UNIVERSITÀ DEGLI STUDI DI FIRENZE - D.S.I.

Technical Report

Lorenzo Falai

Quantitative evaluation using Neko tool: NekoStat extensions

November 2004: Preliminary version in Italian

We are now working on English version of the report

(it will be ready before 24th January 2005)

Capitolo 1

NekoStat

Il framework Neko semplifica notevolmente le fasi di progettazione, testing e analisi di algoritmi distribuiti; in Neko è però assente un supporto per la *valutazione di quantità* ricavabili dall'esecuzione o dalla simulazione di un algoritmo distribuito. Nelle prossime pagine è presente un'analisi dei possibili metodi che possono essere utilizzati per estrarre informazioni quantitative da applicazioni realizzate nel framework, analisi che ha portato alla realizzazione del pacchetto NekoStat.

1.1 Analisi quantitativa con Neko: possibili modalità

La potenzialità di Neko è data sia dalla libreria di comunicazione, che permette il trasferimento di oggetti Java attraverso reti diverse utilizzando primitive comuni, che dall'architettura generale del framework, che permette sia simulazioni che esecuzioni reali della stessa implementazione di un algoritmo distribuito. Il framework è quindi un ambiente ideale per effettuare analisi quantitative di algoritmi distribuiti, nei due possibili contesti di esecuzione.

L'assenza di un supporto all'analisi quantitativa ha portato ad uno studio delle possibili modalità con cui quest'analisi poteva essere effettuata in Neko. Quest'assenza è ancora più evidente nell'analisi simulativa; con il supporto di adeguati strumenti è possibile in questo caso effettuare analisi *on-line*, in parallelo con la simulazione del sistema.

Le modalità utilizzabili per l'analisi quantitativa con Neko sono state schematizzate nei punti seguenti:

1. Realizzazione di *meccanismi specifici* per la singola applicazione

distribuita; i meccanismi saranno rivolti alla valutazione delle misure, alla raccolta dei dati e all'esportazione dei risultati. L'utilizzo di una tecnica di questo tipo ha come svantaggio principale la riscrittura di codice simile per differenti applicazioni; il vantaggio è la semplice gestione di misure locali ai singoli processi, che possono essere gestite in maniera autonoma dai singoli layer.

2. Analisi diretta dei file di *log* di Neko, mediante strumenti con i quali estrarre informazioni quantitative. Neko utilizza come meccanismo per la raccolta delle informazioni sulle esecuzioni dei singoli processi il framework fornito da java.util.logging; questo framework permette di descrivere, all'interno dell'applicazione, i punti in cui effettuare il log degli eventi eseguiti. Il log è organizzato in livelli di severità; nel file di configurazione dell'applicazione distribuita è possibile definire i punti di destinazione delle informazioni di log di ogni singolo livello (ad esempio, su file o su console).

   Per effettuare la valutazione quantitativa, possiamo quindi in una prima fase esportare su file gli eventi utili per la definizione delle misure; al termine della simulazione, o dell'esecuzione, è possibile utilizzare queste informazioni per creare una storia dell'esecuzione, e da questa ricavare le quantità di interesse.

   Il vantaggio di questo approccio è la semplificazione della fase di gestione degli eventi (è sufficiente richiamare le apposite funzioni di log), e la possibilità di valutare durate distribuite; queste possono essere ricavate dalla storia completa dell'esecuzione che possiamo ottenere con la fusione dei log. Il principale svantaggio di questo approccio è l'impossibilità di effettuare analisi on-line per le simulazioni; l'analisi quantitativa vieni infatti con questo metodo effettuata sempre al termine o della simulazione o dell'esecuzione reale, e di conseguenza l'analisi effettuabile è necessariamente off-line.

3. *Estensione del framework Neko*, con strumenti appositi per l'analisi quantitativa, creando appositi strumenti per la gestione e per la raccolta degli eventi. Nel caso di simulazioni possiamo quindi ideare meccanismi che permettano la valutazione quantitativa delle misure in parallelo con la simulazione dell'algoritmo. Per le esecuzioni reali possiamo invece raccogliere gli eventi in apposite strutture dati, dalle quali sia possibile

creare una storia dell'esecuzione. Questo approccio ha quindi i vantaggi di permettere analisi on-line per le simulazioni, e di fornire strumenti comuni per la valutazione quantitativa, che potranno essere utilizzati in contesti differenti.

Da queste considerazioni è nata l'idea di realizzare il pacchetto NekoStat, seguendo il terzo metodo. Questo pacchetto ha lo scopo di estendere il framework Neko in modo da uniformare e semplificare i passi necessari all'analisi di sistemi distribuiti realizzati in Neko.

Nelle prossime sezioni viene presentata l'architettura del pacchetto NekoStat, partendo da una descrizione dei ruoli dei singoli componenti, per spiegare poi le interazioni tra queste componenti nella realizzazione di analisi. Nei due capitoli successivi, invece, vengono presentati due casi di studio in cui è stato utilizzato per la realizzazione e l'analisi il pacchetto NekoStat.

L'idea originale di Neko è il testing di algoritmi distribuiti; l'architettura modulare di Neko permette la semplice realizzazione di meccanismi di fault injection, utili per valutare le prestazioni e il comportamento del sistema analizzato in presenza di guasti.

1.2 Architettura di NekoStat

Alla base di Neko vi è l'idea di permettere allo sviluppatore di realizzare un'unica implementazione dello stesso algoritmo, usabile poi sia per le simulazioni che per la prototipazione. Nella realizzazione di NekoStat si è voluto mantenere questa idea di fondo; per questo, il pacchetto è stato realizzato cercando di minimizzare l'impatto dell'uso di NekoStat nel codice delle applicazioni, e soprattutto cercando di rendere l'uso il più possibile comune ai due tipi di analisi.

Lo sviluppatore che intende utilizzare NekoStat per analizzare le prestazioni (o, comunque, per ottenere delle misure) dell'algoritmo sotto analisi, deve soltanto realizzare due modifiche al codice dell'applicazione Neko:

1. introdurre le chiamate al metodo log(Event) dello StatLogger, nei punti opportuni di codice dove si vuole segnalare il verificarsi di un evento; NekoStat si occuperà poi di gestire gli eventi così segnalati;

2. realizzare uno StatHandler che permetta la gestione degli eventi definiti nel codice, e che quindi permetta di ricavare da questi le opportune quantità.

Come risulta evidente da questa descrizione, l'interfaccia di funzionamento di NekoStat è comune sia alle simulazioni che alle esecuzioni reali dell'algoritmo: l'architettura interna, le interconnessioni tra le componenti, sono notevolmente diverse per questi due tipi di analisi, ma questo viene nascosto all'utente.

Per analizzare più nel dettaglio il funzionamento del pacchetto, nelle prossime pagine è presente un'analisi degli scopi e una breve spiegazione del funzionamento dei singoli componenti.

Una prima suddivisione può essere fatta tra due parti del pacchetto: quella che si occupa di fornire un supporto alle analisi, in simulazioni e in esecuzioni, e quella che fornisce gli strumenti matematici di supporto per la gestione delle quantità.

1.3 Strumenti di supporto all'analisi

1.3.1 Strumenti comuni alle simulazioni e alle esecuzioni distribuite

1.3.1.1 StatLogger

Lo StatLogger ha essenzialmente due ruoli:

- nelle simulazioni, ricevere gli eventi dei differenti processi, richiamare la funzione di gestione degli eventi dello StatHandler definito dall'utente e controllare se la simulazione può essere interrotta, nel caso di raggiungimento di 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.

