Università degli Studi di Roma Tor Vergata

Facoltà di Ingegneria

Corso di Informatica Mobile

SLASH

Professore:
Vincenzo Grassi

Studenti:
Simone Notargiacomo
Lorenzo Tavernese
Ibrahim Khalili

Anno Accademico 2007-2008

Contents

Abstract

Nel mondo dell'Informatica si è diffuso da molto tempo il bisogno di sviluppare applicazioni per dispositivi mobili, quali notebook, smartphone, tablet ed altro ancora; in particolare si è raggiunta la necessità di sviluppare applicazioni distribuite per dispositivi mobili, ovvero il *Mobile Computing*. Sono nate molte nuove tecniche che hanno apportato migliorie alla comunicazione tra dispositivi, come la stipulazione di *SLA* (Service Level Agreement: Contratti basati sul livello di servizio) tra richiedenti e fornitori di servizi, ed infine le *dsm* (Distribuited Shared Memory). Durante il corso di Informatica Mobile è stato richiesto di realizzare un sistema che simulasse il comportamento di alcuni richiedenti e fornitori dopo aver stipulato un contratto sulla qualità del servizio, sfruttando *dsm* per la comunicazione dei dati. In questa relazione verranno trattate le problematiche incontrate nella progettazione del sistema, ed in particolare tutte le scelte implementative effettuate, nonchè i modelli matematici ed empirici. Inoltre verranno presentati anche dei test di esecuzione con i relativi dati sulle performance; infine si riporteranno le conclusioni a cui si è giunti ed i possibili sviluppi futuri.

Chapter 1

Introduzione

Per comprendere a fondo l'entità di tale progetto si riportano di seguito le specifiche del progetto, in quanto consente al lettore di entrare nell'ottica del problema, inoltre si fornirà una breve panoramica relativa al paradigma DSM, agli agenti mobili e al framework JADE.

1.1 Specifiche problema

1.1.1 Logica applicazione

Si richiede di progettare una architettura di supporto al monitoraggio e controllo di SLAJing-Helal-Elmagarmid (1999) in ambiente (possibilmente) mobile. L'architettura del servizio è basata sulla definizione di un certo numero di componenti "logici": SLA Checker (SC), Context Manager (CM), Resource Monitor (RM). Tali entità interagiscono tra loro unicamente tramite il meccanismo DSM ("tuple space"). Il ruolo di tali entità viene descritto come segue:

SLAchecker (SC): data una coppia fornitore/richiedente servizio, che ha stipulato un SLA, SC ha il compito di controllare il rispetto dei parametri del contratto sia da parte del fornitore che del richiedente, e segnalare eventuali violazioni ad entrambi. A questo scopo, SC raccoglie infor-

mazioni fornite da opportuni componenti di tipo Monitor presenti sia sul nodo del fornitore che del richiedente, relative a (per esempio):

- tempo di risposta osservato per una richiesta;
- affidabilità (completamento con successo) di una richiesta;
- intervallo di tempo tra due richieste consecutive.

I dati "grezzi" ricevuti dai componenti di monitoraggio vengono elaborati da SC per calcolare i valori degli indici di interesse.

Context Manager (CM): è un componente associato a un particolare nodo di elaborazione e il suo ruolo è quello di fornire informazioni su vari tipi parametri che caratterizzano il contesto di esecuzione di componenti presenti su quel nodo, p.es.:

- utilizzazione cpu;
- RAM disponibile;
- memoria stabile (disco, o altro) disponibile;
- tipo di rete e banda disponibile;
- energia disponibile.

Resource Monitor (RM): un componente di questo tipo fornisce le informazioni relative a una delle risorse elencate sopra.

1.1.2 Ambiente d'uso

L'ambiente in cui si immagina che il servizio di controllo di SLA venga realizzato è costituito, in generale, da una molteplicità di nodi (fissi o mobili) con vari livelli di disponibilità di risorse interne (memoria, cpu, ecc.), connessi tra loro da infrastrutture di comunicazione di varia qualità. Su tali nodi sono in esecuzione componenti che offrono/richiedono servizi. Ogni volta che una coppia fornitore/richiedente stipula un SLA, il controllo di questo SLA viene affidato a un componente SC.

