverse reti in parallelo, ogni messaggio contiene l'informazione sulla rete che deve essere usata per la trasmissione.

Tipo di messaggio: ogni messaggio è caratterizzato da un tipo, lo sviluppatore definisce nell'applicazione Neko i tipi di messaggi scambiati e associa ad ogni tipo un opportuno identificatore intero; questo tipo è utile per distinguere messaggi appartenenti a differenti protocolli.

3.2 Simulazione ed esecuzione reale: similitudini e differenze

In questo paragrafo analizziamo similitudini e differenze fra i due tipi di esecuzione possibili per un'applicazione distribuita Neko: esecuzione in simulazione o su una rete reale.

3.2.1 Configurazione, startup e shutdown di un'applicazione Neko

La prima cosa da fare quando si intende utilizzare Neko per l'esecuzione di un'applicazione è la sua configurazione che può essere effettuata attraverso un singolo file, contenente le informazioni utili per istanziare i vari processi.
Nel caso di esecuzione su una rete reale si ha un *processo master*, che coordina l'esecuzione, e *m-1 processi slave*. Il master è il processo che ha a disposizione il file di configurazione dell'applicazione, che deve contenere le infomazioni sull'indirizzameno dei processi slave (sotto forma di un indirizzo che dipende dalla rete in uso); inoltre il processo master si occupa di contattare gli slave e di fornirgli le informazioni utili per il loro avvio.
Per non dover riattivare i singoli processi slave ad ogni nuova esecuzione

di un'applicazione, è possibile predisporre di appositi processi sempre in esecuzione (*slave factories*) sugli host, che attendono la comunicazione da parte di un master Neko per istanziare ed eseguire un nuovo slave. Nelle esecuzioni distribuite abbiamo quindi m Java Virtual Machine (JVM), in esecuzione solitamente su m host diversi, che comunicano attraverso la piattaforma fornita dal framework.
Nello startup di una simulazione i singoli processi sono eseguiti sotto forma di differenti thread in esecuzione all'interno della stessa JVM.
Per ciò che riguarda la terminazione di un'applicazione distribuita, Neko fornisce la funzione predefinita **shutdown**; questa funzione può essere richiamata da qualunque processo e porta alla terminazione di tutti i processi che fanno parte dell'applicazione distribuita. Si ha comunque la possibilità di definire funzioni di shutdown specifiche per l'applicazione.

3.2.2 Regole

Come abbiamo detto uno degli scopi principali di Neko è permettere che la stessa implementrazione di un'applicazione sia utilizzata sia in simulazione che in esecuzione reale. Occorre ricordare, però, che questi due approcci sono fondamentalmente differenti, perciò questo paragrafo elenca una serie di regole da seguire se vogliamo che la stessa applicazione sia eseguibile in simulazione e in esecuzione reale.

Nessuna variabile globale: nel caso di simulazione, i singoli layer appartenenti a differeti processi sono in esecuzione all'interno della stessa JVM e possono quindi utilizzare variabili globali per lo scambio di informazioni; l'uso di questo tipo di variabili va evitato se si intende utilizzare la stessa implementazione anche per l'esecuzione reale.

Thread: il modello di thread è differente per le simulazioni e per le esecuzioni reali, queste ultime possono utilizzare le classi thread predefinite nel linguaggio Java, mentre in simulazione è necessario

utilizzare le apposite classi fornite dal simulatore ad eventi discreti. In Neko è stata nascosta questa differenza tramite la classe NekoThread che deve essere utilizzata per istanziare nuovi thread che possono allora essere sia simulati che eseguiti.

3.3 Il pacchetto NekoStat

Neko semplifica notevolmente le fasi di progettazione, testing e analisi di algoritmi distribuiti, tuttavia non fornisce alcun supporto per la *valutazione di quantità* ricavabili dalla simulazione o dall'esecuzione reale di un algoritmo distribuito. In questo paragrafo presento il pacchetto NekoStat ([1]) che ha lo scopo di estendere il framework Neko in modo da uniformare e semplificare i passi necessari all'analisi quantitativa di algoritmi distribuiti.

3.3.1 Architettura

Seguendo la filosofia di Neko, il pacchetto NekoStat è stato realizzato cercando di rendere l'uso il più possibile comune ai due tipi di analisi (simulativa e prototipale); inoltre, lo sviluppatore che intende utilizzare NekoStat per ottenere delle misure dell'algoritmo sotto analisi, deve effettuare soltanto due modifiche al codice dell'applicazione Neko:

1. introdurre le chiamate al metodo **log(Evento)** dello **StatLogger**, nei punti in cui si vuole segnalare il verificarsi di un evento;

2. realizzare uno **StatHandler** che permetta la gestione degli eventi definiti nel codice e che quindi permetta di ricavarne le opportune quantità.

L'architettura di appicazioni NekoStat è mostrata in Figura 3.3 e Figura 3.4. Qui di seguito viene presentata un'analisi degli scopi e una breve spiegazione del funzionamento dei singoli componenti; una prima

suddivisione del pacchetto può essere fatta in una parte che si occupa di fornire un supporto alle analisi e una che fornisce gli strumenti matematici di supporto per la gestione delle quantità.

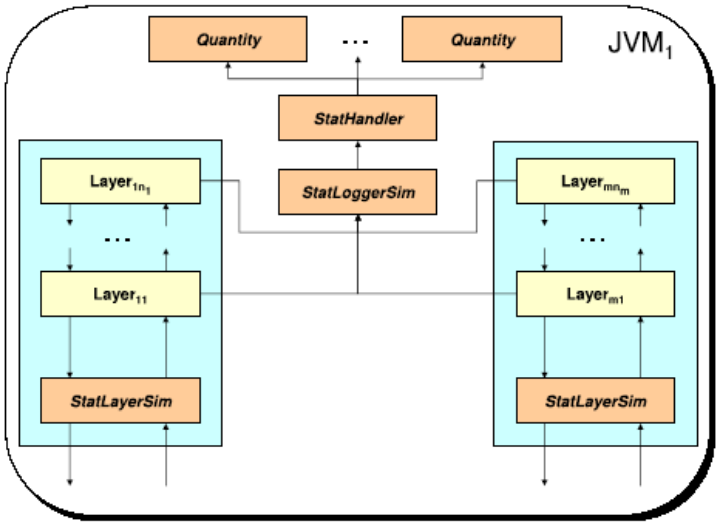

Figura 3.3: Architettura di un'applicazione NekoStat per una simulazione

3.3.2 Strumenti di supporto all'analisi

Strumenti comuni

Gli strumenti utilizzati sia nelle simulazioni che nelle esecuzioni distribuite sono:

- lo StatLogger;
- l'Event;
- lo StatHendler;
- lo StatInitializer;
- lo StatLayer;

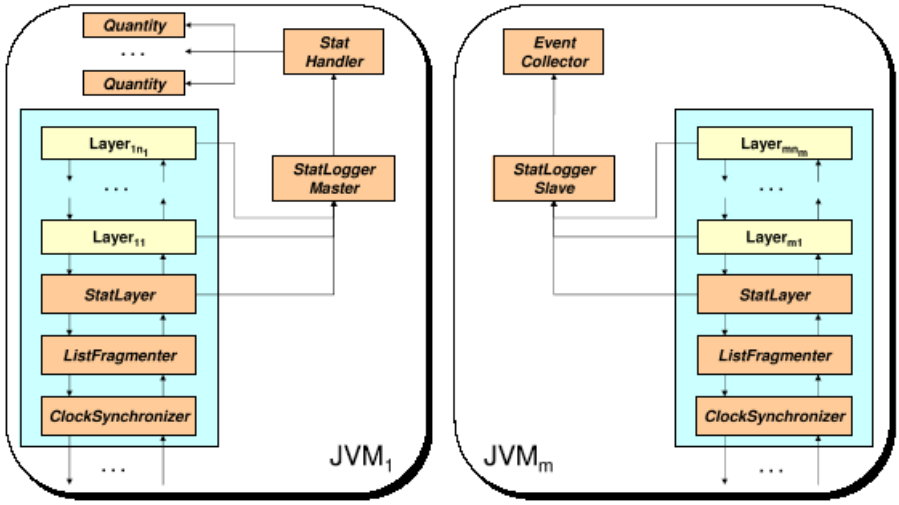

Figura 3.4: Architettura di un'applicazione NekoStat per un'esecuzione reale

Vediamo a cosa servono questi strumenti.

Lo **StatLogger** ha essenzialmente due ruoli:

nelle simulazioni riceve gli eventi dei differenti processi, richiama la funzione di gestione degli eventi dello **StatHandler** definito dall'utente e controlla se la simulazione può essere interrotta, ovvero se è stato raggiunto un grado di confidenza sufficiente sui risultati;

nelle esecuzioni reali si occupa della raccolta degli eventi dei singoli processi in un **EventCollector** e, al termine dell'esecuzione, invia al master gli eventi raccolti per la successiva fase di analisi.

In simulazione i processi sono simulati all'interno di un'unica JVM e quindi lo StatLogger è unico per il sistema, nel caso di esecuzione reale, invece, abbiamo uno StatLogger per ogni processo.
Il funzionamento dello StatLogger è molto diverso per esecuzioni e simulazioni, è per questo che in realtà si tratta solo di un'interfaccia,

che viene poi implementata dalle classi **StatLoggerSim** (per le simulazioni), **StatLoggerMaster** e **StatLoggerSlave** (rispettivamente per il master e per gli slave nelle esecuzioni reali).

L'**Event** è il contenitore delle informazioni degli eventi che avvengono nel sistema distribuito analizzato con NekoStat.
Ogni evento è caratterizzato da:

ID di processo: identificatore intero del processo dove è avvenuto l'evento;

Tempo: il tempo in cui è avvenuto l'evento;

Descrizione: una stringa che descrive il tipo di evento;

Contenuto: un oggetto Java che può essere usato per il trasporto di informazioni da un componente all'altro di NekoStat.

Il tempo associato all'evento può essere di due tipi: *tempo simulato*[1] o *tempo reale*.
Il tempo reale viene misurato in Neko in millisecondi, a partire dal tempo 0 (tempo in cui il processo viene attivato) del processo in cui è avvenuto l'evento.

Lo **StatHandler** ha il ruolo di gestire i singoli eventi definiti per il sistema in esame; esso traduce gli eventi del sistema in quantità.
Lo StatHandler deve essere realizzato dall'utente di NekoStat in quanto dipende sia dal tipo di applicazione che dall'insieme di misure che si intende ricavare con l'analisi.
Ogni StatHandler deve implementare il metodo **handle(Event)**, nel quale l'utente definisce come gestire i vari tipi di eventi utili per ricavare le quantità di interesse (si veda [1]). L'utente può inoltre definire

[1] usato in caso di simulazione.

le condizioni richieste per interrompere la simulazione implementando il metodo **shouldStop**, vedremo più avanti l'utilizzo delle condizioni di stop sulle quantità analizzate.
Nel pacchetto comunque è fornita un'implementazione standard di questa interfaccia, la classe **LoggingStatHandler**, utile per effettuare il debugging dell'applicazione: scegliendo questo StatHandler, infatti, gli eventi raccolti vengono inseriti in un file di log in modo da fornire la storia dell'esecuzione o della simulazione effettuata.

Lo **StatInitializer** predispone i layer e gli altri oggetti necessari per l'analisi. Questa classe ha il ruolo di preparare l'architettura di NekoStat come quella in Figura 3.3 e Figura 3.4.

Lo **StatLayer** si occupa della gestione dei messaggi di controllo di NekoStat, in particolare per la fase di terminazione dell'analisi. Il comportamento del layer è diverso per i due tipi di esecuzione. In simulazione esso si attiva alla ricezione del messaggio di richiesta della terminazione dell'analisi, in seguito all'arrivo di questo messaggio, si attiva la procedura di esportazione dei risultati dello **StatHandler** e termina la simulazione. Nel caso di esecuzione reale, invece, alla ricezione, da parte dello **StatLayer** del master, del messaggio di richiesta della terminazione dell'analisi, il layer invia un messaggio a tutti i processi sui quali è attivo uno **StatLayer** in modo che abbia inizio la procedura di invio degli eventi al master (Figura 3.5); l'invio degli eventi è gestito da un apposito layer **ListFragmenter**, vedremo più avanti il suo funzionamento.

Strumenti specifici

Esistono strumenti utilizzati specificamente per l'analisi di prototipi, vediamo quali sono e a cosa servono:

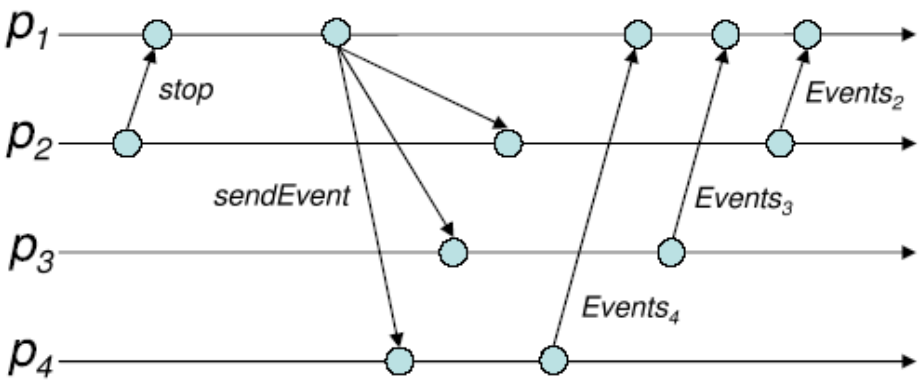

Figura 3.5: Fase finale di un'esecuzione distribuita

- EventCollector;
- ListFragmenter;
- ClockSynchronizer;

L'**EventCollector** ha il compito di raccogliere gli eventi locali ad ogni processo durante l'esecuzione reale distribuita; è grazie alla disponibilità dell'**EventCollector** di tutti i processi che lo **StatLogger** crea la storia dell'esecuzione distribuita. Al termine di un'applicazione NekoStat, gli **EventCollector** dei singoli processi vengono inviati al processo master, il quale, attraverso lo **StatHandler**, gestisce la sequenza di eventi avvenuti durante l'esecuzione. L'**EventCollector** mantiene la lista degli eventi avvenuti sul nodo utilizzando un'apposita struttura dati chiamata **SparseArrayList**, simile ad un **Vector**, vettore di oggetti predefinito in Java, ma con una diversa politica di ridimensionamento in modo da diminuire i tempi di inserimento di un nuovo elemento. L'inserimento in un SparseArrayList pieno, infatti, implica la creazione dello spazio per il nuovo elemento, a differenza dell'inserimento in un Vector pieno che implica un ridimensionamento della struttura al doppio della dimensione originale.

La **ListFragmenter** permette l'invio al master dell'**EventCollector** locale al termine dell'esecuzione, in maniera bufferizzata. Durante l'esecuzione della fase finale di gestione degli eventi, questo layer è utilizzato per inviare al master soltanto gli eventi di cui necessita in quel momento per l'analisi.

Il **ClockSynchronizer** fornisce un metodo di sincronizzazione a tutti i processi. Neko è organizzato in modo da permettere l'utilizzo di diversi meccanismi di sincronizzazione dei clock, il metodo predefinito consiste nell'utilizzo di un meccanismo di sincronizzazione con il tempo reale esterno a Neko e di un layer che permetta la sincronizzazione dei clock dei vari processi all'avvio dell'applicazione. Ogni processo ha accesso ad un clock Neko, che ha granularità di 1 ms e che è settato a 0 all'avvio del processo. I clock Neko possono presentare una fase causata dai differenti istanti di avvio dei singoli processi; il layer predefinito di sincronizzazione tenta di azzerare questa fase imponendo un'unica origine temporale a tutti i processi. L'algoritmo di sincronizzazione è di tipo *master-slave*, ossia, il clock del master è utilizzato come clock di riferimento (Figura 3.6).

3.3.3 Strumenti di supporto all'analisi statistica

NekoStat fornisce stumenti di supporto all'analisi statistica, attraverso l'uso di apposite classi per la gesione di quantità; esse sono estensioni di alcune classi disponibili nella libreria matematica Colt per Java ([27]), sviluppata al Cern di Ginevra con funzionalità più avanzate della libreria predefinita java.Math e con prestazioni confrontabili a quelle raggiungibili con programmi scritti in Fortran, C o C++.
In realtà, in NekoStat, le quantità sono gestibili e rappresentabili nel modo predefinito all'interno dello **StatHandler** specifico per l'applicazione, non c'è quindi alcun obbligo di utilizzo delle classi descritte in

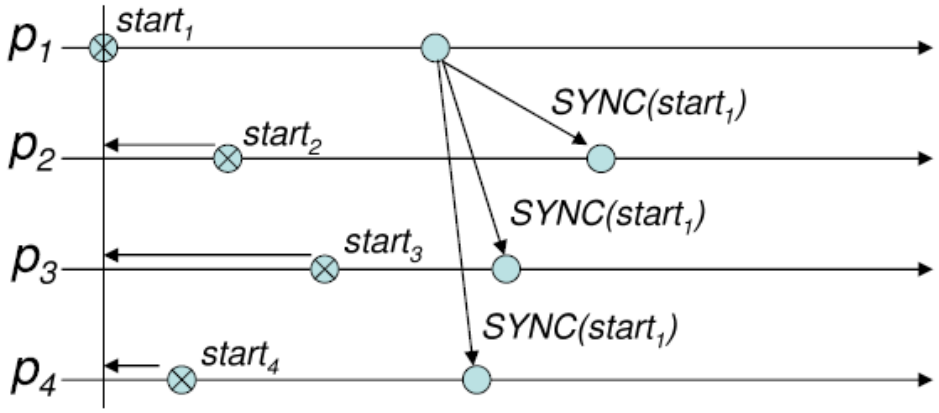

Figura 3.6: Sincronizzazione dell'origine dei tempi in un'esecuzione distribuita

questo paragrafo che sono comunque utili per effettuare molti tipi di analisi.
Queste classi hanno un'interfaccia comune, composta dai seguenti metodi:

- il metodo **add(double)** che permette l'inserimento nel contenitore del valore passato in input. Il contenitore contiene quei valori dai quali si ricavano i principali parametri statistici;
- i metodi come **size()**, **mean()**, **getMedian()**, **standardDeviation()**, **min()**, **max()**, utili per ottenere informazioni statistiche. La lista completa delle informazioni ricavabili da una quantità può essere trovata nella documentazione della libreria Colt ([28]).

In NekoStat sono state realizzate tre classi per la rappresentazione di quantità, esse si differenziano per differenti requisiti di memoria e per differenti funzionalità fornite:

QuantityOnlyStat Questo tipo di quantità permette soltanto la valutazione dei principali parametri statistici della quantità analizzata, tipo la media stimata, la varianza, la deviazione standard,

la mediana. I metodi usati in questa classe sono quelli della classe **hep.aida.StaticBin1d**([28]).