Lo StatLogger è realizzato utilizzando il pattern Singleton, in modo che tutti i layer appartenenti allo stesso processo (alla stessa JVM) utilizzino lo stesso oggetto. In simulazione i processi sono simulati all'interno di un'unica JVM,

e quindi lo StatLogger è in realtà unico per il sistema; nel caso di esecuzione reale, invece, abbiamo uno StatLogger per ogni processo.

Dato che il funzionamento dello StatLogger è molto diverso per esecuzioni e simulazioni, questo è in realtà soltanto un'interfaccia, che viene poi implementata rispettivamente dalle classi StatLoggerSim (per le simulazioni), StatLoggerMaster (per il master nelle esecuzioni reali) e StatLoggerSlave (per gli slave nelle esecuzioni reali). Come spiegato sopra, l'utente utilizza soltanto il metodo log dello StatLogger, che verrà poi eseguito dall'oggetto opportuno.

1.3.1.2 Event

Event è il contenitore delle informazioni degli eventi che avvengono nel sistema distribuita 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 qualunque oggetto Java, che può essere usato per il trasporto di informazioni da un componente all'altro di NekoStat.

Da notare come il tempo associato all'evento può essere di due tipi: un *tempo simulato* o un *tempo reale*.

Il tempo reale viene misurato in Neko in millisecondi, a partire dal tempo 0 del processo in cui è avvenuto l'evento. Il tempo 0 del processo è il momento corrispondente all'attivazione del processo stesso. In caso di simulazione si usa il tempo del simulatore; tutti i processi hanno quindi la stessa visibilità del tempo, il cui scorrimento è determinato dal simulatore ad eventi discreti.

1.3.1.3 StatHandler

Ha il ruolo di gestire i singoli eventi definiti per il sistema in esame.

Le quantità di interesse di un'applicazione distribuita possono essere espresse a partire dagli eventi del sistema; lo StatHandler traduce quindi gli eventi in quantità.

Lo StatHandler dipende sia dal tipo di applicazione che dall'insieme di misure che vogliamo ricavare con l'analisi; deve essere quindi realizzato dall'utente di NekoStat. 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 anche App. 1).

L'utente ha inoltre la possibilità di definire il metodo shouldStop, che può essere utilizzato nel caso di analisi dell'algoritmo in simulazione. Questo metodo viene richiamato dallo StatLogger durante la simulazione, al verificarsi di ogni evento; l'utente può definire in questo metodo le condizioni richieste per interrompere la simulazione. Un modo consigliato di utilizzo è, come vedremo meglio nella prossima sezione, l'utilizzo delle condizioni di stop sulle quantità analizzate, ma è possibile pensare anche a condizioni di tipo diverso (ad esempio, il verificarsi di particolari sequenze di eventi, o una qualunque altra condizione ricavabile a partire dalla storia degli eventi del sistema).

Nel pacchetto 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 per la successiva visualizzazione, che fornisce quindi la storia dell'esecuzione o della simulazione effettuata.

1.3.1.4 StatInitializer

I processi Neko vengono creati attraverso la definizione di opportuni NekoProcessInitializer, definiti dall'utente, che istanziano i layer dei singoli processi. In NekoStat ho realizzato un'apposita classe, StatInitializer, che predispone i layer e gli altri oggetti necessari per l'analisi. In questo modo l'utente deve soltanto richiamare l'inizializzazione di NekoStat, e può così istanziare soltanto i layer dell'applicazione da analizzare. Questa classe ha quindi il ruolo di preparare l'architettura di NekoStat (come visibile nelle Fig. 1.3 e 1.4).

1.3.1.5 StatLayer

Lo StatLayer ha come scopo la gestione dei messaggi di controllo di NekoStat, in particolare per la fase di terminazione dell'analisi.

Nelle simulazioni, il layer si attiva soltanto alla ricezione del messaggio di richiesta della terminazione dell'analisi, un messaggio del tipo NS_STOP; alla ricezione di questo messaggio, si attiva la procedura di esportazione dei risultati dello StatHandler, e può quindi terminare la simulazione.

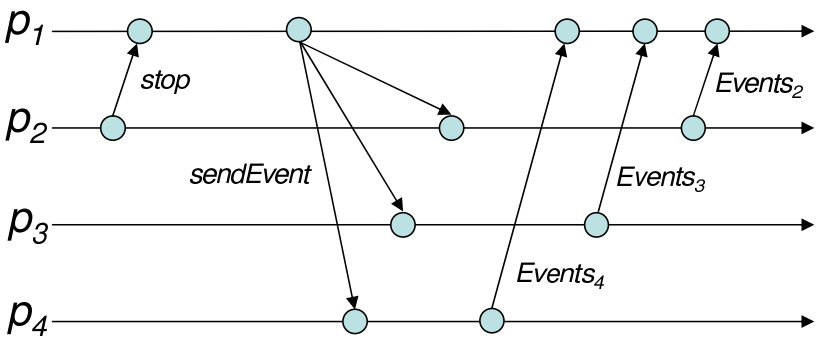


Figura 1.1: Esempio di stop dell'analisi in un'esecuzione reale

Nelle esecuzioni reali, invece, alla ricezione del messaggio NS_STOP da parte dello StatLayer del master, viene attivata una diversa procedura di terminazione, che richiede la collaborazione dei diversi processi.

In una prima fase questo layer invia un messaggio di richiesta NS_SENDEVENTS a tutti i processi, sui quali è attivo uno StatLayer che procederà all'invio degli eventi al master (vedi Fig. 1.1). Per non esaurire la memoria del master, gli slave non inviano l'elenco completo degli eventi avvenuti nel sistema; l'invio è gestito attraverso un apposito layer ListFragmenter, il cui funzionamento è descritto nelle prossime pagine.

1.3.2 Strumenti specifici per l'analisi di prototipi

1.3.2.1 EventCollector

Si occupa di raccogliere gli eventi locali ad ogni processo durante l'esecuzione reale 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. Lo StatLogger del master riceve gli EventCollector dai quali crea la *storia* dell'esecuzione distribuita del sistema; da notare che la comunicazione tra le diverse componenti è realizzata in modo che il *taglio* ottenuto dal master attraverso la fusione dei vari EventCollector sia sempre consistente (vedi Sez. **??**). L'EventCollector mantiene la lista degli eventi avvenuti sul nodo utilizzando un'apposita struttura dati, di nome SparseArrayList; questa struttura dati è simile ad un Vector (vettore di oggetti, predefinito in Java)

ma utilizza una diversa politica di ridimensionamento. L'inserimento in un Vector pieno (la cui memoria allocata è piena) implica un ridimensionamento della struttura al doppio della dimensione originale, per preparare lo spazio per futuri inserimenti. L'inserimento in un SparseArrayList pieno, invece, implica la creazione dello spazio soltanto per il nuovo elemento. In un primo tempo avevo utilizzato un Vector per la memorizzazione degli eventi, ma l'inserimento di un nuovo evento richiedeva un tempo non prevedibile, estremamente grande nel caso di inserimento in una struttura con molti elementi già piena (da esperimenti effettuati, si è ottenuto anche 0.5 sec); la struttura SparseArrayList, invece, richiede un tempo sempre uguale (sotto i 10 msec), e soprattutto indipendente dal numero di oggetti contenuti. La scelta di questa struttura è quindi stata motivata dal tentativo di minimizzare l'overhead del processo di monitoring, in modo da influenzare il meno possibile i risultati ottenibili.