1.1.3 Lavoro progettuale

Si richiede di progettare e realizzare, utilizzando la piattaforma JADE (`http://jade.tilab.com`), l'architettura indicata nella sezione precedente. In particolare, occorre definire una localizzazione dei componenti e organizzazione del modello DSM (basato sulla realizzazione di uno o più "tuple space") che sia adeguata alla esecuzione del servizio di controllo SLA in un ambiente possibilmente mobile, caratterizzato da possibile scarsità di risorse per i componenti in esecuzione su determinati nodi. Il livello di adeguatezza andrà valutato rispetto alla capacità di ottimizzare misure di prestazione quali:

- traffico generato su rete;
- consumo di energia da parte di nodi mobili;
- carico computazionale/di memorizzazione per nodi mobili;

tenendo anche conto del fatto che il contesto (disponibilità di risorse) in cui opera il servizio di controllo SLA può variare nel tempo, per esempio per effetto della mobilità di alcuni nodi.

1.2 Distributed Shared Memory

Il paradigma DSM fornisce agli host in un sistema distribuito la vista di uno spazio comune condiviso attraverso spazi di indirizzamento disgiunti, in cui la sincronizzazione e la comunicazione fra i partecipanti avvengono tramite operazioni sui dati comuni. La nozione di *tuple space* è stato originariamente integrato in Linda, e fornisce una semplice e potente astrazione per accedere alla memoria condivisa. Un *tuple space* è composto di una collezione di tuple ordinate, accessibili in egual modo da tutti gli host del sistema distribuito. La comunicazione tra hosts avviene tramite l'inserimento/rimozione di tuple nel/da *tuple space*. Possono essere eseguite tre operazioni di base: *out()* per esportare una tupla nel *tuple space*, *in()* per importare (e rimuovere) una

tupla e *read()* per leggere, senza rimuovere, una tupla. Il modello di interazione offre disaccoppiamento sia spaziale che temporale, pochè consumatore e produttore non hanno bisogno di conoscersi e il creatore di una tupla non ha bisogno di sapere l'uso che verrà fatto di tale tupla. Nonostante tutto, non si ha un disaccoppiamento da un punto di vista della sincronizzazione dal lato del consumatore.

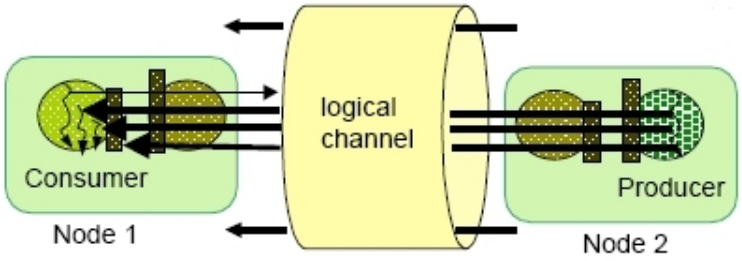

Figure 1.1: DSM

1.3 Mobile Agent

Un agente mobile è un componente software in grado di traserirsi su nodi remoti di una rete e di interagire con le risorse nei nodi visitati, scoprendo i servizi offerti. Dalla definizione fornita si arguisce che l'Agente deve possedere un certo grado di *intelligenza*, sia per le decisioni che deve prendere una volta spedito, sia per la capacità di memorizzare i risultati ottenuti su ciascun nodo (capacità di mantenere ed aggiornare il suo stato). La possibilità per un agente di muoversi e di conseguenza il comportamento che questo può assumere si deve basare sugli obiettivi definiti nel suo stato. Tale concetto di mobilità unito a quello di agente ha permesso di conferire a questo componente la capacità di cooperare al fine di poter risolvere un dato problema. Il processo di trasferimento del codice e del relativo stato può avvenire secondo due modalità differenti:

- Clonazione: il codice e lo stato vengono duplicati nel nodo di arrivo

senza rimuoverli nel nodo di partenza

- Migrazione: il codice e lo stato vengono copiati nel nuovo nodo e rimossi dal vecchio.