QuantityDataOnFile Essa è un'estensione della classe precedente, in questo caso le misure inserite nel contenitore vengono esportate su un file.

Quantity Questa è la classe che permette le capacità massime di analisi; oltre ai parametri statistici visti per le altre classi, permette di ricavare altre utili informazioni, tipo la covarianza rispetto ad un'altra **Quantity**. É possibile, inolte, effettuare la rimozione del transiente, sia iniziale che finale, e la rimozione degli outlier[2].
L'unico inconveniente derivante dall'uso di questa classe è dovuto al fatto che i valori inseriti nel contenitore associato alla **Quantity** permangono in memoria, l'occupazione di memoria può quindi crescere molto velocemente.
Questa classe è un'estensione della classe **hep.aida.DynamicBin1d**([28]).

Ad ognuna delle quantità sopra descritte sono associate diverse possibili *condizioni di stop*. La condizione di stop è una variabile booleana, che rappresenta il raggiungimento di una confidenza abbastanza buona sui dati, essa può essere utilizzata nella simulazione per deciderne la terminazione. Ecco le possibili condizioni di stop:

Nessuna condizione di stop: è quella predefinita in NekoStat; se tutte le quantità sono di questo tipo è necessario impostare esplicitamente nel codice dell'applicazione le situazioni in cui interrompere l'esecuzione.

Stop dopo N misure: il grado di confidenza è considerato sufficiente dopo aver raccolto almeno N misure delle quantità.

[2]Gli outlier sono i valori minimi e massimi della quantità, che spesso corrispondono a situazioni limite non utili per l'analisi del valore medio della quantità.

Stop su intervallo di confidenza: quando la dimensione dell'intervallo di confidenza sulla media, di livello $(1 - \alpha)$, è inferiore ad una percentuale β della media della quantità, la confidenza sui dati si considera raggiunta.

3.3.4 Uso di NekoStat per le simulazioni

L'analisi effettuata su una simulazione di un sistema distribuito ha inizio con la preparazione dello **StatLoggerSim**, dello **StatHandler** e dello **StatLayerSim**, da parte dello **StatInitializer**.
Durante la simulazione, i layer segnalano allo **StatLogger** il verificarsi degli eventi (Figura 3.7), il quale richiama il metodo **handle** dello **StatHandler** che, a sua volta, ottiene le misure a partire dall'evento segnalato. A questo punto, lo **StatLogger** controlla se abbiamo raggiunto il grado di confidenza minimo sulle misure e, in questo caso, richiama l'interruzione della simulazine.
In pratica, l'analisi statistica delle quantità di interesse per le simulazioni viene effettuata *on-line*.

3.3.5 Uso di NekoStat per le esecuzioni reali

Anche l'analisi di un'esecuzione reale di un sistema distribuito richiede un fase iniziale di preparazione degli oggetti implicati; in questo caso però, a differenza di ciò che accade per le simulazioni, occorre distinguere tra la fase di preparazione eseguita per il master e quella eseguita per gli slave.
L'avvio dell'*analisi sul master* richiede la creazione dei livelli **Clock Synchronizer**, **ListFragmenter** e **StatLayer** e degli oggetti **StatLoggerMaster** e **StatHandler**. Inoltre, se l'utente utilizza la sincronizzazione dei clock predefinita, viene eseguita una prima fase di sincronizzazione in modo da avere i vari processi con la stessa origine dei tempi del master.

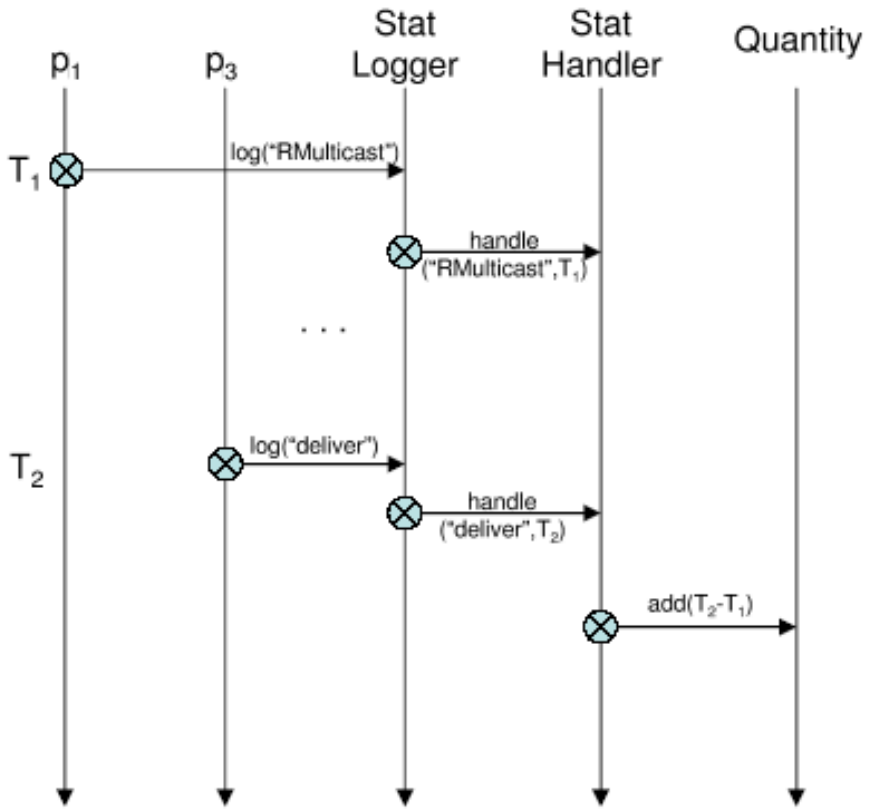

Figura 3.7: Evoluzione nell'analisi dell'algoritmo multicast in simulazione

L'avvio dell' *analisi sugli slave* richiede la creazione dei livelli **ClockSynchronizerSlave**, **ListFragmenter**, **StatLayer** e dello **StatLoggerSlave**.
Soltanto al termine della fase di preparazione ha inizio l'esecuzione dei processi. Durante l'esecuzione lo **StatLogger** locale di ogni processo, raccoglie gli eventi che si verificano nell'**EventCollector** (Figura 3.8). L'interruzione dell'analisi può essere richiesta da qualunque processo tramite l'invio di un apposito messaggio NS_STOP al master. Soltanto all'arrivo del messaggio di stop, lo **StatLayer** del master fa partire la fase di analisi, chiedendo ai singoli partecipanti l'invio degli eventi raccolti. Al momento dell'arrivo degli eventi, lo **StatLayer** esegue il metodo **handle** dello **StatHandler** rispettando però l'ordine temporale fra gli eventi. In fine, vengono esportate le informazioni statistiche di interesse sulle quantità e si avverte il processo che ha chiesto la terminazione dell'analisi. Da quanto detto, si capisce che l'analisi effettuabile sull'esecuzione distribuita è *off-line*: non vengono utilizzate condizioni di stop dell'analisi, quest'ultima deve essere esplicitamente interrotta da un processo del sistema analizzato. É importante notare che, affinchè il protocollo appena descritto funzioni correttamente, è necessario utilizzare una connessione affidabile per lo scambio di messaggi tra i processi coinvolti nell'applicazione distribuita. Nel caso in cui si utilizzi una connessione inaffidabile (come ad esempio UDP), l'utente deve necessariamente istanziare un'apposita rete di controllo per lo scambio dei messaggi di NekoStat e specificare questa rete nel file di configurazione dell'applicazione. Per altre informazioni sull'organizzazione di NekoStat e sul suo uso vedere [1].

3.4 Neko e NekoStat: limiti

In seguito a quanto detto, è chiara la potenzialità del framework Neko e del pacchetto NekoStat; essi forniscono tutti gli strumenti per un'analisi

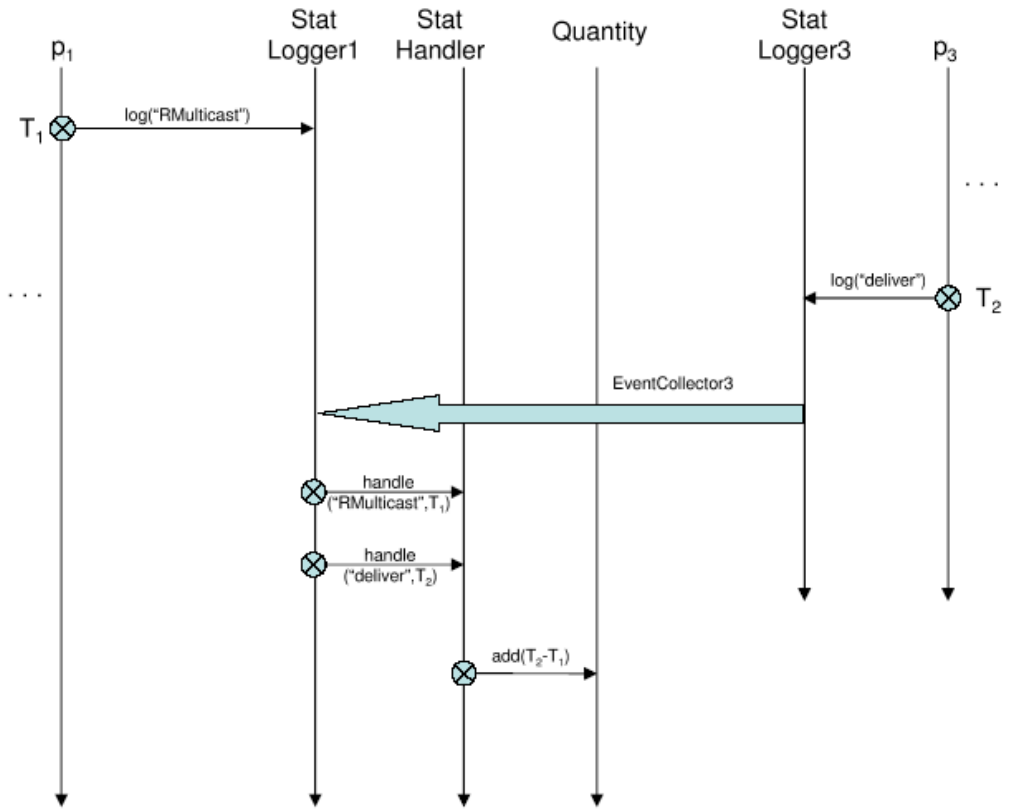

Figura 3.8: Evoluzione nell'analisi dell'algoritmo multicast in esecuzione reale

qualitativa e quantitativà di algoritmi distribuiti. Tuttavia, c'è da osservare che l'implementazione in Java di Neko, se da una parte aumenta la portabilità del framework, dall'altra impedisce l'analisi di algoritmi distribuiti scritti in linguaggi di programmazione diversi da Java. Se dunque, si intende utilizzare il framework per la valutazione di un algoritmo distribuito scritto in un linguaggio di programmazione diverso da Java, occorre considerare che:

- i vantaggi ottenuti, in termini di tempi di sviluppo e di testing, svaniscono di fronte ai costi economici e temporali dovuti al lavoro di traduzione di un algoritmo distribuito complesso da altri linguaggi di programmazione a Java;

- l'accuratezza dei risultati ottenuti dalla valutazione simulativa e da quella basata su misure di un algoritmo distribuito, può essere messa in discussione se si considera che l'operazione di traduzione da altri linguaggi a Java può essere non corretta o che l'algoritmo in Java, anche se esattamente tradotto, può non avere lo stesso comportamento di quello originale.

Per far fronte a tali svantaggi ho realizzato l'estensione del framework Neko presentato nel Capitolo 5.

Capitolo 4

Riuso di software e integrazione di componenti COTS

Questo capitolo ha lo scopo di introdurre i concetti necessari a comprendere i termini e gli strumenti che utilizzerò per effettuare l'integrazione di codice C (o C++) tra i layer di Neko, e le problematiche che occorre affrontare quando si utilizzano tali tecniche.
Così come crescono le dimensioni e la complessità dei sistemi software, allo stesso modo cresce l'interesse sullo sviluppo di sistemi basati su componenti riusabili, conosciuti in letteratura come *COTS (Commercial-Off-The-Shelf)*.
I principali vantaggi di questa nuova tecnologia sono i costi ridotti e i brevi tempi di sviluppo.
La natura dei COTS suggerisce che il modello di sviluppo di software basato su componenti debba essere differente da quello convenzionale; questo approccio si basa sulla costruzione di sistemi software integrando componenti software già esistenti disponibili in commercio. In generale, il ciclo di vita di questi prodotti consiste delle seguenti fasi ([29]):

Identificazione. I componenti possono essere identificati in ogni modo

possibile: ricerche nel Web, letture di riviste specializzate, ecc..

Valutazione. Lo scopo di questa attività è ridurre il numero di componenti identificati valutando le caratteristiche dei vari componenti.

Selezione. La selezione di un componente è una decisione che influenza i requisiti e il design del sistema; è un'operazione risultante dalle due attività precedenti.

Integrazione. Questa attività ha lo scopo di integrare insieme i vari componenti, risolvendo le possibili inconsistenze tra essi e tra i componenti e il resto del sistema.

Aggiornamento. Spesso i componenti subiscono degli aggiornamenti: scopo di questa attività è rendere possibile l'integrazione della nuova versione del componente nel sistema in cui è utilizzato.

In accordo a Wallnau ([30]), i sistemi basati su COTS comprendono una vasta gamma di sistemi, da una parte i sistemi COTS-solution e dall'altra i sistemi COTS-intensive. Con il termine COTS-solution ci riferiamo a quei sistemi che utilizzano un prodotto su misura per trovare una soluzione a un certo problema; i sistemi COTS-intensive sono molto più complessi, questo tipo di sistema integra molti prodotti per fornire diverse funzionalità di sistema. Uno dei principali problemi da affrontare in questi sistemi è il modo in cui i vari componenti sono interconnessi tra loro e come comunicano, è quindi necessario definire un'architettura che permetta l'integrazione dei vari componenti.
In seguito a quanto detto, sembra molto promettente usare componenti COTS per migliorare produttività e qualità nello sviluppo di sistemi software; tuttavia, l'uso di software COTS introduce problemi e rischi nuovi rispetto a quelli che si incontrano costruendo un sistema ex-novo. Molti di questi problemi sono dovuti alla natura black-box dei componenti COTS; in particolare, per costruire un sistema basato su COTS efficace è necessario un accurato processo di valutazione dei COTS.

Per quanto detto fino ad ora non sembra ci sia distinzione tra un COTS e un componente. Effettivamente, ancora oggi, non c'è un'intesa generale sul significato dei due termini; questi due concetti sono strettamente legati e sono usati in molti contesti, tuttavia, occorre fare una distinzione. Brown [31] definisce un componente come: "Una parte non-insignificante, indipendente e sostituibile di un sistema che esegue una chiara funzione nel contesto di una ben definita architettura". In generale, invece, il termine COTS si riferisce a qualcosa che si può comprare, preconfezionato da qualche venditore; si usa questo termine per riferirci a qualsiasi prodotto che non è sviluppato specificatamente per un nuovo sistema, l'acronimo COTS sta per *Commercial off-the-shelf*. Come dice il termine stesso, "off-the-shelf", ci si riferisce a qualcosa che già esiste e che non deve essere sviluppato dall'utente.

4.1 Architettura Software

L'architettura software è la più alta descrizione astratta di un progetto software. É definita nello stadio iniziale dello sviluppo del software ed è comunemente descritta in termini di tre astrazioni: componenti, connettori e configurazioni ([32]). I componenti rappresentano un ampio insieme di elementi differenti, da un singolo cliente a un database, e hanno un'interfaccia usata per comunicare con l'ambiente esterno. I connettori rappresentano gli elementi di comunicazione tra i componenti. La configurazione descrive come i componenti e i connettori sono legati.
La rappresentazione grafica di un'architettura software è mostrata in Figura 4.1. Gli scopi principali dell'architettura software sono:

- definire i maggiori componenti di un sistema;
- definire come i componenti si interfacciano con gli altri;
- definire l'interazione tra componenti.

In generale, lo sviluppo basato su COTS non segue i requisiti architetturali del sistema: ovviamente, quando si costruisce un sistema utilizzando componenti esistenti, è inevitabile l'incompatibilità tra i servizi forniti dai componenti e i servizi che il resto del sistema (ROS) si aspetta dai componenti.

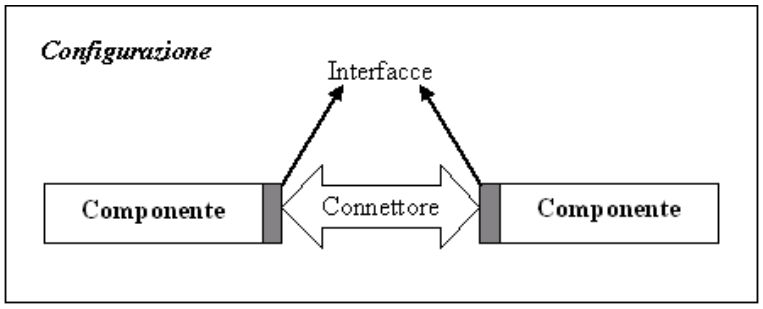

Figura 4.1: Architettura Software

Nel momento in cui questi componenti OTS verranno integrati in un sistema sarà quindi necessario effettuare delle modifiche ai componenti (quando è possibile) e/o al sistema. Le modifiche apportate avranno il principale scopo di minimizzare i conflitti tra componenti diversi e tra un componente e il resto del sistema.
I problemi che si presentano in fase di integrazione dei componenti, quindi, possono essere risolti; in particolare è importante l'utilizzo di elementi software di integrazione, esempi di questi elementi sono:

- il *wrapping*, è una tecnica che permette l'integrazione di un componente *COTS* in un sistema; un *wrapper* è uno speciale componente, inserito tra un componente e il suo ambiente, che ha lo scopo di monitorare e filtrare il flusso di contollo e di dati da e/o verso un *wrapped component*. Il wrapping è particolarmente indicato per l'integrazione di componenti di natura black-box ed è, in generale, una soluzione economica a molti problemi nello sviluppo di software basato su componenti; l'uso del wrapping è vantaggioso per

garantire la dependability del sistema, in pratica protegge sia il sistema contro un comportamento errato di un componente COTS che il componente contro un comportamento errato del resto del sistema , un wrapper usato a tale scopo viene detto *protector*.
Ulteriori motivi che giustificano l'uso del wrapping: ridurre l'impatto che il sistema ha ai cambiamenti di un *wrapped component*, fornire un'interfaccia stadard a un'insieme di componenti, aggiungere o togliere funzionalità a un componente, fornire un solo punto di accesso al componente.