1.3.2.2 ListFragmenter

Permette l'invio al master dell'EventCollector locale, al termine dell'esecuzione, in maniera bufferizzata. Si utilizza questo layer per non incorrere in situazioni nelle quali, a causa dell'ampio numero di eventi ricevuti, la memoria disponibile nel nodo che esegue il processo master si esaurisca. Durante l'esecuzione della fase finale di gestione degli eventi, questo layer viene utilizzato per inviare al master soltanto gli eventi di cui necessita in quel momento per l'analisi; l'invio avviene in blocchi, in modo da garantire comunque buone prestazioni.

1.3.2.3 ClockSynchronizer

Come vedremo più dettagliatamente, per l'analisi quantitativa di algoritmi e sistemi distribuiti è spesso necessario disporre di clock sincronizzati. Per questo, NekoStat 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 all'avvio del processo viene settato a 0. Per la

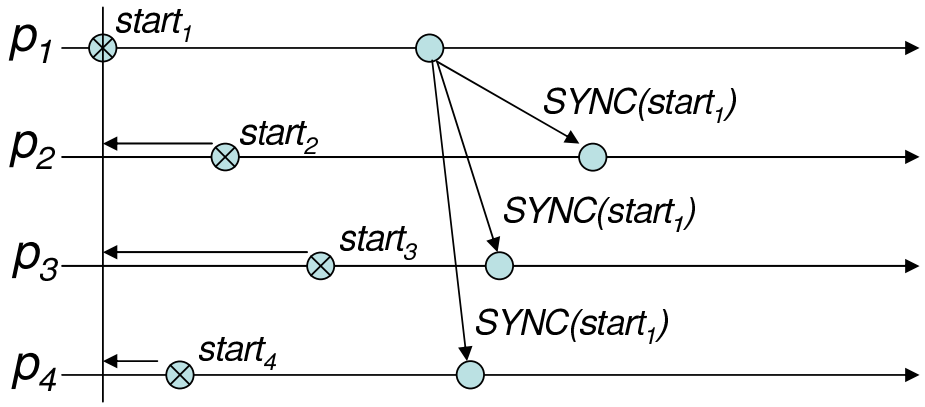


Figura 1.2: Sincronizzazione dell'origine dei tempi

sincronizzazione esterna dei clock è consigliato l'uso del protocollo NTP, per i motivi che vedremo più dettagliatamente in seguito.

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*: il clock del master è utilizzato come clock di riferimento (vedi Fig. 1.2).

È possibile in ogni caso utilizzare metodi di sincronizzazione dei clock differenti, basati ad esempio sull'implementazione di appositi meccanismi di sincronizzazione attraverso layer Neko, integrabili nell'architettura di NekoStat.

1.4 Strumenti di supporto all'analisi statistica

Gli strumenti di supporto all'analisi statistica vengono forniti attraverso apposite classi per la gestione di quantità. Queste classi sono estensioni di alcune classi disponibili nella libreria matematica Colt per Java [6]. La libreria Colt è stata sviluppata al Cern di Ginevra con lo scopo di fornire una libreria matematica con funzionalità più avanzate della libreria predefinita java.Math, e che fornisse prestazioni confrontabili a quelle raggiungibili con programmi scritti in linguaggi di programmazione come Fortran, C o C++.

Nella realizzazione di un'applicazione NekoStat non c'è nessun obbligo di utilizzo delle classi descritte in questa sezione: le quantità sono gestibili e

rappresentabili nel modo preferito all'interno dello StatHandler specifico per l'applicazione. Le classi seguenti possono essere però utili per effettuare molti tipi di analisi.

Le classi di supporto all'analisi sono formate da un *contenitore* di valori, dai quali la classe ricava i principali parametri statistici; queste classi hanno un'interfaccia comune, composta da:

- il metodo add(double) permette l'inserimento nel contenitore del valore fornito come parametro;
- i metodi per ottenere informazioni statistiche sulla quantità, come size(), mean(), getMedian(), standardDeviation(), min() e max(). La lista completa delle informazioni ricavabili da una quantità può essere trovata nella documentazione della libreria Colt ([5]).

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

1.4.1 Quantità

1.4.1.1 QuantityOnlyStat

Questo tipo di quantità permette soltanto la valutazione dei principali parametri statistici della quantità analizzata. In questo caso, abbiamo a disposizione la media stimata, la varianza, la deviazione standard, la mediana, ma non abbiamo ad esempio la possibilità di effettuare eliminazione degli outlier o di effettuare analisi di covarianza con altre quantità.

I metodi usabili su questa classe sono quelli della classe hep.aida.StaticBin1D ([5]), ed altri metodi che permettono l'eliminazione del transiente iniziale, la gestione delle condizioni di stop (vedi più avanti) e quelli per l'esportazione dei parametri statistici della quantità su file.

1.4.1.2 QuantityDataOnFile

Estensione della classe precedente, è un QuantityOnlyStat in cui le misure inserite nel contenitore vengono esportate su file. Le possibilità di analisi statistica sono le stesse della classe precedente.

1.4.1.3 Quantity

Questa classe è quella che permette le capacità massime di analisi; oltre ai parametri statistici sopra descritti, permette di ricavare altre utili informazioni, come la covarianza rispetto ad un'altra Quantity. È possibile inoltre effettuare la rimozione del transiente, sia iniziale che finale, e la rimozione degli outlier. 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. La rimozione degli outlier può risultare molto utile nelle analisi effettuabili con NekoStat: il meccanismo di garbage collection delle applicazioni Java spesso è la causa nell'ottenere valori molto distanti dalla media; in questi casi è possibile utilizzare la rimozione degli outlier per eliminare questi valori. Se è d'interesse non tanto il valore medio di una cerca quantità, ma ad esempio il minimo o il massimo, può non essere significativo effettuare la rimozione degli outlier. Per eliminare l'influenza della garbage collection sui risultati ottenuti, può essere utile in questo caso *preimpostare* una quantità di memoria sufficiente per l'intera durata dell'applicazione, e utilizzare meccanismi incrementali del garbage collector, disponibili come opzioni della JVM (si veda [7]).

Lo svantaggio nell'uso di questa classe deriva dal fatto che i valori inseriti nel contenitore associato alla Quantity permangono in memoria: l'occupazione di memoria può quindi crescere molto velocemente. La classe Quantity è un'estensione della classe hep.aida.DynamicBin1d (si veda [5]).

1.4.2 Condizioni di stop

Alle 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; come dicevamo sopra, questa condizione può essere utilizzata nelle simulazioni, per decidere le situazioni in cui la simulazione può terminare.

Le possibili condizioni di stop sono le seguenti:

Nessuna condizione di stop: è l'impostazione predefinita, la quantità non raggiunge mai la propria condizione di stop; se tutte le quantità sono di questo tipo, è necessario impostare direttamente nel codice dell'applicazione le situazioni in cui interrompere l'esecuzione, che altrimenti potrebbe proseguire ininterrottamente.