Terminato il processo di migrazione l'agente deve essere in grado di interagire con le risorse locali al fine di usufruire dei servizi offerti dal nodo ospite. Il meccanismo di interazione con le varie risorse presenti viene facilitato grazie all'utilizzo della tecnologia ad oggetti, al contrario del procedimento di *discovery* dei servizi, il quale prevede un modello di *introspezione* degli oggetti, che mantenga le informazioni accessibili dinamicamente. Si può quindi intuire che se da un punto di vista teorico si ha grande convenienza nell'utilizzo di Agenti, in pratica non sempre risulta possibile usufruire di tali vantaggi. E' importante evitare di cadere nella tentazione assolutista con la quale si cerca di applicare una tecnologia in ogni possibile dominio o circostanza. Possiamo sicuramente affermare che la soluzione ad Agenti può essere più conveniente rispetto ad altri approcci a seconda di circostanze quali domini, infrastrutture di rete, compatibilità con altre tecnologie di contorno.

1.4 JADE

JADE è un framework interamente realizzato in Java, che permette di semplificare l'implementazione di *Multi Agent Systems* (MAS),associando diversi container agli agenti, i quali hanno la possibiltà di comunicare sulla stessa o su piattaforme differenti. I vari containers possono essere raggruppati formando così una *platform*.

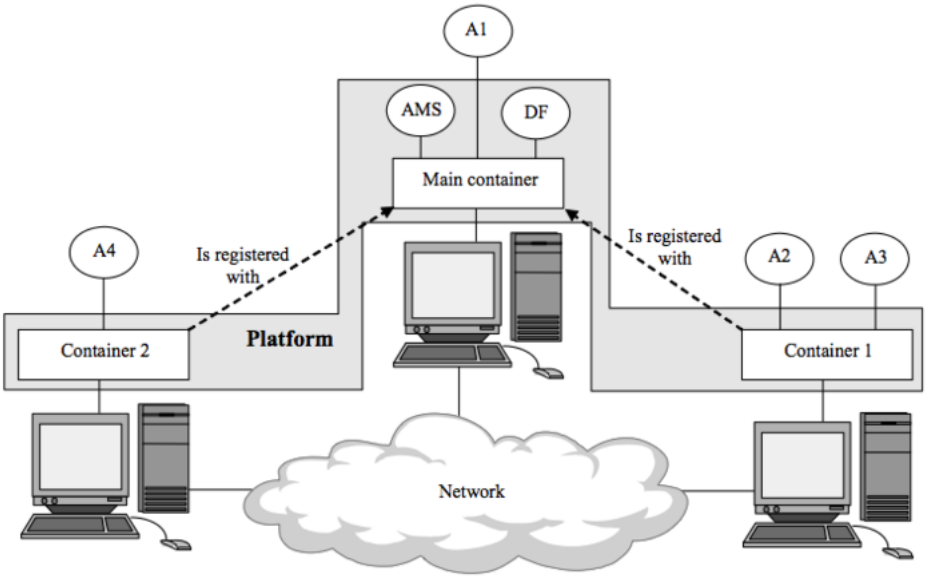

Figure 1.2: architettua di JADE

Ogni platform deve avere un *Main Container*, il quale possiede due agenti specializzati chiamati rispettivamente *AMS agent* e *DF agent*:

- L'*AMS* (Agent Management System) ha il compito di svolgere la funzione di supervisore controllando l'accesso e l'uso della piattaforma da parte degli agenti.Inoltre mantiene un registro degli identificatori degli agenti (AID) e del loro stato, in quanto ogni agente è obbligato a registrarsi per ottenere un AID valido .

- Il *DF* (Directory Facilitator) ha il compito di implementare un servizio di pagine gialle che pubblicizza i servizi degli agenti della piattaforma in modo che altri agenti che richiedono tali servizi possono trovarli.

Le varie operazioni che gli agenti sono in grado di compiere vengono definite attaverso oggetti di tipo *Behaviour*, i quali forniscono l'implementazione dei task rischiesti.Esistono diversi tipi di behaviour predefiniti (composite behaviour,sequential behaviour, parallel behaviour, sender behaviour), tuttavia

è possibile definire behaviour personalizzati a seconda della complessità del task richiesto. Il modello di comunicazione utilizzato dagli agenti per interagire si basa sullo standard ACL (Agent Communication Language), nel quale i messaggi sono scambiati in modo asincrono. La struttura dei vari messaggi viene definita all'interno della classe java (jade.lang.acl.ACLMessage) fornita dal framework stesso. Tale classe fornisce i metodi per impostare e ottenere i valori dei campi definiti da FIPA (Foundation for Intelligent Physical Agent) per l' ACL.