- il *glue code*, è il codice prodotto con lo scopo di collegare insieme i vari componenti di un sistema.
  Tra i compiti che il glue code deve svolgere ricordiamo:

  - il controllo del flusso: invocare in modo opportuno le funzionalità fornite dai componenti;
  - creare un ponte tra componenti: il glue code deve risolvere qualsiasi incompatibilità di interfaccia tra i componenti, per esempio effettuando una conversione di dati;
  - il trattamento delle eccezioni: catturando le eccezioni il glue code può fornire un efficace meccanismo di exception handling.

  In definitiva, se si vuole integrare un componente con lo scopo principale di usufruire dei servizi che può offrire allora è preferibile usare il wrapper come elemento software di integrazione, se invece si desidera integrare un componente e permettergli di interagire fortemente con altre componenti e con il resto del sistema allora è preferibile utilizzare il glue code.

- l'*adattamento dei componenti*, si riferisce all'abilità di migliorare la funzionalità dei componenti; l'adattamento è ottenuto inserendo alcuni elementi aggiuntivi ai componenti con lo scopo di includere

funzionalità che non sono state fornite dal venditore.
L'adattamento non implica necessariamente la modifica del codice sorgente del componente, esempi di adattamento sono gli "script", un'applicazione può essere lanciata eseguendo uno script all'occorrenza di qualche evento.

Occorre comunque sapere che molti dei problemi dovuti all'uso dei componenti COTS possono essere superati attraverso una definizione accurata dell'architettura e della configurazione del sistema.

4.2 Costruire sistemi con componenti COTS

Come abbiamo detto, costruire sistemi con componenti COTS offre l'opportunità di ridurre i tempi e i costi di sviluppo del prodotto software, tuttavia ci sono ancora altri problemi da superare.
Lo sviluppo di sistemi basati sull'integrazione di COTS ha a che fare con un particolare insieme di problematiche economiche, tecniche e non tecniche; qui di seguito elenco alcune delle problematiche da affrontare nello sviluppo di tali prodotti ([32]):

- ***La natura black box dei componenti COTS.*** I clienti spesso non hanno accesso al codice sorgente e non possono quindi cambiare alcuna proprietà del componente; questo implica inoltre che possono essere eseguiti soltanto test a scatola nera durante la valutazione del COTS. Questo tipo di testing è eseguito anche durante i test di accettazione ed è considerato un test fondamentale durante il test di convalida che conferma che il software fornisce le funzionalità richieste.

- ***La selezione dei componenti COTS non è un compito banale.*** É importante scegliere il componente che dovrà essere integrato nel sistema, la scelta viene spesso fatta in base alla documentazione che accompagna il componente stesso. C'è da dire

però che i criteri di valutazione forniti con i componenti sono spesso soggettivi ed ambigui e non forniscono, al cliente, un'effettiva descrizione del prodotto.

- ***Le specifiche COTS sono spesso incomplete e superficiali.*** Molti dei documenti forniti con il componente consistono di un manuale utente e di materiale che pubblicizza il prodotto e non forniscono una precisa descrizione delle capacità e dei limiti del COTS. Ad esempio non forniscono una descrizione del comportamento del componente in presenza di un input inatteso che rappresenta un dato legato alla qualità dell'affidabilità e della stabilità del componente.

- ***Aggiornamenti frequenti del prodotto commerciale.*** I componenti presenti sul mercato cambiano rapidamente e nuove versioni sono rilasciate con molte funzionalità differenti. Questo aspetto influenza significativamente la scelta di un prodotto, poiché una nuova versione può avere una caratteristica che non è disponibile nel prodotto che si stà valutando.

- ***I requisiti cambiano durante il processo di sviluppo.*** Quando si ha a che fare con sistemi basati su COTS, succede spesso che alcuni dei requisiti del sistema vengono conosciuti soltanto dopo la fase di valutazione iniziale e addirittura dopo che l'integrazione del sistema è iniziata. Inoltre, alcuni prodotti impongono requisiti architetturali addizionali come vincoli di affidabilità e interoperabilità che sono di solito inattesi durante la valutazione iniziale.

- ***Aggiungere un nuovo COTS significa aggiungere ulteriori vincoli.*** Questi vincoli possono influire sull'intera architettura del sistema, sugli aspetti funzionali e non funzionali; per esempio, un nuovo componente può avere un'influenza negativa su tutti gli

attributi di qualità del sistema. Ovviamente occorre analizzare e risolvere questi contrasti.

- ***L'adattamento dei COTS è di solito richiesto per nascondere funzionalità indesiderate.*** I produttori di COTS spesso caricano i loro sistemi di un gran numero di funzionalità che non servono all'utente. Poichè questi inconvenienti non possono essere risolti quando si ha a che fare con componenti COTS di tipo black box, gli architetti di sistema devono trovare il modo di mascherare le funzionalità inattese in modo che siano inaccessibili all'utente finale e ai programmatori di sistema.

- ***L'insieme dei componenti COTS può essere male assortito.*** L'integrazione di componenti COTS di solito porta molte strutture di architettura inconsistente all'interno di un singolo sistema; questo problema è stato identificato come un divario architetturale, Garlan ([32]) ha identificato tre categorie di inconsistenza tra componenti in conflitto, le riassumo:

  - la natura dei componenti
  - la natura dei connettori
  - il processo di costruzione

  Queste inconsistenze architetturali costituiscono l'ostacolo fondamentale allo sviluppo basato su componenti.

- ***I clienti non hanno alcun controllo (o ne hanno poco) sull'evoluzione del prodotto.*** I produttori di COTS hanno molti clienti, quindi è quasi impossibile soddisfare specifici bisogni di un singolo cliente; questi ultimi dunque hanno pochissima influenza sull'evoluzione futura del componente e le ulteriori capacità fornite negli aggiornamenti dei componenti sono dettate principalmente da strategie economiche e tendenze tecnologiche.

- ***La dipendenza dal fornitore per ciò che riguarda l'aggiornamento del sistema.*** L'estensione futura di un sistema basato sull'uso dei COTS dipende dalla possibilità di ottenere un aggiornamento dei componenti con le nuove capacità richieste, questo potrebbe essere un problema data la dipendenza dal fornitore. Inoltre non è sempre garantito il supporto del fornitore o la compatibilità tra le versioni successive di un COTS integrato in un sistema; se il fornitore decide di non finanziare ulteriormente il prodotto o addirittura di toglierlo dal mercato, i clienti sono obbligati a cambiarlo con un diverso COTS, questo comporta la modifica dell'architettura del sistema.

- ***Le sostituzioni dei COTS influenzano l'architettura e la qualità del sistema.*** La sostituzione di un particolare COTS non idoneo al sistema che si stà sviluppando, può portare ad una seria inconsistenza e ad una costosa riprogettazione del sistema. Vediamo degli esempi di inconsistenza:

  - un nuovo componente può non essere compatibile con la piattaforma usata dall'utente;
  - un nuovo componente può interferire con alcune funzionalità del sistema;
  - un nuovo componente può interagire in modo inatteso con gli altri componenti.

Allo scopo di affrontare efficacemente i potenziali problemi identificati sopra è necessario che il processo di sviluppo di sistemi basati su COTS si basi su un approccio sistematico e dettagliato. Il prossimo paragrafo fornisce una discussione su come gestire i rischi che possono presentarsi durante lo sviluppo di tali sistemi.

4.3 Indicazioni su come affrontare i rischi associati allo sviluppo di sistemi basati su COTS

In questo paragrafo mostrerò un insieme di indicazioni da seguire per affrontare in modo conveniente i rischi identificati durante il ciclo di vita dello sviluppo di sistemi basati su COTS.
Credo che la seguente classifica copra un'ampia gamma di situazioni critiche nello sviluppo di tali sistemi; per ogni situazione (critica) vengono elencati i rischi e poi le rispettive indicazioni ([32]).

Prima situazione *Assenza di processi di selezione ben definiti.*

Rischi :

1. Un dominio povero di possibilità, per cui il processo di analisi del dominio può essere soggetto a una perdita di qualità.
2. Non si trae insegnamento dalle esperienze passate.
3. Viene fatta pressione alle organizzazioni in modo che compiano un veloce processo di valutazione senza permettere di utilizzare persone qualificate. Questa situazione può portare in una inefficace o errata selezione dei COTS.
4. Molti individui sono contrari allo sviluppo basato su COTS.

Indicazioni :

1. Acquisire informazioni circa il dominio di sistema da specialisti e usare interviste e questionari tecnici.

2. Definire un processo di selezione metodico, sistematico, usare piani strategici ed appropriati metodi e tools di selezione.
3. Nominare un team di valutazione composto da specialisti nei processi di selezione e esperti di dominio.
4. Gestire riunioni educative, spiegando il beneficio dello sviluppo basato sui COTS e descrivendo i casi di successo industriale che sono stati ottenuti usando questo tipo di approccio.

Seconda situazione *Criteri di valutazione inefficaci.*

Rischi :

1. Dare poca importanza ai requisiti non funzionali aumenta i rischi di fallimento dei COTS e il costo del sistema finale.
2. Se i requisiti sono troppo specifici e rigidi potrebbe essere impossibile trovare una soluzione appropriata che vada incontro simultaneamente ai requisiti, all'architettura e alla disponibilità dei componenti COTS.
3. Gli sviluppatori hanno l'idea sbagliata che il costo dei sistemi basati su COTS è legato esattamente al costo di acquisto.

Indicazioni :

1. Usare approcci che specifichino i requisiti non funzionali. L'analisi di questi requisiti aiuta a scegliere tra prodotti in competizione e migliora e accelera il processo di scelta dell'insieme di COTS che sarà integrato nel sistema finale.

2. Iniziare col descrivere in linea di massima i requisiti in modo da trovare potenziali COTS disponidili nel mercato; poi raffinare la descrizione dei requisiti, in particolare di quelli non funzionali. Usare tecniche che aiutino a scegliere un prodotto che soddisfi i requisiti richiesti e i vincoli architetturali.
3. Quando si sviluppa un sistema basato su COTS occorre considerare ulteriori costi dovuti ad esempio alla preparazione, all'adattamento e al mantenimento.

Terza situazione *I componenti COTS sono sviluppati come prodotti generici.*

Rischi :

1. Lo sviluppo di un COTS non ha lo scopo di soddisfare i requisiti di un particolare cliente ma di soddisfare un intero mercato; è per ciò che spesso i prodotti COTS forniscono più funzionalità di quante servono al cliente.
2. Non è garantito che un prodotto COTS soddisfi tutti i requisiti dichiarati.

Indicazioni :

1. (a) Il processo di valutazione deve essere guidato dal contesto, cioè la valutazione dei COTS deve essere effettuata senza considerare lo scopo per cui il sistema è costruito.
   (b) Utilizzare tecniche, tipo il wrapping, per mascherare capacità attualmente inutili.
2. (a) Fare una scala di priorità dei requisiti e cercare di assicurare che almeno i requisiti critici siano soddisfatti.

(b) Occorre sviluppare componenti aggiuntivi per affrontare il problema, ma devono esser specifici per un particolare dominio.

Quarta situazione *Insufficienza di descrizioni dettagliate dei componenti COTS.*

Rischi :

1. La scelta errata di componenti COTS porta a problemi di integrazione.
2. Molto spesso i COTS non sono facilmente confrontabili, la causa principale è l'assenza di un vocabolario unico usato per la loro descrizione.

Indicazioni :

1. Condurre incontri dimostrativi allo scopo di evidenziare le capacità dei prodotti in un ambiente più realistico. Durante queste riunioni, è importante valutare la compatibilità, l'integrabilità e l'interoperabilità dei prodotti con l'architettura del sistema.
2. Usare una documentazione standard per descrivere le capacità dei COTS

Quinta situazione *L'architettura può influenzare significativamente l'intero sistema basato su COTS.*

Rischi :

1. L'architettura stabilita è inflessibile e difficilmente adattabile per particolari circostanze.

2. Il sistema integrato può presentare proprietà inadeguate che possono persistere anche durante la manutenzione; tale situazione può essere dovuta a una errata selezione dei COTS.
3. Esiste un'interazione diretta tra componenti COTS, è perciò che l'aggiornamento di uno di essi può influenzare gli altri componenti o addirittura l'intero sistema.

Indicazioni :

1. Evitare di stabilire a priori un'architettura; le scelte architetturali devono essere prese concorrentemente con la valutazione dei COTS, in modo tale che esse possano essere modificate facilmente nel momento in cui la dichiarazione dei requisiti viene raffinata. Ovviamente i requisiti possono anche essere cambiati per andare incontro ai vincoli architetturali.
2. Eseguire test di verifica per determinare il livello di conformità tra i COTS integrati e la descrizione architetturale.
3. I COTS non dovrebbero interagire direttamente con gli altri; ogniqualvolta un componente viene aggiornato dovrebbero essere modificati anche i wrappers e/o il glue code.

Sesta situazione *I componenti COTS si sviluppano rapidamente.*

Rischi :

1. I clienti non hanno alcun controllo sullo sviluppo dei prodotti COTS.
2. L'accettazione acritica delle affermazioni circa le capacità del prodotto e l'assitenza sul COTS fornito.

Indicazioni :

1. (a) Usare architetture flessibili facilitando la modifica e l'aggiornamento dei COTS.
   (b) Valutare attentamente i precedenti COTS forniti e stabilire una strategia a favore dell'attuale versione del sistema, sincronizzando gli aggiornamenti dei COTS con le versioni del sistema.
2. Stabilire una strategia per convincere i fornitori di COTS a fornire assistenza e negoziare gli accordi in modo critico.

4.4 JAVA e C: integrazione

Nonostante Java sia un linguaggio in continua crescita ([33, 34, 35]), il suo principale "difetto" continua ancora ad esistere: la mancanza di librerie scientifiche specificatamente create per essere utilizzate in Java; questo impone l'utilizzo delle più note librerie scientifiche scritte in linguaggi come C. É quindi chiara la necessità di permettere un'integrazione di elementi scritti in C in elementi scritti in Java; tuttavia i motivi per una tale integrazione possono essere molteplici, a seconda della necessità: nel Capitolo 1 ho elencato le mie motivazioni.
Il modo più appropriato per rendere disponibile in Java qualsiasi oggetto[1] scritto in C o C++, è fornire un'interfaccia ad esso attraverso l'uso di **API**[2] **Java Native Interface** (**JNI**)([36, 37, 38]).
JNI è l'interfaccia appositamente studiata per l'integrazione di codice nativo; è parte del JDK, ciò preserva la portabilità del codice su qual-

[1] potrebbe essere una libreria, una semplice procedura o altro ancora.

[2] **API** è l'acronimo di ***A**pplication **P**rogramming **I**nterface*, indica ogni insieme di procedure disponibili al programmatore, di solito raggruppate a formare un set di strumenti specifici per un determinato compito.

siasi piattaforma nonostante l'uso della tecnologia JNI.
Così com'è possibile rendere disponibile in Java oggetti in C o C++, allo stesso modo, è possibile includere la Java Virtual Machine nelle applicazioni native, utilizzando le ***Invocation API***.
Per esempio, la seguente figura (Figura 4.2) mostra come un programma legacy in C, utilizzando JNI, possa ottenere il link ad una libreria Java, chiamare metodi Java definiti in classi Java e ottenere un riferimento alla JVM. La Figura 4.3 invece, mostra la chiamata di una funzione

Figura 4.2: Come richiamare Java da C

nativa da un'applicazione Java; questo diagramma evidenzia le varie possibilità di utilizzo di JNI, tra cui includere chiamate a routines C e usare classi C++.
In pratica JNI viene usata come *glue code* tra Java e le applicazioni native; la Figura 4.4 mostra come JNI lega la parte in C di un'applicazione alla parte in Java.

4.4.1 I problemi

Data la differenza tra Java e gli altri linguaggi di programmazione, è naturale che sorgano dei problemi nel momento in cui si intende far interagire due oggetti qualsiasi di due linguaggi differenti; i principali problemi sorgono durante la generazione dell'interfaccia. Prendiamo in considerazione il linguaggio C e vediamo alcuni di questi problemi ([39]):

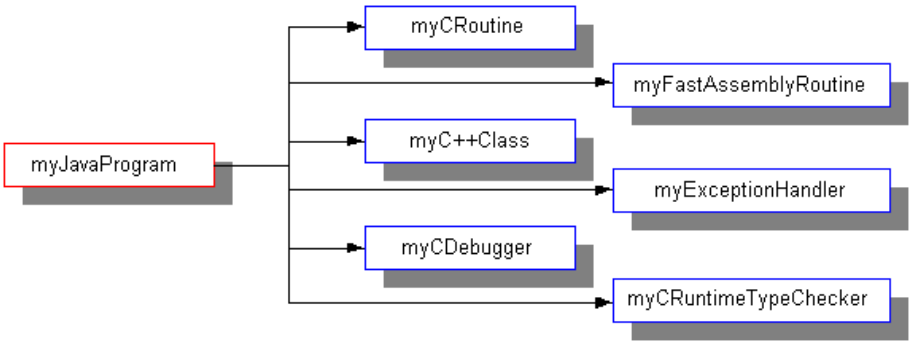

Figura 4.3: Come richiamare codice nativo da Java

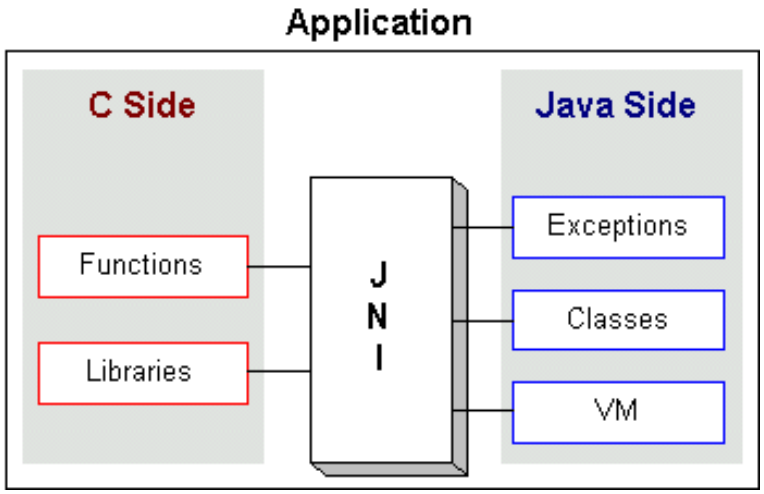

Figura 4.4: JNI come codice colla

- i tipi di dati primitivi in C hanno dimensioni variabili che possono non coincidere con le standardizzate dimensioni dei tipi di dati Java; si pensi, ad esempio, alle stringhe: in Java le stringhe sono memorizzate come sequenze di caratteri Unicode a 16 bit, mentre in C sono memorizzate come sequenze di caratteri a 8 bit.