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

Stop su intervallo di confidenza: si considera raggiunta la confidenza sui dati quando la dimensione dell'intervallo di confidenza sulla media, di livello $(1-\alpha)$, è inferiore ad una percentuale β della media della quantità. In particolare, sia $\widehat{x}$ la media stimata, μ la media della quantità che stiamo analizzando, $S^2(n)$ la varianza stimata della quantità. Abbiamo quindi
$\widehat{x} = \frac{\sum_{i=1}^{n} x_i}{n}$
$S^2(n) = \frac{\sum_{i=1}^{n} \left(x_i - \widehat{x}\right)^2}{n-1}$
Il calcolo della dimensione dell'intervallo di confidenza della media è basato sul fatto che supponiamo il campione $\{x_1, ..., x_n\}$ della quantità, composto dalle singole misure raccolte, distribuito normalmente. Dopo n misure, la dimensione dell'intervallo di confidenza di livello $(1-\alpha)$ della media può essere calcolato come $t_{n-1,1-\frac{\alpha}{2}}\sqrt{\frac{S^2}{n}}$, dove $t_{n-1,x}$ denota il quantile della *t-Student* con $n-1$ gradi di libertà (si veda [1]).
A causa del fatto che stiamo *stimando* la media reale μ attraverso una sua stima $\widehat{x}$, per valutare se l'intervallo di confidenza ottenuto è sufficiente dobbiamo utilizzare al posto della percentuale β, una percentuale corretta $\frac{\beta}{(1-\beta)}$ della media stimata (si veda [8], per i dettagli che motivano l'utilizzo di questo valore corretto). Dopo aver ottenuto una nuova misura andremo quindi a valutare la formula: $t_{n-1,1-\frac{\alpha}{2}}\sqrt{\frac{S^2}{n}} \leq \widehat{x}\frac{\beta}{(1-\beta)}$.
Per $n \geq 30$, invece di utilizzare il quantile della distribuzione *t-Student*, si usano i quantili della distribuzione normale; per questi valori di *n*, infatti, si ha che $t_{n-1,1-\frac{\alpha}{2}} \simeq \phi_{1-\frac{\alpha}{2}}$ (dove con ϕ_x si intende il quantile della legge normale) .

L'architettura ad oggetti di NekoStat rende semplice realizzare possibili nuove funzionalità per le singole classi: in questo caso, sarà possibile ad esempio estendere le classi Quantity impostando nuove condizioni di stop.

1.5 Uso di NekoStat

Nelle prossime sezioni viene descritto il funzionamento dell'analisi nei casi di simulazione e esecuzione reale di un prototipo. Nelle Fig. 1.3 e 1.4 sono disponibili le visioni dettagliate delle componenti di una tipica sessione di analisi con NekoStat.

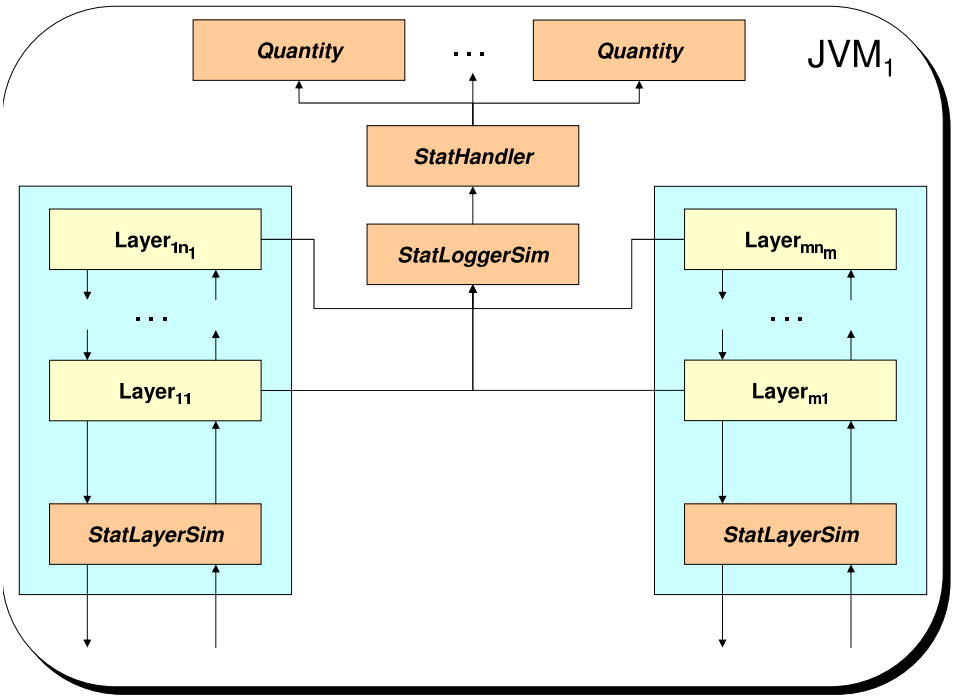


Figura 1.3: Architettura di una tipica applicazione NekoStat - simulazione

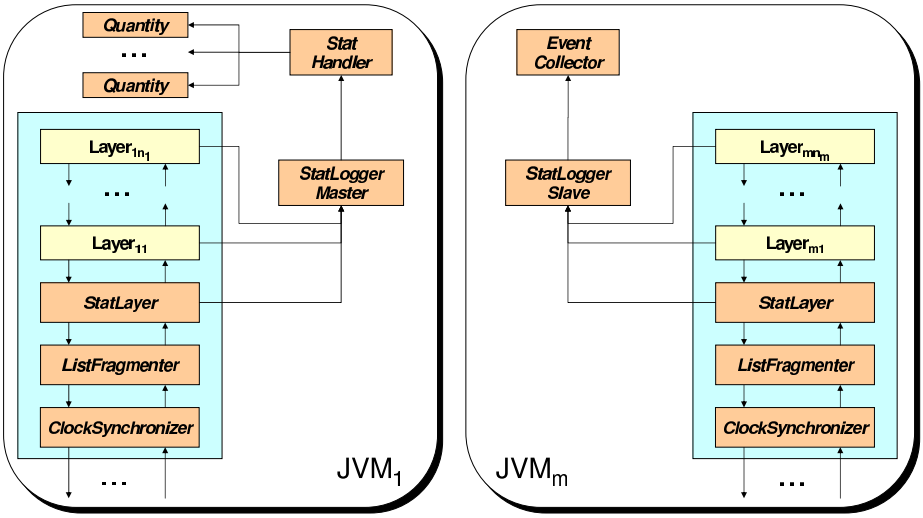


Figura 1.4: Architettura di una tipica applicazione NekoStat - esecuzione reale

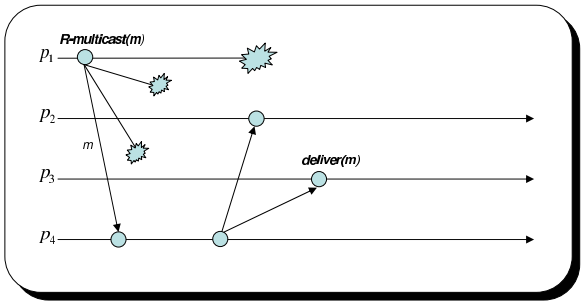


Figura 1.5: Diagramma spazio-tempo di un run di un algoritmo di multicast affidabile

Come supporto alla spiegazione successiva, in Fig. 1.5 è visibile un diagramma spazio-tempo corrispondente ad una esecuzione di un algoritmo per il multicast affidabile, e nelle Fig. 1.6 e 1.7 sono visibili le evoluzioni temporali dell'analisi della quantità di tempo tra il multicast del messaggio *m* e il deliver da parte dell'ultimo processo. La Fig. 1.6 si riferisce al caso di analisi durante una simulazione, mentre la Fig. 1.7 rappresenta una possibile analisi durante una esecuzione reale. Da notare come nel caso di simulazione l'analisi avvenga in parallelo al verificarsi degli eventi nel simulatore, mentre nell'esecuzione reale l'analisi è posteriore al termine dell'esecuzione.