Chapter 2

Architettura

Nella specifica del problema (rif. 1) è stato riportato il funzionamento del sistema da realizzare ei relativi componenti neccessari. Per adempire alle richieste della specifica si è deciso di sviluppare l'applicazione dando priorità ai punti fondamentali, dopodichè sono stati trattati gli aspetti secondari come la taratura dei parametri ei meccanismi di richiesta dei servizi. Nella prossima sezione verrà riportata l'architettura del sistema.

2.1 Sistema

2.1.1 Composizione Nodi

Tutta l'applicazione è stata progettata e sviluppata considerando la presenza di molteplici nodi richiedenti e fornitori, per rendere possibile una simulazione più reale e complessa. Inizialmente sono state prese delle decisioni inerenti la composizione generale di ogni nodo della nostra rete, in particolare si è deciso di supportate nodi di due tipi:

wired: nodo fisso collegato tramite cavo;

wireless: nodo mobile collegato tramite *wireless*.

Effettuando tale scelta si sono scatenate un'altra serie di decisioni attinenti l'hardware dei nodi, ovvero l'energia, la memoria ram, il disco e il carico

della cpu. Tali componenti sono molto dipendenti dal tipo di collegamento del nodo, in quanto un link di tipo wireless ha bisogno di maggiore calcoli e quindi un utilizzo di energia maggiore.

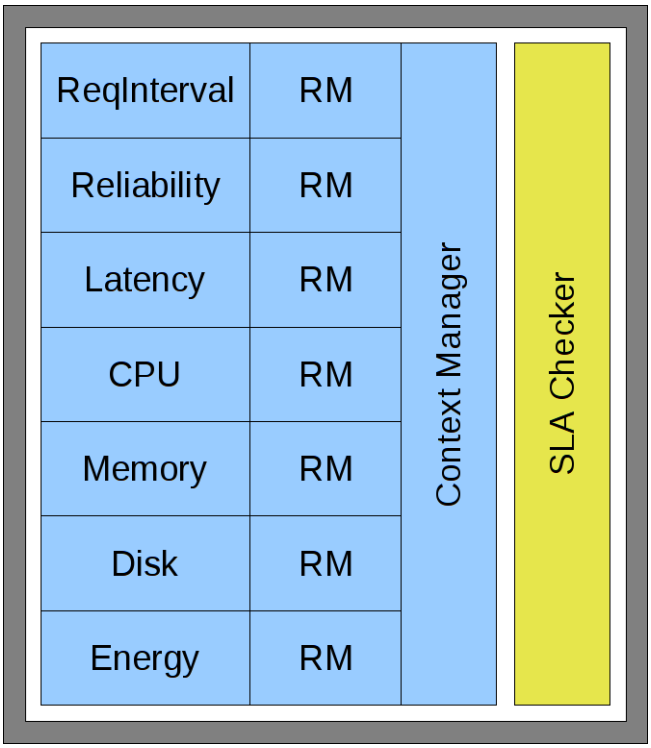

Figure 2.1: Componenti Nodo

Dalla figura 2.1 si può notare come è strutturato un singolo nodo, in particolare si può notare come i componenti da monitorare (cpu, memory, disk, energy, latency, reliability, reqInterval) siano in stretto contatto uno a uno con un *Resource Monitor*. I *Resource Monitor* immettono informazioni dei componenti monitorati sul dsm. Il *Context Manager* si occupa di raccogliere tali dati e raggrupparli in un solo oggetto che viene a sua volta immesso sul dsm. Lo *SLA Checker* (o *SC*), si trova in genere a contatto con il *CM* questo perchè raccoglie principalmente le informazioni sul contesto dei vari nodi. Tutte queste parti elencate saranno chiamate d'ora in poi *Agenti*, considerando che ci si trova in ambiente di programmazione ad agenti (ovvero

Jade). Gli agenti delle risorse sono stati realizzati per consentire una simulazione più realistica, ovvero ogni agente risorsa non fa altro che generare un valore di utilizzazione basato su vari parametri (ved. cap 4).