- in Java non esiste una diretta corrispondenza dei puntatori C ([40]);

- gli arrays multidimensionali in C hanno un layout di dati contiguo, mentre in Java non è così;

- le strutture C possono essere emulate da oggetti Java, ma il layout dei campi degli oggetti può essere diverso da una JVM a un'altra, proibendone l'accesso diretto.

Ulteriori problemi si presentano nel momento in cui si intende legare del codice nativo a Java. In un qualsiasi linguaggio ogni libreria può essere usata da un'applicazione in un modo piuttosto semplice, ossia usando un meccanismo di linking statico o dinamico; a tale proposito, vedere la Figura 4.5.

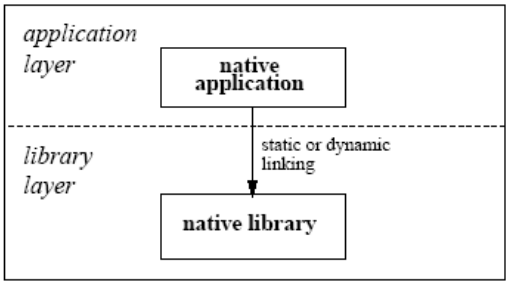

Figura 4.5: Uso di librerie native in un'applicazione

L'uso della stessa libreria in Java è mostrato in Figura 4.6; in questo caso le cose si complicano un pò: i file class, generati dal compilatore Java, sono indipendenti dal sistema e quindi non possono essere direttamente

linkati ad una qualsiasi libreria nativa, il processo di linking deve essere fatto durante l'esecuzione del programma ed è effettuato dalla JVM.
É necessario che la libreria contenga metodi definiti come *nativi* nelle classi Java; in pratica una libreria di questo tipo è semplicemente un wrapper che richiama funzioni native.
Il linking dinamico del wrapper è eseguito dalla JVM con una chiamata al metodo **System.loadLibrary()**, la comunicazione tra Java e il codice nativo è permessa grazie alle funzioni JNI.
In questo modo, in uno stesso momento può essere eseguito il link a più di una libreria nativa e, per ognuna di esse, è possibile interagire con qualsiasi altra libreria, proprio come una tradizionale applicazione a "linguaggio singolo".

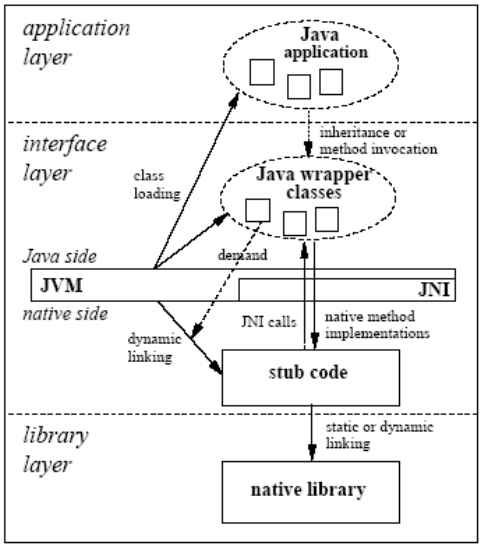

Figura 4.6: Uso di librarie native in un'applicazione Java

4.4.2 Jace

Nonostante i numerosi vantaggi ottenuti dall'uso delle API JNI occorre considerare alcuni problemi che si evidenziano soprattutto nel momento

in cui si lega JNI al linguaggio C:

- la salvaguardia dei tipi è quasi inesistente;
- l'*error checking* è assente;
- occorre "maneggiare" il **JNIEnv**, puntatore a una tabella di puntatori a funzioni[3].

Jace è un toolkit gratuito, open source, progettato per facilitare la programmazione JNI ([41, 42]). Supporta la generazione automatica di *peer classes* e di *proxy classes* C++ da files class Java e l'integrazione delle eccezioni, arrays, packages e oggetti Java.
Il punto di forza si Jace è l'uso delle peer classes e delle proxy classes in C++ per rappresentare i tipi Java; la generazione di proxy classes permette allo sviluppatore di istanziare e manipolare facilmente oggetti Java da C, proprio come se fossero classi native C++, mentre la generazione di peer classes facilita l'implementazione dei metodi nativi dichiarati nelle classi Java.
Il diagramma in Figura 4.7 è una visione ad alto livello delle relazioni tra sviluppatore, i tool[4] per la generazione del codice, messi a disposizione da Jace, e la Jace Runtime Library (JRL). Per capire veramente i benefici delle proxy classes, basta rivedere il sistema dei tipi di JNI. In JNI si usano 24 tipi C per rappresentare l'intero insieme di possibili tipi Java, in particolare si hanno nove tipi primitivi:

- jboolean
- jbyte
- jchar
- jshort

[3]Queste funzioni forniscono la possibilità di utilizzare tipi Java in C e C++.
[4]ProxyGenerator, BatchGenerator, AutoProxy, PeerEnhancer e PeerGenerator.

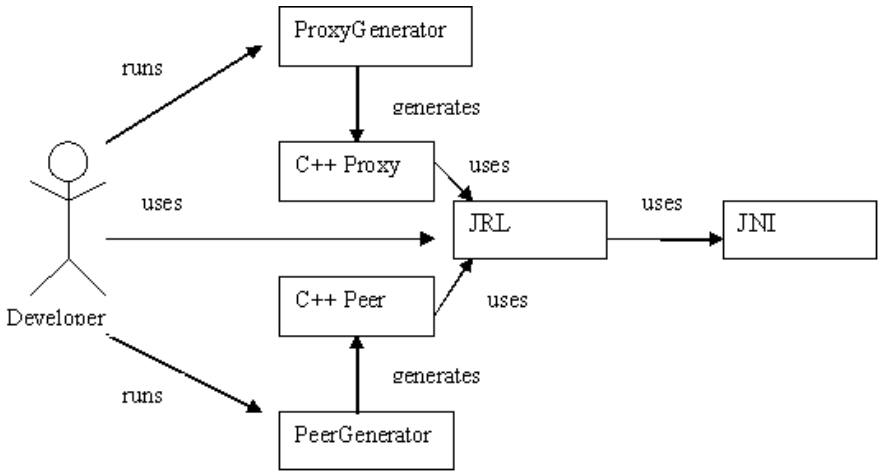

Figura 4.7: Diagramma delle relazioni

- jint
- jlong
- jdouble
- jfloat
- void

quattordici *reference type*, mostrati in Figura 4.8, e il tipo *jvalue* che rappresenta qualsiasi tipo primitivo o reference.
Il sistema di tipi di Jace è costruito direttamente sui 24 tipi JNI; per ogni tipo JNI, Jace ha una proxy class C++ che fa da wrapper ai tipi Java esistenti. La funzione di queste classi wrapper è quella di fornire un facile accesso ai tipi Java; l'utilizzo dei tipi Jace facilita quindi la comunicazione tra Java e C (o C++) rispetto a quanto succede con i tipi JNI.
La creazione delle peer classes e delle proxy classes avviene automaticamente, grazie all'uso dei tool che Jace mette a disposizione dello sviluppatore; vediamo quali sono e qual'è la loro funzione.

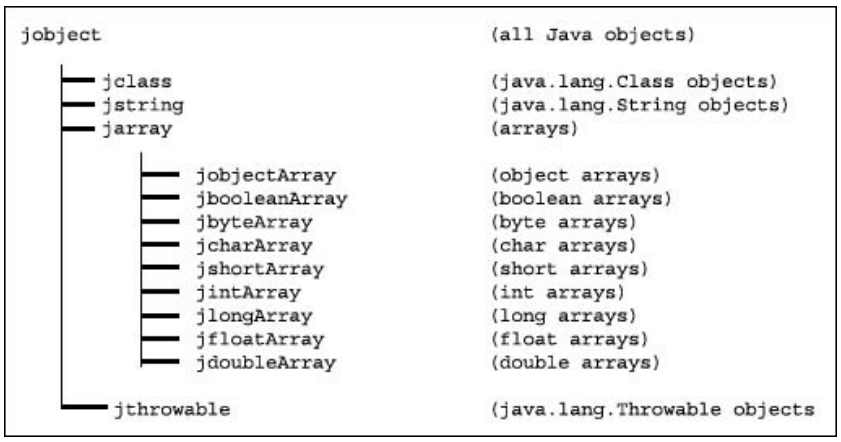

Figura 4.8: I 14 reference type JNI

ProxyGenerator: genera una proxy class C++ per una singola classe Java specificata; in genere, al suo posto si preferisce usare il tool AutoProxy in quanto quest'ultimo è in grado di percorrere l'albero delle dipendenze tra classi e generare tutte le classi dipendenti dalla prima.

AutoProxy: questo tool analizza i C++ header files e i file sorgenti seguendo la direttiva #include e, per ogni header file, crea la corrispondente proxy class seguendo l'albero delle dipendenze. Ad esempio, seguendo
`#include jace/proxy/java/lang/RuntimeException.h`,
AutoProxy genererà le proxy classes per RuntimeException, Throwable, Object, Serializable, String, ecc...AutoProxy è anche usato in combinazione con altri tool come il BatchGenerator e il PeerGenerator.

BatchGenerator: esso è usato per generare tutte le proxy classes all'interno di un file jar. Questo tool è utile quando si intende creare una API C++ per una API Java esistente.

PeerEnhancer: questo tool è usato per arricchire il bytecode delle Java

peer classes con la gestione automatica del codice nativo.

BatchEnhancer: esegue il PeerEnhancer su sorgenti multiple in un singolo run.

PeerGenerator: questo tool genera il C++ header file e il source code necessario per implementare la C++ peer class nativa per le peer classes Java.

Nel capitolo seguente mostrerò in che modo ho utilizzato le due tecnologie (JNI e Jace) per effettuare l'estensione di Neko.

Capitolo 5

Estensione al framework: NekoC

5.1 Approccio seguito

L'idea è di trattare gli algoritmi distribuiti, scritti in C o C++, come dei componenti *Off-the-shelf (OTS)* da integrare ad un sistema esistente, che in questo caso è il framework Neko.
Le strade da seguire per rendere possibile un'integrazione di tali algoritmi tra i layer di Neko, sono essenzialmente due:

- permettere l'integrazione dell'eseguibile, trattare quindi l'algoritmo in C o C++ come un componente di natura black-box.
- permettere l'integrazione del codice sorgente.

Il vantaggio della prima alternativa è essenzialmente quello di non modificare affatto il codice sorgente dell'algoritmo distribuito da valutare; per rendere possibile una tale integrazione il procedimento da seguire è il seguente: catturare, a livello di sistema operativo, le chiamate ai metodi send() e recv(), che permettono la comunicazione tra processi, bloccarle e sostituirle con le chiamate **send(NekoMessage m)** e **deliver(NekoMessage m)** del framework Neko.

Questo procedimento, tuttavia, comporta una dipendenza dal sistema operativo utilizzato; tale dipendenza va contro la filosofia di Neko, dato che sin dal principio l'indipendenza dal sistema operativo è stato uno dei principali vantaggi di Neko. Inoltre, per effettuare un'analisi quantitativa di un algoritmo, utilizzando il pacchetto NekoStat, è necessario "ritoccare" il codice dell'algoritmo in modo da segnalare il verificarsi di un evento allo **StatLogger**, viene da se che se si ha a che fare con un componente black-box queste cose si complicano. Questi i motivi che mi hanno scoraggiato a intraprendere questa strada.
D'altra parte, la seconda alternativa impone di ritoccare, anche se in minima parte, il codice sorgente dell'algoritmo da valutare ma in compenso permette di rendere il procedimento di integrazione indipendente dal sistema operativo e facilita l'utilizzo di NekoStat per un'analisi quantitativa; questo è l'approccio che ho scelto di seguire.

5.2 Descrizione

L'idea che ho seguito è la seguente:

- costruire del glue code (o codice colla) che permetta la comunicazione tra oggetti[1] Neko, scritti in Java, e codice C o C++;
- adattare il componente in modo da permettere la comunicazione tra codice C o C++ e oggetti Neko.

L'architettura che ne risulta è mostrata in Figura 5.1, il **glue code** è un layer Neko interamente scritto in Java, il **LayerC** non è altro che l'algoritmo scritto in C o C++ che si intende valutare.
L'avvio di un'applicazione Neko, modellata secondo l'architettura mostrata in Figura 5.1, comincia con l'inizializzazione dei layer che costituiscono il glue code, sarà compito di questi layer attivare la comunicazione con i **layerC** (vedremo più avanti come ciò avviene); la comunicazione

[1] possono essere layer o NekoProcess.

tra i processi che compongono l'applicazione Neko non è affatto influenzata dall'inserimento di questi nuovi elementi nell'architettura di Neko: la rete (sia reale che simulata) comunica direttamente con il glue code e non è a conoscenza dei nuovi elementi, la stessa cosa vale per gli altri layer che compongono il processo.
D'altra parte, il codice che andrà a comporre il layerC, non comunicherà direttamente con la rete; il codice sorgente sarà adattato (vedremo più avanti come) in modo tale che qualsiasi informazione che si intende inviare sulla rete in realtà sia inviata al glue code. Nei paragrafi seguenti cercherò di spiegare il compito del glue code e il suo funzionamento, nonchè l'adattamento fatto sul codice sorgente dell'algoritmo distribuito.

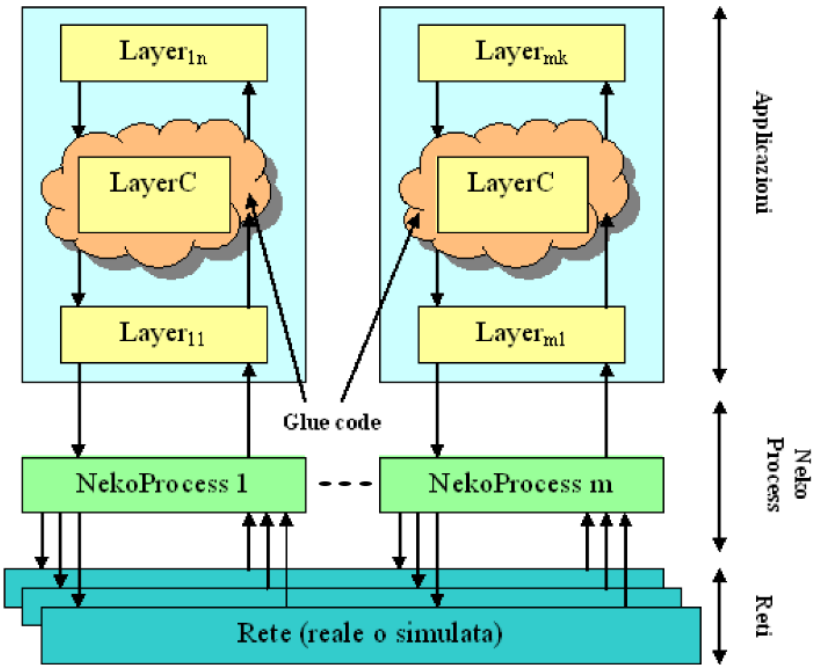

Figura 5.1: Architettura di NekoC

5.2.1 Glue code

Il **glue code** è un **ActiveLayer** direttamente avviato dal processo di inizializzazione ([24]). La prima operazione da eseguire in questo layer è la definizione dei metodi nativi, ovvero di quei metodi che compongono l'algoritmo distribuito da integrare; in genere, un algoritmo distribuito ha un metodo che si occupa della ricezione dei messaggi dalla rete, un metodo di inizializzazione e un metodo che si occupa dell'invio dei messaggi nella rete, questi saranno i metodi da definire nativi nel glue code. La dichiarazione nativa dà la possibilità di eseguire la funzione nativa all'interno della JVM.
Il passo successivo è definire i metodi che avranno il compito di passare e ricevere informazioni dal layerC e/o da qualsiasi oggetto C. I metodi principali, che non devono assolutamente mancare nella definizione del glue code, sono:

Il metodo deliver(NekoMessage m). Compito di questo metodo è quello di scomporre il **NekoMessage**, ricevuto dalla rete o da un layer sottostante, nelle sue componenti e richiamare il metodo nativo che si occupa della ricezione dei messaggi (Figura 5.2). La chiamata del metodo nativo richiede il passaggio di quei parametri ottenuti dalla scomposizione del NekoMessage e che sono utili al **layerC** per la sua esecuzione; ovviamente, le informazioni da passare in input al metodo nativo dipendono dall'algoritmo che si sta analizzando.

Il metodo run(). Questo metodo ha come compito principale quello di richiamare il metodo nativo di inizializzazione e il metodo nativo che si occupa, tra le altre cose, di eseguire l'invio dei messaggi (Figura 5.3); il codice di quest ultimo metodo nativo sarà ovviamente adattato in modo tale da non inviare il messaggio nella rete ma inviarlo al glue code.

il metodo ioinvio(...). É il metodo che si occupa del vero invio dei

NekoMessage (Figura 5.4), anche questo personalizzabile in base alle necessità. Compito di questo metodo è costruire il **NekoMessage** con le informazioni ricevute in ingresso dal metodo chiamante (vedremo più avanti di chi si tratta) ed effettuare la chiamata al metodo **send(NekoMessage m)**.

```
...
 //definizione del metodo nativo
 public native void deliverC(...);
 ...
 public void deliver(NekoMessage m)
  {
      //scompone il NekoMessage m
      ...
      //richiama il metodo nativo del
      //layerC passandogli in input gli
      //opportuni valori
      deliverC(...);
  }
```

Figura 5.2: Porzione di codice del metodo deliver del glue code

```
...
//definizione dei metodi nativi
public native void inizializza_layerC();
public native void runC();
...
public void run()
{
      try{
            sleep(5000);
      }
      catch (Exception ie) {
            logger.info("interrupted!");
      }
      if (containi==0) {
            inizializza_layerC();
            containi++;
            }
      runC();
}
```

Figura 5.3: Porzione di codice del metodo run del glue code

La chiamata di questi metodi nativi è resa possibile grazie all'utilizzo del toolkit Jace (Capitolo 4); la cosa essenziale da fare è generare le

```
public void ioinvio(…)
{
      //i valori ricevuti in input andranno a
      //costruire il contenuto del NekoMessage
      int[] dest = new int[] { 0 };
      NekoMessage m=new NekoMessage(dest,…);
      sender.send(m);
      m=null;
}
```

Figura 5.4: Porzione di codice del metodo ioinvio del glue code

opportune C++ peer classes per poter utilizzare i metodi nativi, questo viene fatto utilizzando i tool messi a disposizione da Jace. I passi da fare per implementare le peer classes sono:

1. inserire nel bytecode della classe peer Java la gestione del codice nativo;
2. generare il C++ header file, contenente le dichiarazioni per la C++ peer class, e il file contenente l'implementazione della C++ peer classe nativa per la Java peer class;
3. generare tutti gli header file richiesti.