1.5.1 Simulazione

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, il quale richiama il metodo handle dello StatHandler; quest'ultimo ottiene le misure a partire dall'evento segnalato (utilizzando se necessario il valore del clock simulato associato all'evento); se si utilizzano le classi sopra definite per la gestione delle misure, vengono inserite negli opportuni contenitori le nuove misure. A questo punto, lo StatLogger controlla se abbiamo raggiunto il grado di confidenza minimo sulle misure, richiamando in questo caso l'interruzione della simulazione.

Per le simulazioni viene quindi effettuata un'analisi statistica on-line delle quantità di interesse.

L'interruzione della simulazione con esportazione dei valori raccolti

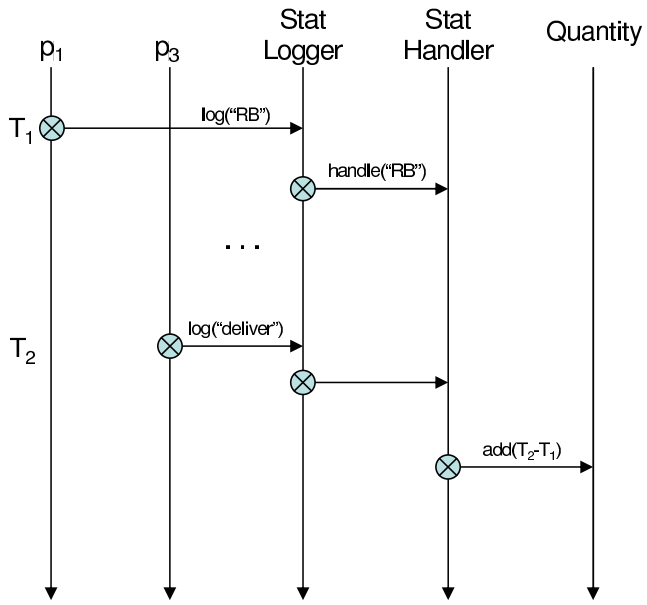


Figura 1.6: Evoluzione nell'analisi dell'algoritmo multicast - simulazione

avviene anche nel caso in cui un layer richieda esplicitamente il termine della simulazione, inviando un messaggio NS_STOP al master.

1.5.2 Esecuzione distribuita reale

L'analisi di un'esecuzione di un sistema distribuito realizzata con NekoStat richiede una prima fase di preparazione degli oggetti necessari; questa fase è leggermente differente per il master e per gli slave.

L'avvio dell'analisi sul master richiede la creazione degli appositi livelli ClockSynchronizer, ListFragmenter e StatLayer, e degli oggetti StatLoggerMaster e StatHandler. Se l'utente utilizza la sincronizzazione dei clock predefinita, viene eseguita una prima fase di sincronizzazione, che porta i vari processi ad ottenere la stessa origine dei tempi del master.

L'avvio dell'analisi sugli slave richiede la creazione dei livelli ClockSynchronizerSlave, ListFragmenter, StatLayer, e dello StatLoggerSlave.

Al termine di questa prima fase, può avere inizio l'esecuzione dei processi Neko.

Durante l'analisi, come è visibile nell'esempio in Fig. 1.7, lo StatLogger locale a ogni processo raccoglie nell'EventCollector gli eventi che via via si verificano. Un qualunque processo può richiedere l'interruzione dell'analisi,

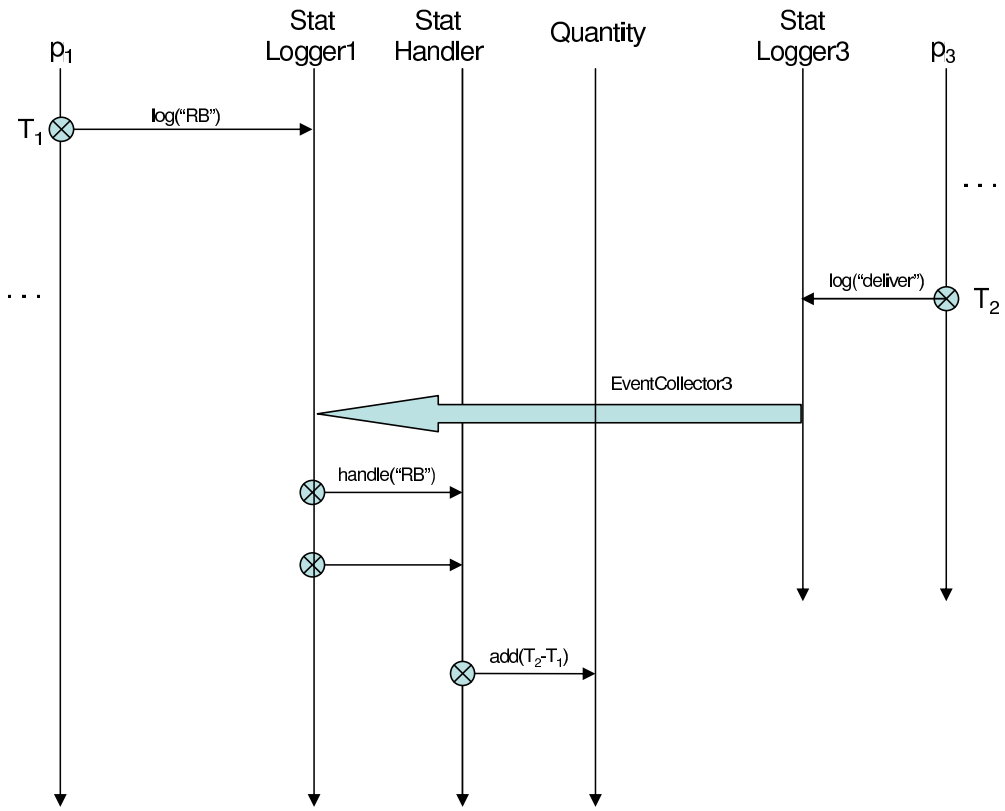


Figura 1.7: Evoluzione nell'analisi dell'algoritmo multicast - prototipazione

tramite l'invio al master di un apposito messaggio NS_STOP. Alla ricezione di questo messaggio, il livello StatLayer del master fa partire la fase di analisi; in questa fase, il master richiede ai singoli partecipanti gli eventi raccolti, ed esegue, rispettando l'ordinamento temporale tra gli eventi, il metodo handle dello StatHandler. Al termine vengono esportate le informazioni statistiche di interesse sulle quantità, e si avverte il processo che aveva richiesto la terminazione dell'analisi.