2.1.2 Composizione Rete

La rete che si è deciso di creare è formata da tanti di questi nodi, sia richiedente che fornitore. Nello specifico ogni nodo fornitore si occupa di fornire un servizio generico, il quale viene registrato nelle pagine gialle del sistema. Ogni richiedente, invece, può richiedere il servizio ad uno qualsiasi dei fornitori disponibili. Questa operazione è stata resa possibile per fare in modo che la simulazione si attenesse ad un tipico caso reale. Per quanto riguarda lo *SLA Checker* si è progettato il sistema considerando che tale agente dovesse *migrare* da un nodo all'altro in base alle condizioni del contesto. Un esempio della rete in questione con 3 nodi può essere visto in figura 2.2. Si può notare che lo *SC* si trova solo su uno dei nodi della rete in quanto deve poter migrare fra di loro.

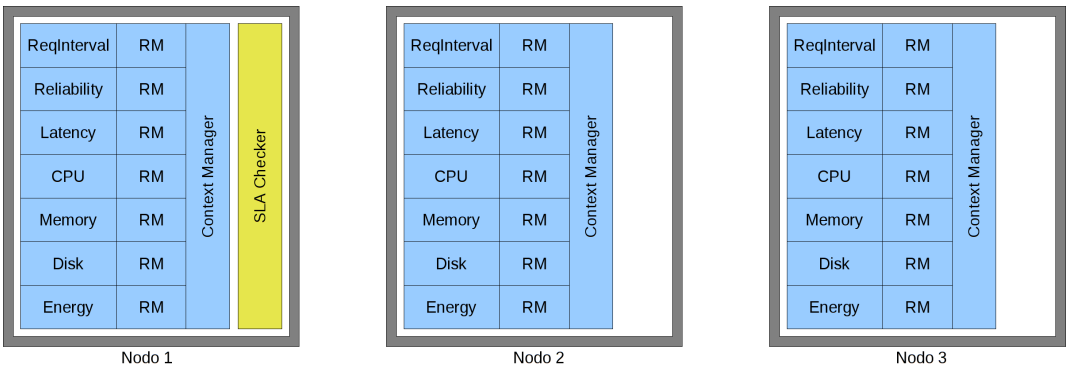

Figure 2.2: Esempio di rete con 3 nodi

2.2 Funzionamento

Per spiegare il funzionamento del sistema in questione si è deciso di definire uno scenario d'uso e quindi descrivere il comportamento dei relativi nodi.

2.2.1 Scenario d'uso

Un tipico scenario d'uso potrebbe essere una rete con 3 nodi di cui:

- 1 nodo fornitore;
- 2 nodi richiedenti;

in particolare ci troviamo in una situazione in cui ognuno dei due nodi richiede un servizio che si attenga al contratto prestabilito con il nodo fornitore. Ogni contratto contiene le seguenti informazioni:

Publisher: fornitore del servizio;

Subscriber: richiedente del servizio;

ReqInterval: intervallo di tempo tra le richieste del richiedente;

Latency: tempo impiegato per espletare il servizio;

Reliability: affidabilità di servizio da parte del fornitore.

Tali informazioni servono per fare in modo che sia il fornitore che il richiedente facciamo il possibile per attenersi ai valori specificati nel contratto. Su uno dei nodi del sistema è presente lo *SLAChecker* che si occupa di controllare la validità di tutti i contratti stipulati; nel caso in cui le condizioni di un contratto vengono violate da uno dei due nodi allora lo SC informerà immediatamente entrambi i nodi di tale condizione. Tutti i componenti dei nodi sono fortemente dipendenti dalla presenza dello SC, in quanto comporta un aumento oneroso in termini di calcoli. A causa delle risorse limitate di alcuni nodi (come l'energia) è necessario effettuare un controllo sullo stato attuale dei componenti dei nodi per evitare di sovraccaricarlo, in modo tale che in

caso di necessità lo *SC* migri su un nodo con condizioni di carico migliori. La selezione del nodo migliore viene fatta utilizzando una specifica politica di selezione (vedi cap. XXX). Un esempio di migrazione può essere visto nelle due figure seguenti, in cui nel primo caso lo SC si trova sul nodo 1, mentre nel secondo caso lo SC è migrato sul nodo 2 a causa di scarsità di risorse sul nodo 1.