I tool che occorre utilizzare per l'esatta generazione di questi file sono elencati quì di seguito:

1. PeerEnhancer;
2. PeerGenerator;
3. AutoProxy.

Nell'Appendice A sono mostrati i passi da seguire per utilizzare il toolkit Jace all'interno di Neko.

5.2.2 Adattamento del componente da integrare

Come già detto, la scelta di integrare il codice sorgente tra i layer di Neko comporta una piccola modifica del codice; in particolare:

- includere, attraverso la direttiva #include, il file header del glue code e i files header dei tipi Jace che i metodi in C o C++ riceveranno in input dal glue code; i file header sono generati automaticamente dal tool *PeerGenerator* di Jace. Per capire meglio, supponiamo che l'algoritmo originale in C contenga il metodo *rcv(...int num...)*, i file da includere sono: il file header del glue code e il file `jace/proxy/types/JInt.h`; è necessario includere quest ultimo file in modo da poter utilizzare il tipo JInt di Jace, ossia il wrapper del tipo int di Java. Un'ulteriore cosa da fare è specificare la direttiva `#include ''jace/proxy/JObject.h''`, vedremo più avanti a cosa serve;

- sostituire il metodo che si occupa della ricezione di un messaggio dalla rete, con un metodo, definito nativo nel glue code, che riceve in input le componenti del messaggio dal glue code stesso. Rifacendoci all'esempio di prima, sostituiamo la signatura del metodo *rcv(...int num...)* con la seguente *rcv(...JInt num...)*.

- sostituire la chiamata al metodo *send(...)*, che si occuppa dell'invio di un messaggio nella rete, con la chiamata ad una funzione definita in una libreria C++, appositamente creata, il cui compito è essenzialmente quello di richiamare il metodo **ioinvio(...)** del glue code, nel prossimo paragrafo spiego meglio il funzionamento di questa libreria. Perchè la libreria C++ possa comportarsi correttamente, occorre passarle in input oltre alle informazioni che andranno a comporre il NekoMessage, anche l'oggetto Java che ha chiamato il metodo nativo e l'identificatore del processo (inteso nel senso di identificatore di NekoProcess). L'oggetto Java che ha richiamato il metodo nativo è ottenuto grazie al metodo `jobject ogg=getJavaJniObject()`, ecco perchè è stato necessario includere il file header `jace/proxy/JObject.h`.

5.2.3 Passaggio dal metodo nativo a Java

Fino ad ora ho mostrato come il glue code, grazie all'uso del tool Jace, riesce a garantire un passaggio di informazioni, contenute nei NekoMessage, dai layer sottostanti al layerC; manca da definire come avviene lo scambio di informazioni dal codice nativo al glue code e quindi ai layer sottostanti.
Il passaggio dal layerC al glue code, avviene attraverso un'apposita libreria (vedi Figura 5.5), da me definita, richiamata dal layerC al posto

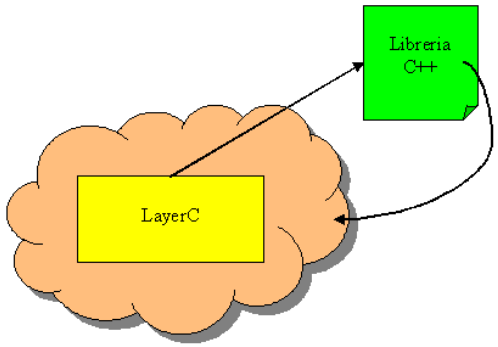

Figura 5.5: Passaggio dal layerC al glue code

del metodo send(...).
Il suo compito principale è utilizzare la tecnologia JNI per consentire la comunicazione tra C++ e Java. In questo paragrafo mostrerò in che modo ho costruito questa libreria e quali passi occorre eseguire per poter richiamare da essa un metodo Java.
Prima di poter richiamare un qualsiasi metodo Java definito nel glue code, occorre eseguire alcuni passi fondamentali:

- la prima operazione è includere il file header C jni.h; questo file contiene tutte le definizioni di tipi e di funzioni che servono per utilizzare JNI da un qualsiasi file C;

- dopodichè, occorre dichiarare tutte le variabili che andremo ad usare all'interno del programma; tra queste abbiamo
  - `JNIEnv *env`: rappresenta l'ambiente di esecuzione JNI; in realtà, è un puntatore all'ambiente JNI che rende possibile l'utilizzo della JVM da parte del thread corrente;
  - `JavaVM jvm`: è un puntatore alla JVM;
- l'ultima operazione da fare, prima di passare alla porzione di codice che si occupa effettivamente di creare un "ponte" tra C++ e Java, è creare la JVM; in realtà, nel nostro caso la JVM è già stata istanziata da Neko quindi quello che si fa è verificare che la JVM sia ancora in esecuzione e ottenere un riferimento ad essa, questa operazione viene eseguita dalla funzione:
  `JNI_GetCreatedJavaVMs(&jvm,5,&jvmCount)`;
  il primo parametro di questa funzione è il puntatore alla JVM, il secondo indica il numero massimo di JVM che possono essere in esecuzione, il terzo parametro conterrà il numero effettivo di JVM in esecuzione;

A questo punto abbiamo settato l'ambiente necessario per la chiamata di un metodo Java definito nel glue code; ciò che resta da fare è trovare un modo per identificare questo metodo. I passi da seguire sono i seguenti:

Ottenere un riferimento all'ambiente di esecuzione JNI. É essenziale ottenere il riferimento al JNIEnv, questo, infatti, è l'unico modo per ottenere un puntatore a tutte le funzioni messe a disposizione da JNI per la chiamate di un metodo Java; esso si ottiene grazie alla funzione:
`jvm->GetEnv((void**) &env,JNI_VERSION_1_4)`.

Trovare e caricare la classe Java. Per poter richiamare un metodo Java occorre, ovviamente, trovare la classe in cui è definito e ottenerne un riferimento; quest'operazione è eseguita utilizzando la

funzione `cls = env->FindClass(nome_classe_codicecolla)`. Se la classe viene trovata, la variabile *cls* rappresenta una "maniglia" (o handle) alla classe Java, altrimenti vale zero;

Trovare il metodo Java. Rimane da trovare il metodo **ioinvio(...)** all'interno della classe del glue code, questo si ottiene utilizzando la funzione `mid=env->GetMethodID(cls,ioinvio,(I)V)`. Il primo parametro è l'handle alla classe del glue code, il secondo è il nome del metodo che intendiamo richiamare, il terzo parametro indica i tipi dei parametri in input e di ritorno del metodo Java, nell'esempio di sopra I stà per int e V per void (per ulteriori informazioni vedere [37, 38]); se il metodo viene trovato la variabile *mid* rappresenta un handle al metodo, altrimenti vale zero.

Chiamata del metodo Java. Finalmente possiamo richiamare il metodo **ioinvio(...)**, solo a questo punto il comando passa al glue code che ha il compito di costruire il NekoMessage e inviarlo al livello sottostante.

La funzione che mi permette di richiamare l'esatto metodo Java è `env->CallNonvirtualVoidMethod(ogg,cls,mid,valore)`, il secondo e il terzo parametro sono, rispettivamente, l'handle alla classe del glue code e l'handle al metodo **ioinvio(...)**, il quarto parametro è il valore che andrà a far parte del contenuto del NekoMessage (ovviamente varia a seconda dell'applicazione C che si sta integrando), il primo parametro identifica l'oggetto Java che per primo ha chiamato il metodo nativo che utilizza la libreria. É importante non dimenticarsi di trovare il riferimento all'oggetto chiamante all'interno del layerC, è solo grazie ad esso che siamo in grado di richiamare l'esatto metodo Java. Per capire meglio l'importanza di questo parametro basta osservare che JNI mette a disposizione tante altre funzioni utilizzabili per le chiamate dei metodi Java, ad esempio esiste la funzione:

`env->CallVoidMethod(cls,mid,valore);`

tuttavia, questa, come tutte le altre, è utilizzabile quando esiste una più semplice comunicazione tra C o C++ e Java e non, come in questo caso, un passaggio da Java a C e poi di nuovo da C a Java. Ecco in cosa consiste la differenza: nel primo caso, l'oggetto Java non esiste ancora e viene istanziato direttamente da JNI, nel nostro caso invece, l'oggetto Java è già istanziato ed è importante riferirsi ad esso per la chiamata al metodo.

Questi appena elencati sono tutti e soli i passi necessari alla libreria per permettere la comunicazione tra C e Java; per chiarire ancora meglio come ho programmato la libreria riporto la parte centrale del suo codice in Figura 5.6.
Se tutti i passi elencati vengono eseguiti senza che si verifichi alcun errore, la comunicazione tra C e Java è garantita.

```
if (JNI_GetCreatedJavaVMs(&jvms,5,&jvmCount)==0 && jvmCount>0)
{
      jvm=&jvms[0];
}
int result=jvm->GetEnv((void**) &env,JNI_VERSION_1_4);
if (ide==0) //sender ha ide=0, receiver ha ide=1
{
      cls = env->FindClass("classe del glue code del sender");
}
else
{
      cls = env->FindClass("classe del glue code del receiver");
}
if(cls!=0)
{
      mid=env->GetMethodID(cls,"ioinvio","(III)V");
      if (mid!=0)
      {
            env->CallNonvirtualVoidMethod(ogg,cls,mid,risp,ec,ecRisp);
            if (env->ExceptionOccurred())
            {
                  env->ExceptionDescribe();
                  env->ExceptionClear();
            }
      }else printf("mid è = 0\n");
}else printf("cls = 0\n");
```

Figura 5.6: Porzione di codice della libreria che si occupa del passaggio dal layerC al glue code

Lo sviluppatore che intende utilizzare questa libreria deve apportare alcune piccole modifiche; esse si riferiscono soprattutto alle signature di funzioni e metodi in essa definiti e utilizzati che, ovviamente, cambiano in base ai parametri da prendere in input. In particolare:

- modificare la signature della funzione definita nella libreria in modo che prenda in input gli esatti valori che costituiranno il contenuto del NekoMessage; attenzione però a non dimenticare il riferimento all'oggetto e l'identificatore del processo;
- inserire l'esatto nome della classe del glue code nella chiamata alla funzione `FindClass(...)`;
- modificare il terzo parametro specificato nella signatura della funzione `GetMethodID`, specificando l'esatto numero e l'esatto tipo di dati presi in input dal metodo **ioinvio(...)**;
- inserire l'esatto numero e gli esatti tipi dei parametri da passare in input al metodo **ioinvio(...)** nella chiamata alla funzione JNI `CallNonvirtualVoidMethod(...)`.

Ci si potrebbe chiedere come mai abbia utilizzato il passaggio attraverso una libreria per permettere la comunicazione tra C (o C++) e Java e non l'abbia fatto direttamente da C; i motivi sono essenzialmente due: da una parte perchè in questo modo vado a modificare il meno possibile il codice sorgente dell'algoritmo in C (o C++) da valutare, dall'altra perchè in questo modo ho fornito un oggetto, la libreria appunto, non dipendente dall'algoritmo che si intende analizzare ma riutilizzabile per l'analisi di qualsiasi algoritmo, ovviamente apportando le opportune modifiche.

5.3 Analisi quantitativa e NekoC

Fino ad ora non ho fatto riferimento su come la mia estensione interagisca con il pacchetto NekoStat per un'analisi quantitativa dell'algorit-

mo. In realtà, così come succede per le reti e i layer dell'applicazione Neko, gli strumenti messi a disposizione da NekoStat non sono affatto influenzati dalla presenza degli oggetti C e C++ che la mia estensione prevede; questo sempre grazie all'uso del glue code come elemento software di integrazione tra Neko e l'algoritmo in C.
Le aggiunte da effettuare a quanto visto fino ad ora sono veramente poche; vediamo comunque i passi da seguire per effetture un'analisi quantitativa di un algoritmo in C o C++ inserito come layerC di un'applicazione Neko:

1. definire nel glue code un metodo, **writeStat(String evento[])**, con il solo compito di richiamare il metodo **log(Evento)** dello **StatLogger** e che prenda in input una stringa che identifichi l'evento che si è verificato;

2. definire nella libreria C++ una funzione che prenda in input una stringa che identifichi l'evento avvenuto e richiami il metodo **writeStat(String evento[])** del glue code;

3. nel codice del layerC, individuare i punti opportuni in cui si vuole segnalare un evento e, invece di introdurre la chiamata al metodo **log(Evento)** dello **StatLogger**, come succede per le normali applicazioni Neko, inserire la chiamata alla nuova funzione definita nella libreria C++ fornendole in input la stringa che identifica l'evento. Sarà quest'ultima che si occuperà di richiamare il metodo **writeStat(String evento[])** che, a sua volta, riporterà l'evento allo **StatLogger**.

La cosa è abbastanza semplice: sappiamo già come si richiama un metodo Java dalla libreria C++, il codice del layerC, ancora una volta, subirà soltanto una piccola modifica, e nel glue code verrà solamente inserito un nuovo metodo.
C'è soltanto un'osservazione da fare, che deriva dall'uso delle stringhe per la segnalazione dell'evento: come ho già accennato nel capitolo

precedente, il tipo stringa Java non è lo stesso di quello in C, occorre quindi fare attenzione quando dalla libreria C++ si passa un parametro di tipo stringa al glue code.
Ciò che viene passato dal layerC alla libreria C++ è un array di caratteri, il problema sorge quando dalla libreria si richiama il metodo **writeStat(String evento[])**; ecco come ho evitato il problema di tipi:

1. ho costruito un array di oggetti String, utilizzando la funzione JNI

   ```
   NewObjectArray(1,
                  env->FindClass(''java/lang/String'',
                  env->NewStringUTF('''')
                  ).
   ```

   La funzione che appare come terzo parametro

   ```
   env->NewStringUTF('''')
   ```

   converte un array di caratteri C in una jstring[2] usando l'Unicode Trasformation Format (UTF). Unicode è un codice standard che assegna ai caratteri codici formati da due byte[3].

2. ho convertito l'array di caratteri ottenuto dal layerC in una jstring e l'ho inserito nell'array costruito al passo precedente, tutto ciò con la chiamata alla funzione

   ```
   env->SetObjectArrayElement(array_di_ogg, 0,
                              env->NewStringUTF(array_char)
                              ).
   ```

3. utilizzando la funzione

   ```
   env->CallNonvirtualVoidMethod(ogg,cls,mid,ret)
   ```

[2]jstring è il tipo JNI che rappresenta l'oggetto String del linguaggio Java.
[3]Java memorizza i caratteri di tutte le stringhe nei loro valori Unicode.

ho chiamato il metodo **writeStat(String evento[])** passandogli in input l'array di oggetti String costruito nei due passi precedenti. I parametri *ogg*, *cls* e *mid* sono quelli visti per la chiamata del metodo **ioinvio(..)**.

Nel capitolo seguente mostrerò un esempio di utilizzo della mia estensione integrando tra i layer di Neko un algoritmo di message freshness detection (ovviamente scritto in C); eseguirò anche un'analisi quantitativa di questo algoritmo, mostrando, più in dettaglio, le modifiche da apportare al codice del layerC, alla libreria e al glue code.

Capitolo 6

Esempio di utilizzo

In questo capitolo, oltre a mostrare l'utilizzo della nuova estensione al framework Neko, presenterò una serie di risultati, ottenuti da simulazioni, che hanno due scopi principali:

1. mostrare che il processo di analisi quantitativa degli algoritmi distribuiti, scritti in C o C++, non è influenzata dai nuovi elementi introdotti e che il livello di intrusività degli strumenti utilizzati per questo tipo di analisi non cambia;

2. dimostrare l'opportunità del mio lavoro, ovvero, dimostrate come, grazie alla nuova estensione di Neko, è possibile evitare di incorrere in situazioni indesiderate quali: la possibile introduzione di errori nell'algoritmo distribuito in fase di traduzione al linguaggio Java e l'impossibilità di rappresentare in Java alcuni oggetti facilmente rappresentabili in C.

Per dimostrare quanto detto, mostrerò come integrare tra i layer di Neko un algoritmo distribuito scritto in C, l'algoritmo in questione è un meccanismo per la message freshness detection in ambito ferroviario ([5, 43]): ASFDA (*Available and Safe Freshness Detection Algorithm*).

6.1 I timing failure detector

I *timing failure detector* sono meccanismi per il rilevamento di fallimenti di tipo timing; questo tipo di rilevamento è spesso utilizzato in quei sistemi in cui il fattore tempo è fondamentale, i sistemi *real-time*. Questi sistemi vengono classificati in ([3]):

- sistemi *hard real-time*, un sistema è di questo tipo quando il mancato rispetto delle specifiche temporali ha un alto costo associato e quindi deve essere evitato; questi sistemi sono progettati per prevenire fallimenti di tempo.

- sistemi *soft real-time*, in questi sistemi il mancato rispetto dei vincoli temporali è considerato accettabile;

- sistemi *mission-critical real-time*, questi sistemi rappresentano un pò una via di mezzo tra i sistemi hard e soft real-time; il fallimento dei requisiti temporali può avere un costo associato. In questi sistemi, i fallimenti dei requisiti temporali devono verificarsi soltanto come eccezioni del comportamento corretto del sistema.

In seguito ad un rilevamento di un fallimento di tipo timing, le azioni da intraprendere sono molteplici, dalla riconfigurazione del sistema (per garantire deadline meno stringenti) fino allo shutdown safe. L'utilizzo dei timing failure detector, come ho già detto, è spesso legato ai sistemi real time; questi sistemi sono di solito definiti attraverso azioni temporizzate, ossia azioni che si devono verificare all'interno di un intervallo temporale $[t_{liveness}, t_{deadline}]$. Il timing failure detector si occupa di verificare che l'azione temporizzata termini all'interno dell'intervallo definito dalla specifica del sistema; se consideriamo l'intervallo temporale di prima, si controlla che l'azione termini prima di $t_{deadline}$.
Questi failure detector, così come i failure detector per crash, possono commettere errori di tipo *falso positivo* e *falso negativo*. I falsi positivi

corrispondono alle situazioni in cui il servizio offerto dal sistema è considerato *late* senza esserlo. I falsi negativi corrispondono alle situazioni in cui il servizio offerto è considerato valido quando invece è late; in applicazioni critiche i rischi maggiori sono spesso legati a questo tipo di errore. Un falso negativo può condurre a comportamenti non corretti del sistema; errori di questo tipo devono essere evitati in applicazioni safety-critical (come nel caso descritto in questo capitolo). Un falso positivo invece, porta solitamente a diminuire la disponibilità del servizio. Anche per i timing failure detector si può parlare di proprietà di *completezza* e *accuratezza*, che in questo caso sono ovviamente legate a concetti temporali. Per definire queste proprietà supponiamo di poter assumere che le azioni temporizzate sono caratterizzate da un evento di terminazione *end*, che secondo la specifica deve verificarsi prima dell'istante di tempo reale $t_{deadline}$. Con queste assunzioni, definiamo:

Proprietà di completezza: esiste un T_{max} tale che, per ogni azione temporizzata, il timing failure viene rilevato entro l'istante di tempo $t_{deadline} + T_{max}$.