Da notare come, affinché il protocollo sopra descritto funzioni correttamente, è necessario l'utilizzo di una connessione affidabile che permetta lo scambio dei messaggi tra i processi che compongono l'applicazione distribuita. Nel caso di utilizzo di connessioni inaffidabili (come UDP), l'utente ha l'obbligo di istanziare un'apposita rete di controllo per lo scambio dei messaggi di NekoStat, e specificare questa rete nel file di configurazione dell'applicazione (vedi App. 1).

L'analisi effettuabile sull'esecuzione distribuita è *off-line*: non vengono utilizzate condizioni di stop dell'analisi, che deve quindi essere interrotta esplicitamente da un processo appartenente al sistema analizzato.

1.6 Sincronizzazione dei clock

Nell'analisi di quantità di un sistema distribuito è fondamentale l'utilizzo di clock sincronizzati; senza la sincronizzazione non abbiamo la possibilità di misurare tutte quelle quantità *globali*, che dipendono cioè dagli eventi verificati in diversi processi che fanno parte del nostro sistema. Alcune quantità non possono essere infatti misurate con precisione senza l'uso di un riferimento temporale comune ai diversi processi: esempi sono il tempo di trasmissione di un messaggio, o la differenza fra i tempi in cui si verificano eventi locali a processi diversi del sistema. Un *clock locale* ad un singolo processo, infatti, permette soltanto di misurare durate locali e durate round-trip[1]. Per misurare invece durate distribuite è necessario avere a disposizione nei vari processi che compongono il sistema di un *clock globale* (vedi Sez. **??**).

In un sistema distribuito il valore dei clock visibile ai singoli processi può differire sia per *fase* (la differenza tra i tempi ottenuti da diversi clock in corrispondenza dello stesso tempo reale) che per *velocità* (il drift rate).

[1]Con durata round-trip si intende una durata a catena chiusa, corrisponde cioè ad una catena di eventi che ha inizio e fine nello stesso processo, ma che può attraversare diversi processi prima della terminazione.

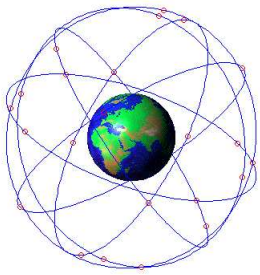
In alcune applicazioni, ad esempio in applicazioni real-time, può essere necessario risolvere due problemi:

1. la sincronizzazione dei clock degli host appartenenti ad un sistema distribuito;
2. la sincronizzazione con il *tempo reale*.

Per la realizzazione di NekoStat sono stati analizzati i possibili meccanismi disponibili per la sincronizzazione dei clock. Visto l'uso di Neko e NekoStat, strumenti rivolti all'analisi di algoritmi distribuiti, la cui naturale applicazione è la misurazione di prototipi di applicazioni, si è cercato di trovare un meccanismo che potesse essere utilizzato sia in reti locali che su sistemi distribuiti su scala geografica (principalmente, connessioni WAN su Internet). Le scelte possibili erano l'uso di meccanismi proprietari di sincronizzazione (algoritmi specifici da implementare in Neko) o l'utilizzo di protocolli esistenti, esterni all'ambiente. La soluzione scelta è mista: lo sviluppatore ha la possibilità di scegliere il proprio algoritmo di sincronizzazione, da integrare come layer base dell'architettura di NekoStat, oppure utilizzare l'algoritmo di sincronizzazione di tipo *master-slave* visto per la sincronizzazione dell'origine dei tempi (realizzato dal layer predefinito ClockSynchronizer). Oltre al layer di sincronizzazione, si prevede l'uso di meccanismi esterni a Neko.

Per la sincronizzazione con il tempo reale dei clock locali si è deciso di usare il protocollo *NTP* disponibile su Internet. La motivazione di questa scelta è dipesa dall'ampia disponibilità di questo protocollo e dalla possibilità di ottenere un'accuratezza molto buona. NTP è stato utilizzato con successo sui computer che hanno ospitato i casi di studio; in questo caso si è avuto la necessità di sincronizzare gli orologi di due PC Linux, che si trovavano uno in Italia (a Firenze), l'altro in Giappone (presso il *JAIST*). La sincronizzazione è avvenuta utilizzando server NTP geograficamente vicini ai singoli PC: per il PC in Italia ho utilizzato i server pubblici ntp.ein.it e ntp2.ein.it, per il PC in Giappone il server clock.nc.fukuoka-u.ac.jp. Questo ha permesso di ottenere un'accuratezza della sincronizzazione difficilmente raggiungibile tramite l'uso di un unico algoritmo integrato in Neko. Rispetto alla classificazione delle due tipologie di sincronizzazioni possibili nei sistemi distribuiti fornita sopra, NTP è un protocollo specializzato per risolvere il problema della sincronizzazione con il tempo reale: anche se questo può sembrare più complesso del primo, su sistemi distribuiti geograficamente è spesso di difficoltà equivalente.

Nel caso in cui sia necessario, per il problema in esame, sincronizzare i clock con il tempo reale con una precisione maggiore di quella ottenibile con NTP utilizzando server in rete, sarà possibile sincronizzare gli host del sistema basandosi su soluzioni hardware, principalmente di due tipi: *ricevitori GPS* o *ricevitori a radiofrequenza*.



I ricevitori GPS (*Global Positioning System*) sincronizzano il clock basandosi sui messaggi che alcuni satelliti continuamente inviano sulla Terra. Il sistema *GPS* è composto da una serie di 24 satelliti (più 8 di riserva), che contengono al loro interno un orologio atomico, con precisione dell'ordine di pochi nanosecondi. Con l'uso di questi ricevitori è possibile garantire una fase tra il clock locale e il clock del satellite dell'ordine dei 100 nanosecondi - 1 microsecondo.

I ricevitori a radiofrequenza si basano invece sui segnali inviati via radio da vari organismi, come il *NIST* (USA) e il *DCF77* (Germania). La soluzione radio permette una fase con il tempo reale dell'ordine del millisecondo.

Se non si utilizzano software o hardware specializzati, sia per la soluzione GPS che per radio, la precisione con cui è possibile sincronizzare il clock del singolo host può essere molto diversa dai valori sopra descritti, a causa dell'inevitabile rumore: basta considerare, per esempio, il tempo di latenza tra la ricezione del messaggio contenente il clock del satellite, e l'istante di tempo in cui il clock del sistema viene aggiornato da parte del sistema operativo.

Su LAN si può realizzare la sincronizzazione utilizzando un server NTP locale, che può essere usato come tempo di riferimento per gli altri nodi della rete; in questo caso, visti i tempi solitamente bassi di latenza, la sincronizzazione dei vari clock può essere molto buona (ordine di grandezza 10^{-5} sec), anche se la sincronizzazione con il tempo reale può non esserlo (dipende dall'accuratezza del clock del server NTP locale).

1.6.1 Il protocollo NTP

Un sistema molto utilizzato per la sincronizzazione di host in rete è il protocollo NTP (Network Time Prococol - [9], [12]).

NTP è un sistema per la sincronizzazione espressamente studiato per funzionare sulla rete Internet. Le caratteristiche principali di questo protocollo sono:

1. la sincronizzazione continuativa del clock, realizzata attraverso un demone sempre in esecuzione sui client;
2. l'utilizzo possibile sia per la sincronizzazione di singoli host che di intere sottoreti;
3. l'ampia disponibilità di client e server per gran parte delle architetture e dei sistemi operativi esistenti;
4. la tolleranza ai guasti, realizzata attraverso l'utilizzo da parte dei client di più server, sui cui risultati vengono effettuati dei controlli per valutare la correttezza dei valori inviati e eventuali problemi temporanei di connessione con gli stessi (gestito attraverso un'architettura piramidale - [12]);
5. il tipo di sincronizzazione, che, come già detto, è una sincronizzazione con il tempo reale.