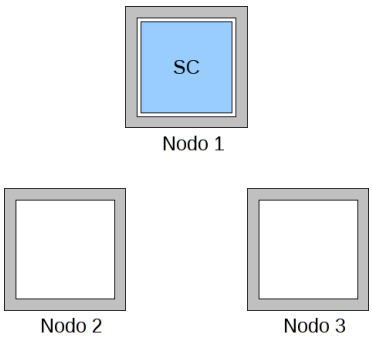

Figure 2.3: Esempio di scenario

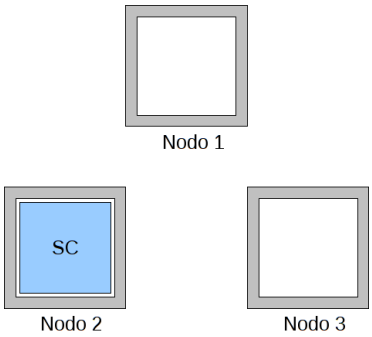

Figure 2.4: Esempio di scenario

2.2.2 Distribuited Shared Memory

Per consentire la comunicazione tra i nodi del sistema è stato aggiunto un ulteriore livello nell'architettura in questione, ovvero il *dsm*. Tale componente è stato progetto sfruttando i meccanismi di comunicazione di Jade, in

particolare lo scambio di messaggi tra server e client dsm avviene tramite messaggi di Jade.

DsmServer

E' stato realizzato un agente che si occupa principalmente della ricezione e la computazione delle richieste. Nello specifico quando il server riceve una specifica richiesta effettua la rispettiva operazione sul database dsm.

DsmData

Questo componente è necessario per la gestione del dsm e delle tuple, in quanto consente operazioni di `IN` , `OUT` , `UPDATE` e `READ` .

DsmClient

Quest'ultimo componente è fondamentale per ogni agente che deve inviare o riceve informazioni dal tuple space, infatti consente l'invio delle richieste direttamente all'agente server che provvederà ad esaurirle.

2.2.3 Context Manager

Il Context Manager, come già anticipato, si occupa di raccogliere tutte le informazioni sulle risorse del proprio su un contenitore chiamato `Context` , nello specifico il contenuto è il seguente:

cpu: valore del processore;

ram: valore della ram presente nel nodo;

memory: valore percentuale dell'occupazione di memoria del nodo;

energy: valore di energia residua;

latency: latenza di trasmissione attuale del nodo publisher;

reliability: valore dell'affidabilità del nodo;

reqInterval: frequenza di richieste;

location: contiene informazioni su cui risiede CM;

network: tipo di rete del nodo;

bandwidth: banda di rete del nodo.

Per raccogliere tali informazioni il CM effettua delle operazioni di `IN` sul dsm ad intervalli predefiniti. Raccolti i dati immette il contesto sul dsm tramite operazioni di `OUT` . Il CM si occupa inoltre si richiede ed accettare contratti SLA in base al tipo di nodo in cui si trova, se il nodo è di tipo *publisher* accetta contratti invece se è di tipo *subscriber* richiede contratti. Una volta accettatto un contratto viene rinviato sul dsm tramite `OUT` per essere passato allo SC.

2.2.4 SLA Checker

E' il componente su cui si basa tutto il funzionamento del sistema realizzato, in quanto è l'agente che si occupa di controllare la validità dei contratti e migrare quando necessario. Per effettuare tali operazioni è necessario essere a conoscenza del contesto dei vari nodi e dei contratti SLA stipulati. Lo SC richiede continuamente i contesti dei vari nodi dal dsm, inoltre memorizza tutti i contratti che sono stati stipulati per essere a conoscenza delle relazioni tra i nodi. La conoscenza di tutte le informazioni descritte è necessaria all'algoritmo di *checking*, in quanto ha bisogno di tante informazioni per scoprire il miglior nodo su cui migrare (ved. 4).