Proprietà di accuratezza: esiste un T_{min} tale che un'azione temporizzata che termina prima dell'istante $t_{deadline} - T_{min}$ è considerata valida.

La completezza, così come per i crash failure detector, è legata alla safety del sistema; l'accuratezza è legata alla liveness. Ovviamente, si cercano dei meccanismi che garantiscano sia la completezza che l'accuratezza, garantirne soltanto una è di scarsa utilità.
Un tipico scenario in cui è necessario un timing failure detector è quello della *message freshness detection*; in questo caso, si cercano dei metodi per la valutazine della freshness dei messaggi ricevuti. La freshness di un messaggio può essere valutata come la differenza tra il tempo reale in cui l'informazione è stata creata dal mittente, t_{create}, e il tempo reale in cui la stessa informazione è ricevuta dal destinatario, $t_{received}$.

Un meccanismo di message freshness detection deve valutare se l'informazione ricevuta rispetta il vincolo $t_{received} \leq t_{create} + maxDelay$, il valore *maxDelay* è definito nella specifica del sistema.

6.2 ASFDA

ASFDA, Available and Safe Freshness Detection Algorithm, è un meccanismo per la message freshness detection in ambito ferroviario ([5, 43]); è nato come layer di uno stack di protocolli da utilizzare in applicazioni safety-critical ferroviarie. ASFDA è un timing failure detector che viene utilizzato per stabilire la connessione tra due entità, un Sender e un Receiver, e per effettuare controlli sulla freshness dei messaggi ricevuti, prima di passarli ai livelli superiori dello stack. La specifica del sistema, inoltre, prevede di definire un valore di tempo massimo, *maxDelay*, in modo tale da chiudere la connessione se non vengono ricevute nuove informazioni entro quel tempo stabilito.
Una connessione ASFDA è formata da due processi comunicanti, un Sender e un Receiver; entrambi i processi hanno una durata costante dei cicli di esecuzione definita dalla specifica del sistema. Durante una connessione, il Sender invia al termine di ogni ciclo, una nuova informazione al Receiver. Il processo Receiver sarà caratterizzato, in ogni istante, dal proprio numero del ciclo di esecuzione, chiamato Elaboration Cycle Receiver ($EC_{receiver}(t)$), così come il processo Sender avrà associato un valore $EC_{sender}(t)$.
ASFDA è basato su un meccanismo che consente al processo Receiver di calcolare il valore previsto del ciclo elaborativo attuale del Sender, il valore $VEC(t) = EC^{prev}_{sender}(t)$. Il processo Sender invia messaggi con associato un timestamp, impostato con il valore $EC_{sender}(t)$ ossia il numero di ciclo in cui il messaggio è stato spedito. Grazie a questa informazione, il Receiver può decidere in base al valore *VEC(t)*, se accettare il messaggio oppure chiudere la connessione.

Lo scopo principale per cui è stato realizzato ASFDA è quello di ottenere il massimo della *safety*; il tipo di applicazione per cui è stato creato, il sistema ferroviario, è un tipico ambiente *safety-critical real-time*: il mancato rispetto dei vincoli temporali del sistema, se non rilevato e gestito nella maniera opportuna, può portare a gravi conseguenze.
Per dimostrare la safety dell'algoritmo in [5, 43], è stato dimostrato che i messaggi con ritardi superiori al valore *maxDelay*, imposto dalla specifica, non vengono mai accettati come validi; questo perchè il processo Receiver utilizza un meccanismo per il calcolo di $VEC(t)$ in grado di garantire che $\forall t VEC(t) \geq EC_{sender}(t)$ (si veda [5, 43] per la dimostrazione).
Per garantire una maggiore disponibilità del servizio è stata proposta una modifica all'algoritmo base; essa prevede che il meccanismo di sincronizzazione utilizzato per il calcolo di $VEC(t)$, può essere eseguito più volte all'interno della stessa connessione, in particolare ogni tempo T_{resync}, in modo da minimizzare la differenza tra il valore di $VEC(t)$ e quello di $EC_{sender}(t)$.
In Figura 6.1, Figura 6.2 e Figura 6.3 sono rappresentati gli scambi di messaggi di tre tipiche situazioni in cui si può trovare l'algoritmo. Per i particolari dell'algoritmo vedere [5, 43].

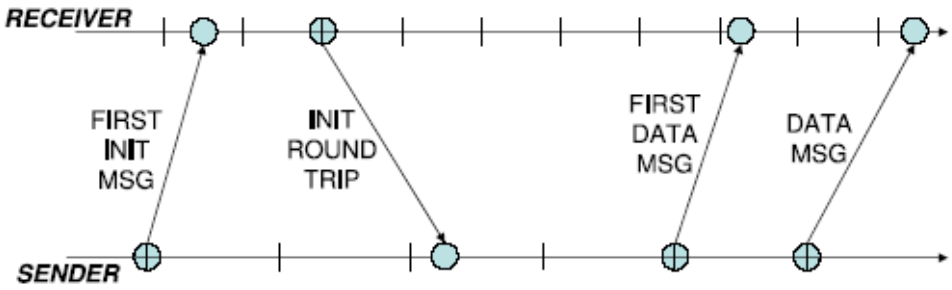

Figura 6.1: Diagramma spazio temporale di un periodo di sincronizzazione per la creazione di una nuova connessione ASFDA

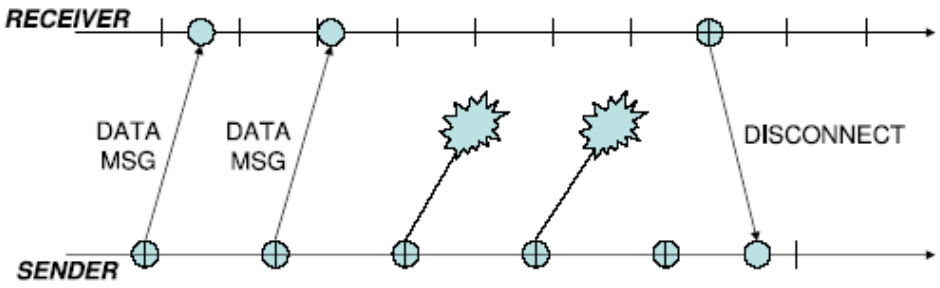

Figura 6.2: Diagramma spazio temporale del periodo finale di una connessione ASFDA; la disconnessione è causata dalla perdita di due messaggi consecutivi

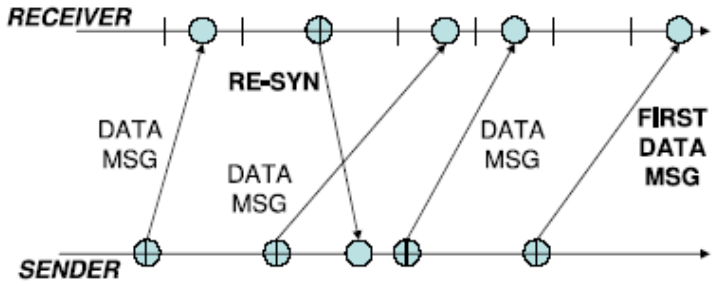

Figura 6.3: Diagramma spazio temporale di una resincronizzazione durante una connessione ASFDA

6.3 Integrazione dell'algoritmo tra i layer di Neko

Il primo passo da eseguire per permettere l'inserimento di ASFDA tra i layer di Neko, è distinguere il processo Receiver dal processo Sender e, per ognuno di essi, distinguere la parte di codice che si occupa della ricezione dei messaggi dalla rete, da quella che si occupa della valutazione dei messaggi ricevuti e dell'invio dei messaggi di risposta nella rete. A questo punto, modifichiamo le signature dei metodi dedicati alla ricezione dei messaggi, in modo che siano pronti a ricevere le informazioni provenienti dal glue code. Come ultima modifica, sostituiamo la chiamata al metodo send(...) con quella alla libreria C++ vista nel capitolo precedente, ricordandoci ovviamente l'identificatore del processo e il riferimento all'oggetto chiamante (come ho spiegato nel capitolo precedente, questo riferimento è necessario per poter richiamare l'esatto metodo Java del glue code). A questo punto, per permettere l'integrazione di ASFDA tra i layer di Neko ho realizzato i layer **SfdaReceiver** e **SfdaSender** che svolgono il ruolo di **glue code**; compito di questi layer è permettere il passaggio dei messaggi dalla rete al layerC, contenente ASFDA, e da quest ultimo alla rete. In particolare, ho implementato, per ognuno di essi

- il metodo **deliver(NekoMessage m)** con il compito di scompattare il **NekoMessage** proveniente dalla rete e inviare al layerC soltanto i dati di interesse, come l'EC del messaggio ricevuto;
- il metodo **run()** con il compito di avviare l'esecuzione del layerC;
- il metodo **ioinvio(...)** che riceve in input i parametri che andranno a comporre il **NekoMessage** e, una volta costruito il messaggio, lo invia sulla rete.

L'ultima cosa da fare è adattare la signatura della funzione definita nella libreria C++ per permettergli di ricevere gli esatti valori da inviare

al layer **SfdaReceiver** o al layer **SfdaSender**. La funzione principale di questa libreria ha, come già detto nel capitolo precedente, il compito di "riferirsi" alla JVM in esecuzione e richiamare il metodo opportuno della classe SfdaReceiver o SfdaSender per la creazione del NekoMessage e l'invio del messaggio nella rete; l'identificatore di processo ricevuto da ASFDA permette di identificare l'esatto **NekoProcess** e quindi l'esatto metodo **ioinvio(...)** da richiamare.
Quanto fatto basta per avviare l'applicazione Neko ed effettuare un'analisi qualitativa; ulteriori ritocchi sono invece necessari per l'analisi quantitativa con NekoStat.
In questo caso, le modifiche da apportare al layerC sono le stesse da apportare a qualsiasi algoritmo distribuito Java: inserire una chiamata ad un metodo per la segnalazione di un evento. La differenza consiste nel metodo da chiamare: mentre in un qualsiasi algoritmo Java si richiama il metodo **log(Evento)** dello **StatLogger**, in questo caso richiamo la funzione **statneko(jobject ogg, char buffer[10])** (Figura 6.4), che definisco nella libreria C++. Lo scopo di questa funzione è richiamare un metodo Java, **writeStat(String evento[])**, che definisco appositamente per l'analisi quantitativa e che a sua volta ha il compito di richiamare il metodo **log(Evento)** dello StatLogger. Come si vede dalla Figura 6.4, per la chiamata della funzione **statneko(jobject ogg, char buffer[10])** non è necessario l'identificatore del processo: le misure di interesse all'analisi quantitativa di ASFDA (così come sono definite nel paragrafo seguente) possono essere espresse basandoci sul solo processo Receiver, il processo di riferimento, quindi, è già conosciuto a priori.

6.4 Analisi quantitativa

Una volta completata l'integrazione di ASFDA tra i layer di Neko, ho effettuato l'analisi quantitativa dell'algoritmo utilizzando il pacchetto NekoStat. In particolare, ho deciso di confrontare i risultati ottenuti

```
void statneko(jobject ogg, char buffer[10])
{
 ...
 cls = env->FindClass("lse/neko/mio/SfdaReceiver");
 if(cls!=0)
 {
      jobjectArray ret=(jobjectArray)env->NewObjectArray(1,
                        env->FindClass("java/lang/String"),
                        env->NewStringUTF(""));
      env->SetObjectArrayElement(ret,0,env->NewStringUTF(buffer));
      mid=env->GetMethodID(cls,"writeStat","([Ljava/lang/String;)V");
      if (mid!=0)
      {
            env->CallNonvirtualVoidMethod(ogg,cls,mid,ret);
            if(env->ExceptionOccurred())
            {
                  env->ExceptionDescribe();
                  env->ExceptionClear();
            }

      }
 }
}
```

Figura 6.4: Porzione di codice della funzione definita nella libreria C++ che si occupa di richiamare il metodo writeStat del layer SfdaReceiver

dalle simulazioni da me effettuate, con quelli forniti in [1], in cui compare una prima analisi quantitativa di ASFDA con NekoStat; la sua valutazione, però, ha richiesto la traduzione dell'algoritmo originale in Java.
L'architettura utilizzata per le simulazioni è mostrata in Figura 6.5, dove con il termine Sender e con il termine Receiver mi riferisco al glue code e al layerC rispettivamente del lato Sender e Receiver di ASFDA.
L'analisi quantitativa dell'algoritmo si rivolge a una delle metriche che permettono di definire la *disponibilità* del meccanismo; la metrica in questione è

- T_{sync} il tempo necessario per stabilire una nuova connessione;

Gli eventi utili per ricavare tale metrica sono i seguenti:

- **StartSynchronization** evento corrispondente alla richiesta di stabilire una nuova connessione, ovvero all'invio al **Sender**, da parte del **Receiver**, di un messaggio DISCONNECT.

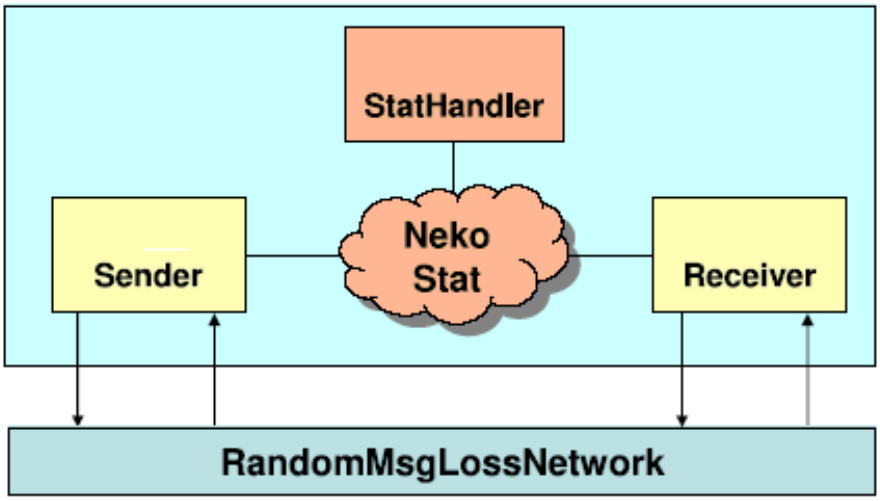

Figura 6.5: Architettura NekoStat utilizzata per le simulazioni

- **EndSynchronization** evento corrispondente all'attivazione della connessione, ovvero alla ricezione da parte del **Receiver** del primo FIRST DATA MSG valido per una nuova connessione.

La valutazione di T_{sync} avviene nel seguente modo:

- $T_{sync} = t^i_{end} - t^i_{start}$, il tempo per instaurare una nuova connessione è il tempo che intercorre tra gli eventi **StartSynchronization** e **EndSynchronization** della connessione *i-esima*;

Nella simulazione ho utilizzato una rete simulata, **RandomMsgLossNetwork**, realizzata in [1]; è una rete che presenta ritardi esponenziali di media *meanDelay*, con un valore minimo *minDelay* e con probabilità P di perdita del singolo messaggio durante il trasferimento; la rete è caratterizzata inoltre da alcuni periodi in cui i messaggi vengono persi con maggiore probabilità, chiamati periodi di *burst* ([1]).
Lo **StatHandler** utilizzato per le simulazione viene costruito direttamente dalla misura sopra definita, è compito del pacchetto NekoStat istanziare in maniera automatica le opportune strutture di supporto

all'analisi. Inoltre, la misura di T_{sync} si basa sul clock del solo processo Receiver, quindi non è stato necessario utilizzare metodi per la sincronizzazione dei clock.

6.4.1 Simulazioni

Come ho già detto, uno degli scopi delle simulazioni è dimostrare che le modifiche apportate al framework Neko non influenzano affatto il processo di analisi quantitativa e che il livello di intrusività di questi strumenti non cambia. Per ottenere questi risultati ho organizzato un gruppo di simulazioni per la valutazione di T_{sync}:

- **Valutazione di** T_{sync}: i risultati ottenuti da questo gruppo di simulazioni forniscono i tempi necessari per stabilire una nuova connessione, in presenza di ritardi di rete variabili e con probabilità di perdità di messaggi uguale a zero;

La condizione di stop utilizzata per le simulazioni è quella sulle quantità analizzate, la simulazione viene interrotta quando i semintervalli di confidenza di livello 95% sulle metriche analizzate sono inferiori al 5% della media stimata della metrica.