Il protocollo per la comunicazione di informazioni temporali è basato su di una serie di algoritmi per la sincronizzazione con il segnale di input temporale, che può essere prelevato sia dalla rete (tramite richieste a uno o più server NTP), che da una sorgente direttamente connessa al sistema (ricevitori GPS, ricevitori radio). L'infrastruttura su cui si basa NTP è composta da un insieme di server primari, che detengono i tempi di riferimento (con fase rispetto al tempo reale dell'ordine dei nanosecondi), e un insieme di server secondari (con fase dell'ordine dei millisecondi).

1.6.2 La sincronizzazione continua di NTP e i salti temporali

NTP impiega diversi minuti per raggiungere il massimo grado di accuratezza; l'accuratezza massima si ottiene quando i messaggi scambiati con il server che detiene il tempo di riferimento sono in numero sufficiente per riuscire a

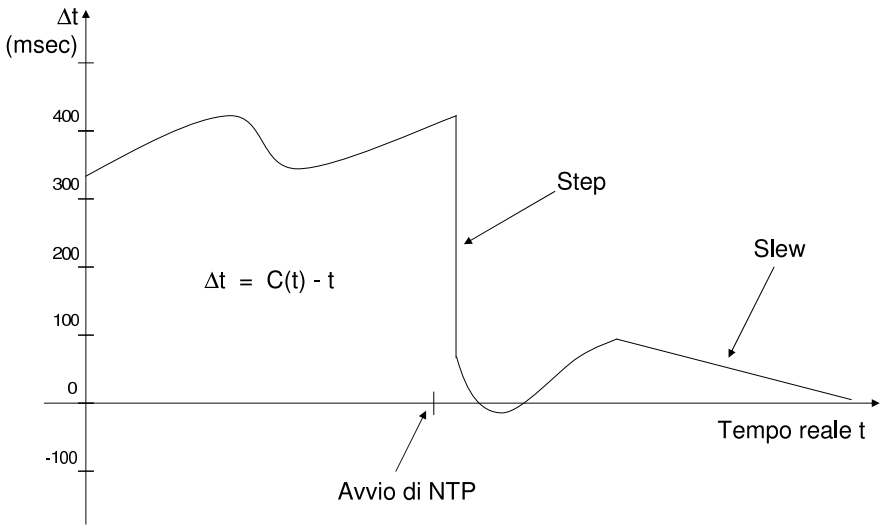


Figura 1.8: Step e slew nel protocollo NTP

stimare con precisione sufficiente la latenza media. Per permettere ai clock di raggiungere velocemente una buona accuratezza, impedendo però salti temporali troppo frequenti, in NTP si usa un metodo di sincronizzazione in cui i *grandi* aggiustamenti occorrono *velocemente* (*stepping*), mentre per quelli più *piccoli* si usa un metodo *graduale* (*slewing*, vedi Fig. 1.8).

Nelle analisi effettuabili con NekoStat, nel caso di utilizzo di NTP per la sincronizzazione, è necessario porre attenzione a trovarsi soltanto nel caso di sincronizzazioni con slewing: salti temporali troppo evidenti possono infatti falsificare le misure raccolte. Nelle analisi effettuate nel caso di studio del Cap. 5, ad esempio, è stata richiesta manualmente una sincronizzazione di NTP qualche minuto prima dell'inizio dell'esperimento.

1.7 Neko e NekoStat

Il pacchetto NekoStat entrerà a far parte della distribuzione del software Neko dalla prossima versione del framework. Informazioni maggiori sul tool Neko

sono disponibili sulla pagine web [10]; da questa pagina è anche possibile scaricare il tool. La patch NekoStat è invece disponibile all'indirizzo [11]. Nell'App. 1 è presente una breve all'installazione di NekoStat su Neko, e un tutorial esemplificativo.

Appendice A

Manuale d'uso di NekoStat

Nelle prossime pagine viene descritta l'installazione di NekoStat; l'utilizzo del tool viene poi descritto attraverso un breve tutorial che presenta gli strumenti di NekoStat. Nella versione di NekoStat disponibile in [11], che sarà integrata in Neko all'uscita della prossima versione del tool, è presente un pacchetto Java lse.neko.examples.stat contenente un esempio di utilizzo di NekoStat. L'esempio è una versione semplificata del caso di studio descritto nel Cap. 5; in particolare, è un'applicazione per la valutazione della qualità di servizio del failure detector basato su strategia push rivisto da CTA, descritto in Sez. **??**.

A.1 Installazione di NekoStat

Per installare NekoStat è sufficiente seguire i passi seguenti:

1. Effettuare il download del file neko0.8.tar.gz, il tool Neko 0.8 scaricabile da [10]; con la decompressione del file (ad esempio utilizzando gunzip) verrà creata la directory neko;

2. Effettuare il download del file nekostat0.8.tar.gz, scaricabile da [11]; porsi nella directory dove è stato decompresso Neko al passo precedente (directory padre dove si trova neko come sottodirectory) e decomprimere il file. Con la decompressione verranno creati i file di NekoStat e verranno sovrascritti alcuni file dell'installazione base di Neko.

3. A questo punto è possibile effettuare un'installazione standard di Neko, seguendo le istruzioni presenti nel file neko/README.

L'installazione di Neko e NekoStat richiede il JAVA SDK con versione superiore alla 1.3; essendo realizzato in Java, il tool è utilizzabile su differenti sistemi operativi: Linux, MacOS, Windows. Alcuni programmi di supporto e alcuni script sono per ora disponibili soltanto su sistemi Unix.

A.2 Tutorial sul pacchetto lse.neko.examples.stat

Con i passi descritti è stato installato Neko con le estensione e le librerie di supporto NekoStat. Nel pacchetto lse.neko.examples.stat è presente il codice di esempio di una possibile applicazione in cui è possibile usare NekoStat per effettuare analisi quantitative.

Il contenuto della directory neko/lse/neko/examples/stat è il seguente:

FailureDetectorCTA.java ActiveLayer che implementa il failure detector push rivisto da CTA (vedi Sez. **??**);

FailureDetectorCTAInitializer.java classe che permette di creare il processo contenente come unico layer applicativo il FailureDetectorCTA, e contenente le chiamate al pacchetto NekoStat per l'inizializzazione dei layer e degli altri strumenti necessari all'analisi;

FailureDetectorCTAStatHandler.java lo StatHandler che permette di valutare le metriche T_M, T_{MR} e il ritardo trasmissivo oneway dei messaggi di heartbeat;

HeartbeatCTA.java ActiveLayer che implementa il meccanismo base di heartbeat;

HeartbeatCTAInitializer.java classe che contiene le chiamate al pacchetto NekoStat per l'inizializzazione dei layer e degli altri strumenti necessari per l'analisi, oltre alla predisposizione del processo Neko e del layer applicativo HeartbeatCTA

network.config, simulation.config file di configurazione dell'applicazione, rispettivamente per esecuzione su rete reale e per simulazioni.