Chapter 3

Dsm

In questo capitolo verrà riportato dettagliatamente il funzionamento e la strutturazione del componente dsm necessario alla comunicazione tra agenti, nello specifico verra spiegato come è stato realizzato il tuple space e tutti i meccanismi ad esso correlati. Per il progetto in questione sono stati realizzati i comandi prettamente necessari, che sono: IN, OUT, UPDATE e READ.

3.1 DsmData

Per la gestione dei dati è stato realizzato una sorta di database volatile tramite lo sfruttamento di *Hashtable* e *Queue*. In particolare si realizzata una struttura del genere:

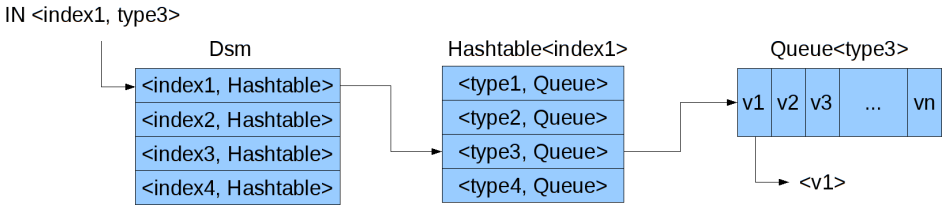

Figure 3.1: Esempio di IN nel Dsm

Dalla figura è riportato un esempio delle operazioni eseguite sul database nel momento in cui si effettua una richiesta `IN <index1, type3>` , ovvero si

richiede un valore di tipo 3 che si trova nell'indice 1. Le operazioni eseguite sono elencate di seguito:

1. Quando perviene la richiesta al db si cerca immediatamente la presenza di una chiave `index1` nell'hashtable di primo livello;

2. se la chiave è presente si prende la hashtable associata e si procede con il controllo delle presenza della chiave `type3` all'interno dell'hashtable di secondo livello;

3. se la chiave è presente si prende la coda che gli è associata e si estrae il valore che si trova in testa.

Operazioni analoge vengono effettuate per gli altri tipi di operazioni. In particolare le operazioni implementate hanno il seguente comportamento:

OUT: equivale alla tupla `OUT <index1, type3, v1>` e consiste nell'inserire un valore sul tuple space.

IN: rappresentata dalla tupla `IN <index1, type3>` ed il suo scopo è reperire il valore in testa alla coda e quindi eliminarlo dal tuple space.

READ: tupla uguale a IN, ma ha l'unica differenza di non eliminare il valore dopo averlo reperito.

UPDATE: tupla come OUT con la differenza che se il valore esiste lo sostituisce.

3.2 DsmClient

Questo componente consente la comunicazione con il server dsm sfruttando lo scambio di messaggi che mette a disposizione Jade con il suo *framework*. Il modulo in questione consente due meccanismi principali che sono: l'invio di messaggi bloccanti e l'invio di messaggi non bloccanti. Per quanto riguarda i messaggi bloccanti (IN e READ) si ha:

1. Il Client invia la richiesta al server ed attende la risposta;
2. Il Server riceve la richiesta, la esaurisce ed invia la risposta;
3. Il Client riceve la richiesta e la ritorna all'agente chiamante.

I messaggi non bloccanti (OUT e UPDATE) invece vengono inviati nel seguente modo:

1. Il Client invia la richiesta al server e ritorna all'agente chiamante;
2. Il Server riceve la richiesta e la esaurisce;

3.3 DsmServer

Tale componente può essere inteso come un *proxy* in quanto non fa altro che tradurre le richieste del client a operazioni sul `DsmData` e quindi ritornare una risposta (in caso di messaggi bloccanti). Una caratteristica importante di questo agente è che può migrare tra i vari nodi in base alla disponibilità delle risorse dei nodi, nello specifico non fa altro che seguire lo SC, ovvero quando lo SC migra su un altro nodo il DsmServer lo segue (rif SC).

Chapter 4

Modello Matematico

Il modello matematico riguarda tutta la parte generativa dei valori delle risorse e dell'indice di occupazione dei nodi. Molte delle relazioni che sono state utilizzate sono state ricavate empiricamente grazie ad un operazione di *tuning* che ci ha consentito di raggiungere una situazione il più reale possibile.