$meanDelay \in \{150,200,250,300\} msec$
$minDelay = 10msec$
$maxDelay \in \{1200,1500,1800,2100\} msec$

Tabella 6.1: Tabella dei parametri utilizzati per la valutazione della metrica T_{sync}

6.4.2 Risultati

I parametri utilizzati per le simulazione sono quelli delle Tabella 6.1. I risultati ottenuti dalla valutazione della metrica T_{sync} sono rappre-

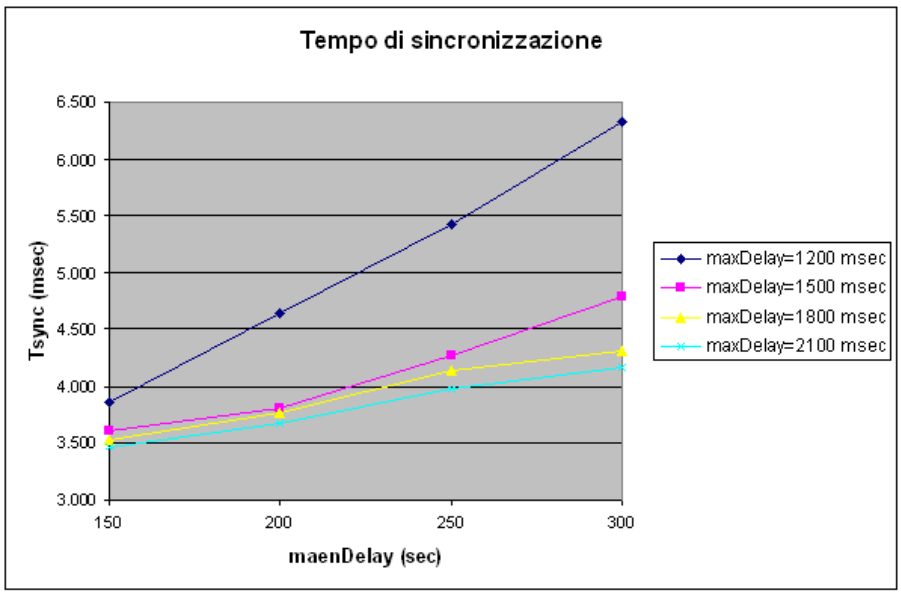

Figura 6.6: Grafico dei risultati delle simulazioni per la valutazione della metrica T_{sync}

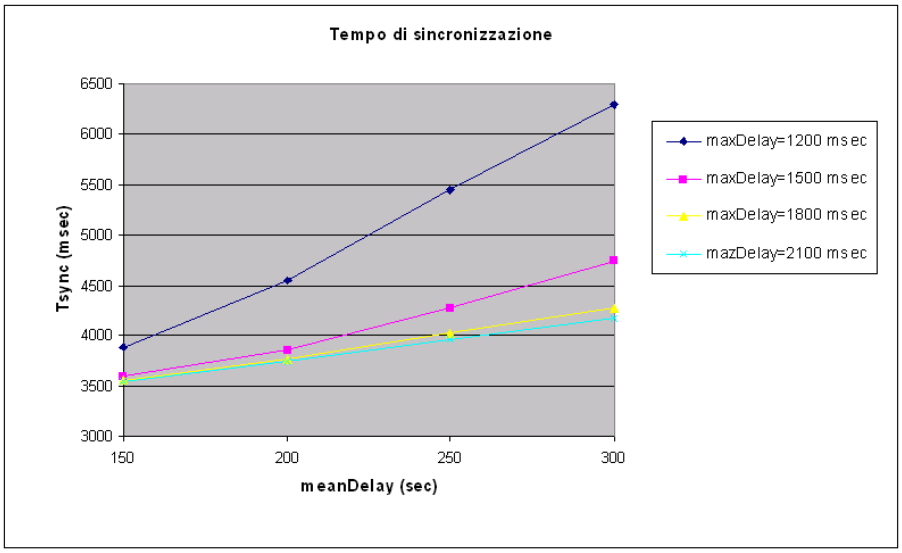

Figura 6.7: Grafico dei risultati delle simulazioni per la valutazione della metrica T_{sync} ottenuti in [1]

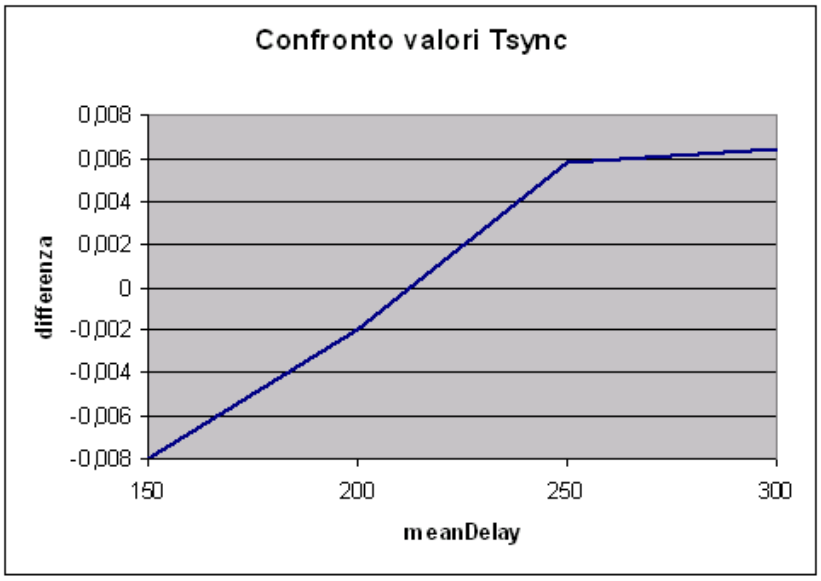

Figura 6.8: Confronto tra i valori ottenuti dalle simulazioni effettuate utilizzando NekoC e i valori ottenuti in [1]

sentati in Figura 6.6; in questa figura è rappresentato il grafico del tempo necessario per stabilire una nuova connessione su diversi valori del parametro *maxDelay* dell'algoritmo, in funzione del ritardo medio di trasmissione della rete simulata, *meanDelay*. I risultati ottenuti possono essere confrontati con quelli forniti in [1], rappresentati nel grafico in Figura 6.7; il confronto tra i due risultati è riportato in Figura 6.8, si vede facilmente come i risultati delle due simulazioni siano simili (la loro differenza è vicinissima allo zero) dimostrando come l'integrazione tra i layer di Neko di una algoritmo scritto in C non influenzi l'analisi quantitativa dell'algoritmo e dimostrando, ancora una volta, l'affidabilità dei risultati ottenuti con NekoStat.

6.5 La correttezza degli algoritmi

Uno dei fattori principali che può compromettere la safety di ASFDA è l'errata approssimazione delle variabili reali. É noto infatti, che la rappresentazione di un numero reale in un elaboratore può comportare errori di precisione, in quanto i numeri reali rappresentabili in un calcolatore sono solo un sottoinsieme di tutto l'insieme dei numeri reali. Nel caso in cui un elaboratore si trovi in presenza di un numero reale non rappresentabile è necessario ricorrere ad un'approssimazione del numero in questione; esistono quattro tipi di approssimazione ([44]):

1. approssimazione al numero rappresentabile più vicino (TONEAREST);
2. approssimazione al più grande numero rappresentabile inferiore (DOWNWARD);
3. approssimazione al più piccolo numero rappresentabile superiore (UPWARD);
4. approssimazione verso zero (TOWARDZERO), ovvero approssimare i numeri positivi in modo DOWNWARD e quelli negativi in modo UPWARD.

Dal punto di vista matematico l'approssimazione TONEAREST è la migliore in quanto l'errore commesso non è più grande di quello relativo agli altri metodi; tuttavia, non è detto che questa tecnica di approssimazione sia applicabile in tutti i casi in cui è richiesta un'approssimazione dei valori reali. Nel caso dell'algoritmo di freshness detection che si sta valutando ([5]), ad esempio, non è opportuno usare questo metodo di approssimazione. Basta ricordare che lo scopo di ASFDA è quello di valutare la freschezza dei messaggi ricevuti e la sua specifica prevede di chiudere la connessione se non vengono ricevuti nuovi messaggi entro un tempo massimo stabilito; è chiaro che, poichè si cerca un'approssimazione superiore del ritardo dei messaggi, le operazioni

matematiche sulle variabili reali devono essere definite utilizzando approssimazioni DOWNWARD e UPWARD e non TONEAREST, in modo da sapere sempre se un certo valore, per poter essere rappresentato, è stato aumentato o diminuito. Per capire meglio l'importanza di questa argomentazione, vediamo due esempi che illustrano le situazioni in cui possiamo trovarci se non viene fatta la giusta approssimazione dei valori di *delta*[1] e *delta massimo*[2]:

- Supponiamo di poter rappresentare correttamente tutti i numeri fino alla quarta cifra decimale compresa e di usare il tipo di approssimazione TONEAREST. Supponiamo che il calcolo di *delta massimo* restituisca il numero 1.35426 che sarà approssimato in 1.3543. Supponiamo, ancora, che arrivi un messaggio che porta ad un valore di *delta* pari a 1.35428, in seguito all'approssimazione *delta* sarà pari a 1.3543 quindi uguale a *delta massimo*, per cui il messaggio sarà giudicato fresco quando in realtà andrebbe scartato.

Quindi, per quanto riguarda l'approssimazione da usare per il calcolo di *delta massimo*, la più sicura è quella di tipo DOWNWARD: in questo modo non si corre mai il rischio di aumentarne il valore. Rifacendoci all'esempio precedente, utilizzando l'approssimazione DOWNWARD, il valore di *delta massimo* sarebbe stato 1.3542 e non 1.3543.
Le considerazioni da fare non finiscono quì, consideriamo ancora un altro esempio:

- Supponiamo di trovarci nelle stesse condizioni dell'esempio precedente e che il calcolo di *delta massimo* restituisca il valore 1.35421; con l'approssimazione DOWNWARD otteniamo il valore 1.3542. Supponiamo ora che arrivi un messaggio per cui *delta* valga 1.35424;

[1] *delta* è la stima del ritardo subito dal messaggio.

[2] *delta massimo* è il ritardo massimo del messaggio, convertito in cicli sender; vedere [5] per ulteriori dettagli.

con l'approssimazione TONEAREST il valore di *delta* sarà pari a 1.3542. In seguito a questi arrotondamenti il messaggio è considerato fresco quando in realtà andrebbe scartato.

É chiaro che per quanto riguarda l'approssimazione da usare per tutte le quantità coinvolte nel calcolo di *delta*, quella più adatta è l'approssimazione di tipo UPWARD: in questo modo non si corre mai il rischio di diminuirne il valore. Un ragionamento simile può essere fatto per tutti i valori reali coinvolti nell'algoritmo di message freshness detection.
La scelta del linguaggio C per la scrittura dell'algoritmo è stata quindi dettata dalla necessità di evitare situazioni simili a quelle viste nei due esempi precedenti, in modo da garantire la safety del sistema e aumentarne la disponibilità.
Traducendo l'algoritmo in Java, purtroppo, non è più garantita l'assenza di simili errori di approssimazione: l'unica approssimazione possibile in Java è quella TONEAREST.

6.5.1 Confronto

Ho svolto una serie di esperimenti mettendo a confronto l'algoritmo di message freshness detection originale ([5]) e quello tradotto in Java ([1]); ho modellato il sistema, composto dai due processi Sender e Receiver nel seguente modo (Figura 6.9):

- da una parte ho inserito come layerC di Neko il lato *Sender* di ASFDA;

- dall'altra ho costruito un processo composto da due livelli sovrapposti; il livello base è un **Multiplexer**, esso si occupa di inoltrare i messaggi in arrivo ai layer superiori, dove troviamo un layer contenente il lato *Receiver* di ASFDA (compreso di libreria e glue code) e uno contenente il lato *Receiver* tradotto in Java. Il Multiplexer è stato programmato in modo da terminare l'applicazione

non appena i due layer sovrastanti rispondono con due messaggi di tipo diverso.

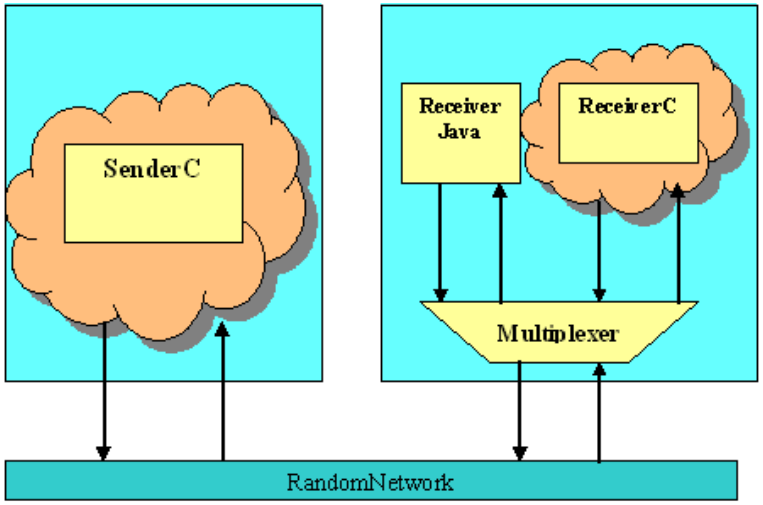

Figura 6.9: Architettura Neko per gli esperimenti di confronto

Ho eseguito gli esperimenti utilizzando la rete **RandomNetwork**, rete simulata messa a disposizione da Neko, e utilizzando valori variabili del parametro *maxDelay*.
Osservando i risultati degli esperimenti eseguiti ho notato come, usando la tecnica di approssimazione di tipo TONEAREST, l'algoritmo in Java è portato a compiere errori di valutazione sulla freschezza dei messaggi; in particolare ho osservato due possibili scenari di errore, il primo porta a giudicare vecchi messaggi in realtà freschi, il secondo, ancora più grave, porta a giudicare come freschi messaggi vecchi.
La frequenza con cui si verificano questi scenari non è particolarmente alta, ma basta per avere un'indicazione delle situazioni in cui ci possiamo trovare quando si decide di tradurre in Java un qualsiasi altro algoritmo.
Per capire meglio i tipi di errore in cui incorre l'algoritmo in Java, vediamo qual'è il giusto comportamento, ottenuto utilizzando le opportune

tecniche di approssimazione[3], che il processo *Receiver* deve tenere in due diversi scenari che si sono presentati per alcuni valori di *maxDelay*.

Primo scenario. In seguito all'arrivo di un messaggio di tipo DATA MESSAGE, il processo *Receiver* aggiorna il valore di *delta* che risulta essere minore di *delta massimo*; nel ciclo successivo, il processo non riceve alcun messaggio e questo porta ad incrementare ancora il valore di delta per verificare se la connessione possa o meno continuare. L'aggiornamento di *delta* porta ad un valore minore/uguale di *delta massimo*, la connessione quindi non deve essere interrotta: nel ciclo successivo potrebbe arrivare un messaggio fresco.

Secondo scenario. In seguito all'arrivo di un messaggio di tipo DATA MESSAGE, il processo *Receiver* aggiorna il valore di *delta*, ottenendo un valore minore di *delta massimo*; per due cicli consecutivi il *Receiver* non riceve alcun messaggio e incrementa il valore di delta. Il secondo incremento porta ad un valore di *delta* maggiore di quello di *delta massimo*: la connessione viene chiusa e non vengono accettati altri messaggi.

In base ai valori di *delta* e *delta massimo* opportunamente approssimati quelli appena mostrati rappresentano i corretti comportamenti del processo *Receiver*. In Tabella 6.2 e Tabella 6.4 sono mostrati gli esatti confronti relativi rispettivamente al primo scenario e al secondo scenario, ottenuti utilizzando le giuste tecniche di approssimazione in relazione ai valori di *maxDelay* specificati.
I risultati ottenuti utilizzando l'architettura in Figura 6.9 mostrano che i due processi *Receiver*, quello originale e quello in Java, non seguono entrambi il comportamento appena descritto. In pratica, quello che

[3]Approssimazione UPWARD per il calcolo di *delta* e approssimazione DOWNWARD per il calcolo di *delta massimo*.

succede è che il *Receiver* in Java, approssimando tutti i valori in modo TONEAREST, si ritrova nel primo caso a chiudere una connessione che in realtà è ancora valida, nel secondo caso a rimanere in attesa di un messaggio che sappiamo già essere vecchio. Soltanto l'algoritmo originale riesce a valutare correttamente se la connessione va chiusa o meno, è il solo processo che si comporta come mostrato nei due scenari precedenti.
In particolare, in seguito agli esperimenti eseguiti, ho osservato il verificarsi di due comportamenti differenti:

1. per alcuni valori di *maxDelay*, quelli in Tabella 6.3, ci toviamo nel primo scenario descritto precedentemente: trascorso un ciclo senza l'arrivo di nessun messaggio di tipo DATA MESSAGE entrambi i processi aggiornano il proprio valore di *delta*; ecco cosa succede:

   - il Receiver dell'algoritmo in C, grazie agli opportuni arrotondamenti, ottiene un valore di *delta* minore/uguale del valore di *delta massimo*, decide quindi di continuare a rimanere in attesa di un messaggio DATA MESSAGE;
   - il Receiver dell'algoritmo in Java, a causa di errori di approssimazione, dovuti al fatto che in Java qualsiasi valore reale non esattamente rappresentabile viene approssimato in modo TONEAREST, ottiene un valore di *delta* maggiore del valore di *delta massimo* e di conseguenza invia al sender un messaggio DISCONNECT, quando in realtà la connessione è ancora valida.

2. per un valore di *maxDelay* pari a $889,9999999999864$ msec, ci toviamo nel secondo scenario precedentemente descritto: i due layer Receiver dopo che per due cicli consecutivi non ricevono alcun messaggio, aggiornano per due volte di seguito il valore di *delta*; ecco cosa succede:

- il Receiver dell'algoritmo in C ottiene un valore di *delta* maggiore di quello di *delta massimo* e invia al Sender il messaggio DISCONNECT;
- il Receiver dell'algoritmo in Java, a causa degli errori di approssimazione, ottiene un valore di *delta* minore/uguale del valore di *delta massimo* e rimane in attesa di un nuovo messaggio.

In entrambi i casi l'unico algoritmo a comportarsi correttamente è quello originale, scritto in C; l'algoritmo tradotto in Java a causa dell'approssimazione non controllata dei valori reali commette errori di valutazione chiudendo una connessione ancora valida o rimanendo in attesa di messaggi non più freschi.
I dati ottenuti dalle simulazioni che forniscono uno scenario del primo tipo sono riassunti in Tabella 6.3; confrontando i risultati della versione in C e di quella in Java con quelli riporatti in Tabella 6.2 si vede bene come l'unico algoritmo ad effettuare il giusto confronto è quello originale; l'algortimo tradotto in Java ottiene un valore di *delta* maggiore di *delta massimo* chiude quindi una connessione ancora valida. É chiaro che risultati di questo tipo, relativi al primo scenario, non rappresentano una situazione catastrofica ma danno un'indicazione su come la traduzione dell'algoritmo in Java fornisca la descrizione di un sistema meno "disponibile" di come si presenta nella realtà; la versione in Java di ASFDA è portato a commettere errori di tipo *falso positivo* giudicando vecchi messaggi in realtà freschi. Un'ultima osservazione che posso fare è che l'errore di approssimazione di delta, per i valori di *maxDelay* relativi alla Tabella 6.3, si ripercuote sulla disponibilità dell'algoritmo non poi così in là nel tempo, in realtà bastano meno di 200 cicli dell'algoritmo di message freshness detection perchè si verifichi tale situazione.
Il secondo scenario è quello più interessante, si verifica per un valore di *maxDelay* pari a $889,9999999999864$; il corretto funzionamento dell'algoritmo di message freshness detection ottenuto con le opportune

maxDelay	**Esatta approssimazione**
1240	*delta*≤*delta massimo*
1590	*delta*≤*delta massimo*
2290	*delta*≤*delta massimo*

Tabella 6.2: Esatto confronto tra *delta* e *delta massimo* ottenuto utilizzando le adeguate tecniche di approssimazione

maxDelay	**Versione C**	**Versione Java**
1240	*delta*≤*delta massimo*	*delta*>*delta massimo*
1590	*delta*≤*delta massimo*	*delta*>*delta massimo*
2290	*delta*≤*delta massimo*	*delta*>*delta massimo*

Tabella 6.3: Risultati della simulazione a confronto tra la versione ASFDA in C e la versione tradotta in Java

approssimazioni di *delta* e *delta massimo* è mostrato in Tabella 6.4, il risultato ottenuto utilizzando l'architettura in Figura 6.9 è riassunto in Tabella 6.5. I risultati riportati in quest'ultima tabella, confrontati con il corretto comportamento riportato in Tabella 6.4, dimostrano che l'unico algoritmo a comportarsi correttamente è quello scritto in C; l'algoritmo tradotto in Java rimane in attesa di un messaggio mentre, in realtà, la giusta approssimazione dei valori di *delta* e *delta massimo* suggerisce che la connessione deve essere chiusa. Il pericolo è che la versione in Java al ciclo successivo accetti un messaggio in realtà vecchio.
Questo tipo di errore di valutazione è molto più interessante del precedente: l'impossibilità, che si riscontra in Java, di poter utilizzare diverse tecniche di approssimazione fornisce un algoritmo che può commettere errori di *falso negativo*, mettendo in discussione la sicurezza del sistema. É ovvio che in un'applicazione safety-critical di questo tipo la safety del sistema deve essere garantita e quindi errori di questo tipo devono essere evitati.