Le classi FailureDetectorCTAInitializer e HeartbeatCTAInitializer sono quelle che permettono la creazione dei NekoProcess; per istanziare il processo, in modo da poter effettuare analisi quantitative, è sufficiente creare un

nuovo StatInitializer e richiamare il metodo init(process, config, statHandler) all'inizio del metodo init() di queste classi, come riportato in Fig. A.1. Lo StatInizializer si occuperà di istanziare tutto quello che serve per l'analisi, rispetto all'ambiente di esecuzione in cui questa viene effettuata (simulazioni o esperimenti).

La classe FailureDetectorCTAStatHandler è lo StatHandler per le analisi quantitative specifico per l'applicazione. In Fig. A.2 è riportata la parte del codice dello StatHandler che permette la valutazione della metrica T_M. Il metodo init() viene richiamato all'avvio dell'analisi, e quindi all'avvio della simulazione o al termine dell'esecuzione su rete reale. In questo metodo si possono istanziare le quantità che rappresentano le metriche da analizzare, impostando le eventuali condizioni di stop (in questo caso, viene impostata la condizione di stop sulla metrica T_M rispetto all'intervallo di confidenza sulla media).

Il metodo handle(Event) viene richiamato per la gestione degli eventi. In questo caso, per valutare la metrica T_M, è sufficiente calcolare l'intervallo di tempo tra gli eventi $startSuspect$ e $endSuspect$, dato che non consideriamo situazioni in cui il processo monitorato è effettivamente crashed. Il metodo shouldStop viene richiamato ogni volta che viene gestito un nuovo evento durante una simulazione: in questo metodo possiamo definire ulteriori condizioni di stop per la simulazione, eventualmente utilizzando le condizioni di stop definibili sulle quantità (come in questo caso).

Al termine dell'analisi viene richiamato il metodo writeResults(), nel quale si definiscono le informazioni da esportare: in questo caso, vengono esportati, attraverso il metodo export() della QuantityDataOnFile, i parametri statistici principali della metrica (media, deviazione standard, massimo, minimo,

```
public class FailureDetectorCTAInitializer
        implements NekoProcessInitializer
{

public void init(NekoProcess process, Configurations config) {
        new StatInitializer().init(process, config,
                        new FailureDetectorCTAStatHandler());
        // qui va il codice per istanziare i layer del processo
}
}
```

Figura A.1: Esempio di NekoProcessInitializer e codice per l'inizializzazione di NekoStat

```
private QuantityDataOnFile tm;

public void init() {
        tm = new QuantityDataOnFile("tm");
        tm.setStopConditionInterval(0.95, 0.05);
}

public void handle(Event e) {
        if (e.getDescription().equals("startSuspect")) {
                // inizio di un sospetto erroneo
                tmChrono = new Chronometer(e.getTime());
                return;
        }
        if (e.getDescription().equals("stopSuspect")) {
                // se attualmente 'suspect' –> 'trust'
                if (tmChrono != null) {
                        tmChrono.setStop(e.getTime());
                        tm.add(tmChrono.getValue());
                }
        return;
        }
}

public boolean shouldStop() {
        // metodo utile per le simulazioni
        // permette di definire le condizioni di stop della simulazione
        return (tm.isStopReached());
}

public void writeResults() {
        // esportazione su file dei risultati
        tm.export();
}
```

Figura A.2: FailureDetectorStatHandler

```
# Le reti usate:
# UDP per heartbeat, TCP per messaggi NekoStat
network = lse.neko.networks.comm.UDPNetwork,
lse.neko.networks.comm.TCPNetwork
stat.network.index = 1
```

Figura A.3: Parte del file di configurazione dell'applicazione distribuita per istanziare la rete di controllo NekoStat

intervallo di confidenza).

Per quest'applicazione è stata scelta la rete UDPNetwork negli esperimenti (file di configurazione network.config). Poiché la rete UDPNetwork è inaffidabile, è necessario istanziare una rete secondaria di tipo TCPNetwork, che verrà utilizzata per gli scambi di pacchetti necessari per l'analisi: per fare questo è sufficiente descrivere le reti da usare come in Fig. A.3.

Gli eventi vengono attivati nel codice dei singoli layer, attraverso il metodo log() dello StatLogger; questo si occuperà di raccogliere gli eventi, nel caso di esecuzione reale, oppure di passare il nuovo evento allo StatHandler per l'immediata gestione.

Si rimanda al codice, ampiamente commentato, per maggiori informazioni sul funzionamento di NekoStat; il codice principale del pacchetto è contenuto nella directory neko/lse/neko/stat.

```
public class FailureDetectorCTA
        extends ActiveLayer {

public void deliver(NekoMessage m) {
[...]
        if (timestamp > lastTimestamp) {
                lastTimestamp = timestamp;
                if (timestamp >= (pass - 1)) {
                        if (suspected) {
                        statLogger.log("stopSuspect", null);
                        suspected = false;
                }
        }
[...]
}

public void run() {
[...]
while (true) {
        deadline = initialClock + pass * eta + delta;
        sleep(deadline - clock());
        if (lastTimestamp < pass) {
                if (!suspected) {
                        suspected = true;
                        statLogger.log("startSuspect", null);
                }
        }
pass++;
[...]
}
```

Figura A.4: Porzione del codice del failure detector per la valutazione della metrica T_M

Bibliografia

[1] P. Baldi. Calcolo delle Probabilità e Statistica. *McGraw-Hill*, 1998.

[2] A. Casimiro, P. Martins, P. Verissimo, L. Rodrigues. Measuring Distributed Durations with Stable Errors. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, Dec. 2001.

[3] K. Chandy, L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM*, Vol. 3(1), 1985.

[4] W. Chen, S. Toueg, M. K. Aguilera. On the Quality of Service of Failure Detectors. *IEEE Transactions on Computers*, Vol. 51(5), 2002.

[5] Colt Library API documentation.
http://dsd.lbl.gov/~hoschek/colt/api/index.html

[6] Colt Library website.
http://dsd.lbl.gov/~hoschek/colt/

[7] $JAVA^{TM}$ 2 Platform, Standard Edition, ver. 1.4.2 - API Specification.
http://java.sun.com/j2se/1.4.2/docs/api/

[8] A. M. Law, W. D. Kelton. Simulation, Modeling and Analysis. *McGraw-Hill*, 2000.

[9] D. L. Mills. Internet Time Synchronization: the Network Time Protocol. *IEEE Trans. Communications 39*, 10, Oct. 1991.

[10] Neko v0.8 website.
http://lsrwww.epfl.ch/neko/

[11] NekoStat v0.8 website.
http://www.arcetri.astro.it/~falai/nekostat.html

[12] RFC 1305. Network Time Protocol (Version 3).
http://rfc.sunsite.dk/rfc/rfc1305.html

[13] RFC 2330. Framework for IP Performance Metric.
http://www.ietf.org/rfc/rfc2330.txt

[14] P. Urban. Evaluating the performance of distributed agreement algorithms: tools, methodology and case studies. *Ph.D. Thesis, École Polytechnique Fédérale de Lausanne*, 2003.

[15] P. Urban, X. Defago, A. Schiper. Neko: a single environment to simulate and prototype distributed algorithms. In *Proceedings 15th International Conference on Information Networking (ICOIN-15)*, Feb. 2001.