Per capire l'importanza della corretta approssimazione dei valori reali, basti pensare a cosa potrebbe succedere se la traduzione in Java di ASFDA fosse inserita come layer dello stack di protocolli utilizzati nelle applicazioni ferroviarie: le conseguenze di un errore di tipo falso negativo sarebbero catastrofiche, si parla addirittura di perdite di vite umane. In conclusione la versione di ASFDA in Java fornisce la descrizione di un sistema che per alcuni valori di *maxDelay* commette errori di valutazione dei messaggi compromettendo la disponibilità e, cosa ancora più importante, la safety del sistema. Questo significa che la traduzione di algoritmi distribuiti, da diversi linguaggi di programmazione a Java,

maxDelay	**Esatta approssimazione**
889,9999999999864	*delta*>*delta massimo*

Tabella 6.4: Esatto confronto tra *delta* e *delta massimo* ottenuto utilizzando le adeguate tecniche di approssimazione per un valore di *maxDelay* pari a $899,9999999999864$

maxDelay	**Versione C**	**Versione Java**
889,9999999999864	*delta*>*delta massimo*	*delta*≤*delta massimo*

Tabella 6.5: Risultati della simulazione a confronto tra la versione originale di ASFDA e la versione tradotta in Java

può non essere praticabile data l'impossibiltà di rappresentare esattamente tutti gli oggetti definibili in altri linguaggi.
La cosa che viene da chiedersi a questo punto è come mai un tale problema di valutazione non si sia riscontrato durante la valutazione quantitativa di ASFDA, la risposta è semplice: il verificarsi di situazioni del tipo mostrate in questo paragrafo non sono così frequenti da compromettere l'analisi quantitativa dell'algortimo; sicuramente è compromessa la sua analisi qualitativa.

L'algoritmo di message freshness detection analizzato non è l'unico sistema che presenta queste caratteristiche, la maggior parte dei sistemi real-time e safety-critical sono scritti in C; l'estensione di Neko da me realizzata permette la valutazione di tali sistemi senza la necessità di tradurre gli algoritmi in Java, evitando quindi situazioni indesiderate come quelle riscontrate in questo paragrafo. Utilizzare NekoC per la valutazione di ASFDA o di qualsiasi altro algoritmo scritto in C o C++, garantisce che i risultati ottenuti dalla sua analisi, sia qualitativa che quantitativa, si riferiscono esattamente all'algoritmo che sarà usato nella realtà; qualsiasi sviluppatore che utilizza NekoC per l'analisi dei suoi algoritmi non dovrà aspettarsi sorprese nel momento in cui utilizzerà il suo algoritmo nel sistema reale.

Capitolo 7

Conclusioni

Il lavoro presentato in questa tesi ha avuto come scopo la definizione e realizzazione di un'estensione alle metodologie utilizzate per l'analisi qualitativa e quantitativa dei sistemi distribuiti. In particolare, questo lavoro ha richiesto una prima analisi dei concetti base della dependability, delle problematiche relative allo sviluppo dei sistemi distribuiti e dei modelli che permettono la descrizione di tali sistemi. É stato necessario, inoltre, lo studio delle problematiche relative all'integrazione di componenti COTS (*Commercial Off-The-Shelf*) all'interno di un sistema esistente e degli elementi software di integrazione, ossia di quegli elementi che rendono possibile la convivenza tra componenti diversi.
Ho rivolto quindi la mia attenzione ad uno degli strumenti più potenti che è possibile utilizzare per la definizione di algoritmi distribuiti e per la loro analisi, sia simulativa che sperimentale: il framework Neko.
Neko, insieme al pacchetto NekoStat, consente una valutazione completa, sia qualitativa che quantitativa, degli algoritmi distribuiti. Il limite di questo strumento era dovuto all'impossibilità di valutare algoritmi distribuiti scritti in linguaggi come il C o il C++; la valutazione di tali algoritmi richiedeva, fino ad oggi, la loro traduzione in Java, con tutte le conseguenze che questa traduzione comporta, ad esempio la possibilità di introdurre errori durante il processo di traduzione al linguaggio Java

e l'impossibilità di rappresentare in Java oggetti definiti in C o C++.
Con l'estensione da me realizzata, NekoC, si dà la possibilità di valutare anche questi algoritmi permettendo una riduzione dei costi di valutazione, in termini di tempo e di denaro, e una maggiore garanzia sui risultati ottenuti. L'utilizzo di elementi software di integrazione, quali il glue code, mi ha permesso di integrare gli algoritmi scritti in C o C++ tra i layer di Neko senza che questo comportasse alcuna modifica del framework stesso; in pratica, nessun componente di Neko (layer, processo o rete) è a conoscenza dei nuovi elementi introdotti.
A dimostrazione dell'uso del nuovo strumento e dell'opportunità di questo lavoro, ho esercitato NekoC per effettuare la simulazione di un meccanismo di freshness detection nelle sue versioni in C e in Java. L'analisi dei risultati delle simulazioni ha mostrato, da una parte che non è stata introdotta ulteriore intrusività, e dall'altra come, grazie all'utilizzo di NekoC, si riescono a evitare situazioni indesiderabili dovute alla traduzione degli algoritmi in Java.
Questo lavoro può essere proseguito generalizzando l'applicabilità dei concetti affrontati per rendere ancora più universale questo tool; a tale proposito esistono due percorsi da seguire:

- permettere l'integrazione, tra i layer di Neko, non di codice sorgente ma di codice eseguibile;
- esplicitare la potenzialità di NekoC come interfaccia, utilizzando il linguaggio C come "ponte" tra Java e altri linguaggi di programmazione.

Appendice A

Manuale d'uso

In queste pagine descriverò quali sono gli strumenti necessari per utilizzare NekoC e mostrerò i passi da eseguire per rendere possibile la comunicazione tra Java e C (o C++), tra C (o C++) e C++ e tra quest'ultimo e Java.

A.1 Installazione degli strumenti

Gli strumenti necessari per eseguire una valutazione di algoritmi distribuiti scritti in C o C++, sono i seguenti:

1. **Neko**, scaricabile da [45]; nella versione 0.9 è già disponibile il pacchetto **NekoStat**;
2. **Jace**, scaricabile da [41];

Per l'installazione di Neko eseguire la decompressione del file scaricato (l'ultima versione è **neko0.9.gz**), verrà direttamente creata la directory **neko**; eseguire un'installazione standard di Neko seguendo le istruzioni presenti nel file **neko/README**.
Per l'installazione di Jace, una volta scaricato il file **jace1_03.jar**, basta estrarlo utilizzando un qualsiasi tool zip, per esempio l'utility jar del JDK, e verrà direttamente creata la cartella **jace** il cui contenuto è

mostrato in Figura A.1. É importante leggere la documentazione di

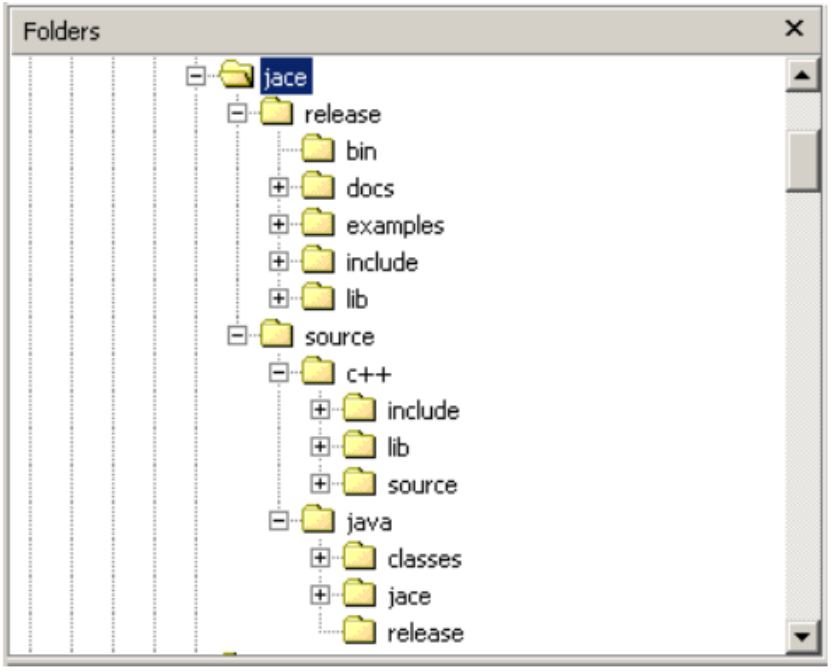

Figura A.1: Contenuto della cartella jace

Jace contenuta nella cartella **jace/release/docs**.
L'installazione di Neko richiede il JAVA SDK con versione superiore alla 1.3; l'installazione di Jace richiede la presenza di un compilatore C, in particolare: per il sistema operativo Windows è necessario il Visual C++6.0, o il Visual C++.NET(7.0 o 7.1) oppure ancora il GCC 3.x, quest'ultimo è necessario anche per il funzionamento di Jace sotto Linux.

A.2 Come utilizzare Jace all'interno di Neko

Una volta installati tutti gli strumenti di sviluppo, i passi da seguire sono i seguenti:

1. implementare le classi che avranno il ruolo di **glue code**, ricordandosi di definire come nativi i metodi del layerC;

2. creare una directory, chiamarla ad esempio **neko_jace**, in cui saranno inseriti tutti i file generati dall'uso del toolkit Jace e in cui sarà inserito il file contenente il codice dell'algoritmo in C o C++ che sarà inserito come layerC.

3. implementare le funzioni della libreria C++ seguendo i passi visti nel Capitolo 5 e compilare;

4. apportare le modifiche viste nel Capitolo 5 al codice C.

5. compilare le classi del glue code e utilizzare i tool PeerEnhancer, PeerGenerator e AutoProxy per la generazione di tutti i file necessari ad una comunicazione tra le classi Java e i metodi nativi; organizzare tutti i file generati da questi tool all'interno della cartella **neko_jace**; la cosa migliore da fare è includere le chiamate ai tool all'interno di un file **xml**, così com'è mostrato nella documentazione allegata a Jace;

6. compilare il file dell'algoritmo da integrare come layerC ricordandosi di specificare il link alla libreria di Jace e alla libreria C++ creata al passo 3.

Quanto detto basta per rendere possibile l'integrazione di un qualsiasi algoritmo scritto in C o C++ tra i layer di Neko, per maggiori dettagli sull'uso dei tool messi a disposizione da Jace consiglio di guardare gli esempi contenuti nelle seguenti cartelle:
jace/release/examples/peer_example1
jace/release/examples/peer_singleton.

Ringraziamenti

Giunta alla conclusione del mio percorso di studio vorrei ringraziare le persone che hanno reso possibile il raggiungimento di un traguardo tanto desiderato. In particolare voglio ringraziare papà, mamma e nonna Giuseppina, per aver creduto in me e per il sostegno morale ed economico che mi hanno dato; i miei fratelli e Emilio per essermi stati vicino in questi anni di studio e nel periodo di preparazione della tesi.

Bibliografia

[1] L. Falai. Metodologie e strumenti per la valutazione quantitativa sperimentale e simulativa di algoritmi distribuiti. Tesi di laurea. Università degli Studi di Firenze 2003-2004.

[2] B. Randell A. Avizienis, J. Laprie. Fundamental concepts of dependability. *Research Report N01145, LAAS-CNRS*, Apr. 2001.

[3] L.Rodrigues P.Verissimo. *Distribuited Systems for System Architects*. Kluwer Academic Publishers, 2001.

[4] William H. Sanders. *Introduction to Computer System and Network Validatio.* Disponibile all'indirizzo http://www.crhc.uiuc.edu/PERFORM.

[5] E. De Giudici. Progettazione ed analisi di un meccanismo per garantire la sicurezza in applicazioni ferroviarie. Tesi di laurea. Università degli Studi di Firenze 2002-2003.

[6] Brian Randell Algirdas Avizienis, Jean-Claude Laprie and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on dependable and secure computing*, january-march 2004.

[7] A. Coccoli. *Il problema del consenso nei sistemi distribuiti.* In *Rapporto interno CNUCE*, Aprile 1999.

[8] Galvin P.B. Silberschatz A. *Sistemi operativi.* Addison-Wesley, 5 edition, 1998.

[9] Tucci-Piergiovanni Baldoni, Marchetti. *Appunti Integrativi su Distemi Distribuiti*, A.A.2001/2002. Corso di Reti di Calcolatori.

[10] Peter Sturm Friedemann Mattern. From distribuited systems to ubiquitous computing. *The State of the Art, Trends and Prospects of Future Networked Systems.*

[11] M.Weiser. The computer for the 21st century. *Scientific American*, September 1991.

[12] M.S.Patterson. M.J.Fischer, N.A.Lynch. Impossibility of distribuited consensus with one faulty process. *Journal of the ACM. Vol.32(2)*, Apr. 1985.

[13] L.Stockmeyer. C.Dwork, N.Lynch. Consensus in the presence of partial synchrony. *Journal of the ACM. Vol.35(2)*, Feb. 1988.

[14] C.Fetzer. F.Cristian. The timed asynchronous distribuited system model. In *28th International Symposium on Fault-tolerant Computing.* IEE Computer Society Press, 1998.

[15] C.Almeida P.Verissimo. Quasi-synchronism: a step away from the traditional fault-tolerant real-time system models. *Bullettin of the Technical Committee on Operating Systems and Application Environments (TCOS)*, 1995.

[16] V.Hadzilacos and S.Toueg. *Fault-Tolerant Broadcast and Related Problems.*, chapter 16. Distribuited System, addison wesley edition, 1993.

[17] M.O.Rabin. Randomized byzantine generals. In *24th Annual Symposium on Foundations of Computer Science*, 1983.

[18] L.Lamport. *Tyme, clocks, and the ordering of events in a distributed system.* Communication of the ACM. Vol.21(7), July 1978.

[19] T.Chandra and S.Toueg. *Unreliable Failure Detectors for Reliable Distributed Systems.*, pages 225–267. Journal of the ACM, Mar. 1996.

[20] V.Hadzilacos T.Chandra and S.Toueg. *The weakest failure detector for solving consensus.*, volume 43(4). Journal of the ACM, July. 1996.

[21] R.Jhonson W.D.Oball II M.A.Qureshi M.Rai W.H.Sanders J.Couvillion, R.Freire and J.E.Tvedt. Performability modeling with ultrasan. In *IEEE Software, Special Issue on Software for Performance Analisys*, volume 8, no. 5, pages 69–80, Sept. 1991.

[22] Doyl J.M. Daly D., Deavours D.D. Mobius: An extensible framework for perfomance and dependability modeling. In *Eighth International Workshop on Petri Nets and Performance Models*, Sep. 1999.

[23] P.Urban. *Evaluating the performance of distribuited agreement algorithms: tool, methodology and case studies.* PhD thesis, École Polytechnique Fédérale de Lausanne, 2003.

[24] A.Schiper P.Urban, X.Defago. *Neko: a single environment to simulate and prototype distribuited algorithms.* PhD thesis, Feb. 2001.

[25] Y.Ishikawa H.Tezuka, A.Hori and M.Sato. Pm: An operating system coordinated high performance communication library. In *Proc.High-Performance Computing and Networking(HPCN 97 Europe)*, 1997.

[26] M.Hayden. The ensemble system. *Technical Report TR98-1662*, 1998. Cornell University.

[27] Colt Library website.
Disponibile all'indirizzo
http://dsd.lbl.gov/∼hoschek/colt.

[28] Colt Library API documentation.
Disponibile all'indirizzo http://dsd.lbl.gov/∼hoschek/colt/api/index.html.

[29] P.Oberndorf. Facilitating component-based software engineering:cots and open systems. In *Proceedings of the Fifth Internazional Symposium on Assessment of Software Tools (SAST'97)*, June 1997.

[30] Carney D. e Pollak B. Wallnau K.C. *How COTS Software Affects the Design of COTS-Intensive Systems.* Technical report, Software Engineering Institute, Carnegie Mellon University, USA, Jun. 1998.

[31] Brown A. W. and Wallnau K.C. *Engineering of Component-Based Systems, Component-Based Software Engineering.* Software Engineering Institute, ieee computer society press edition, 1996.

[32] Jaelson Castro Carina Alves, João Bosco Pinto Filho. *Analysing the Tradeoffs Among Requerements, Architectures and COTS Components.* Technical report, Centro de Informàtica, Universidale Federal de Pernambuco Recife, Pernambuco.

[33] Steel G. Gosling J., Joy B. *The Java Language Specification.* Addison-Wesley, 1996.

[34] Fox G. Li X. Wen Y. Zhang G., Carpenter B. *The HPcpmd Model and its Java Binding*, chapter 14. High Performance Cluster Computing, 1999.

[35] Philippsen M. Is java ready for computational science? In *European Parallel and Distribuited Systems Conference*, Vienna, July 1998. Disponibile all'indirizzo http://math.nist.gov/javanumerics/.

[36] Bruce Eckel. *Thinking in Java.* 2nd edition, 2000. Disponibile all'indirizzo www.BruceEckel.com.

[37] *Java Native Interface.* Disponibile all'indirizzo http://java.sun.com/products/jdk/1.2/docs/guide/jni/.

[38] *Java programming with JNI.* Disponibile all'indirizzo http://ibm.com/developerWorks.

[39] Kurzyniec D. Bubak M. and Luszczek Piotr. A versatile support for binding native code to java.

[40] Demaine E.D. Converting c pointer to java reference. Technical report, Standford University, Palo Alto, California. Disponibile all'indirizzo http://www.cs.ucsb.edu/conferences/java98/papers/pointers.ps.

[41] Jace homepage.
Disponibile all'indirizzo http://jace.reyelts.com/jace.

[42] Ulteriori informazioni su Jace.
Disponibili all'indirizzo http://sourceforge.net/projects/jace.

[43] S. Porcarelli F.Zanini S. Sabina. A. Bondavalli, E. De Giudici. A freshness detection mechanism for railway applications. In *10th International Symposium Pacific Rim Dependable Computing*, Mar. 2004.

[44] Roland McGrath Andrew Oram Sandra Loosemore, Richard M. Stallman and Ulrich Drepper. *The GNU C Library Reference Manual*, 0.10 edition.

[45] Neko v0.9
Disponibile all'indirizzo http://lsrwww.epfl.ch/neko/.