4.1 Valori risorse

Le risorse del sistema realizzato possono essere suddivise in due gruppi: dipendenti dalla presenza dello SC (e Dsm) e indipendenti dalla loro presenza. Quanto detto è stato fatto per rappresentare più realmente un caso normale. Si consideri il seguente esempio, ovvero un nodo mobile con connessione *wireless* e un nodo fisso con connessione *wired*. Il nodo mobile viene collegato e scollegato dalla rete elettrica quando necessario. Sul nodo mobile sono in esecuzione delle normali operazioni di richiesta di informazioni verso il nodo fisso. Sul nodo fisso è in esecuzione il server che risponde alle risposte di vari nodi mobili. Lo SC è al momento sul nodo fisso, questo implica un carico computazione molto elevato per tale nodo, quindi se necessario tale agente SC migrerà sul nodo mobile che al momento ha molte risorse disponibili. Per consentire questo meccanismo di migrazione è stato necessario realizzare delle formule di generazione adattabili al loro stato attuale (presenza o non

presenza dello SC oppure collegato o scollegato dalla rete elettrica). Tutti i valori generati rappresentano delle percentuali. Di seguito vengono riportate tutte le formule usate per la generazione.

4.1.1 Cpu

La risorsa *cpu* dipende fortemente dalla presenza dello SC, infatti si è deciso di applicare la seguenti relazioni, la prima nel caso "senza SC"

$$cpu = (\alpha \cdot 100) mod 60 \tag{4.1}$$

in cui α rappresenta un valore casuale da 0 a 1. Il modulo 60 è stato usato per limitare il valore a 60. Nel caso "con SC" invece è stata usata la seguente relazione:

$$cpu = ((\alpha \cdot 100) mod 60) + 40 \tag{4.2}$$

in cui si aggiunge il valore 40 che rappresenta il carico supplementare dovuto alla presenza dello SC.

4.1.2 Ram

La risorsa viene generata come per la cpu.

$$ram = (\alpha \cdot 100) mod 60 \tag{4.3}$$

Nel caso "con SC" invece è stata usata la seguente relazione:

$$ram = ((\alpha \cdot 100) mod 60) + 40 \tag{4.4}$$

4.1.3 Memory

La memoria ha un comportamento leggermente più complesso, infatti è stato realizzato un meccanismo di riempimento e svuotamento della memoria dipendente dalla presenza dello SC. All'avvio del nodo viene generato un valore iniziale tramite la seguente formula:

$$memory = (\alpha \cdot 100) mod 60 \tag{4.5}$$

Ad ogni *tick* si modifica il valore nel seguente modo:

$$memory = memory + 0.2 \cdot ((\alpha \cdot 100) mod 10) \tag{4.6}$$

Quest'ultima relazione consiste nell'aggiungere al valore precedente il 20% di un valore modulo 10, grazie a ciò si ottiene un incremento molto lieve dell'utilizzazione della memoria. Nel caso "senza SC" invece si ha

$$memory = memory - 0.2 \cdot ((\alpha \cdot 100) mod 10) \tag{4.7}$$

a differenza del caso precedente si sottrae il nuovo valore da quello vecchio, questo sta ad indicare una diminuzione dell'utilizzazione. Inoltre nel caso in cui la memoria raggiunge valori minori di 0 o maggiori 100 si rigenera il valore della memoria con la formula usata inizialmente (ved. 4.5).

4.1.4 Energy

Nel caso del componente riguardante l'energia sono state applicate delle formule più adattative, in quanto il suo valore dipende dal collegamento o scollegamento alla rete elettrica e dalla presenza o non presenza dello SC. Inoltre dipende dal tipo di connessione, ovverro se è wireless o wired. All'avvio viene generato il valore tramite la seguente formula.

$$memory = (\alpha \cdot 100) mod 60 \tag{4.8}$$

Ad ogni tick dell'agente si può avere una delle seguenti formule in base alle condizioni. In caso di connessione wired si ha:

$$energy = 100 \tag{4.9}$$

considerando che il wired è stato assunto come nodo fisso collegato alla rete elettrica. In caso di connessione wireless si deve distinguere dal caso in cui è connesso alla rete elettrica (`powerOn`) e il caso in cui non lo è (`!powerOn`). Per rendere reale i valori generati si è realizzato un meccanismo per gestire la connessione e sconnessione dalla rete elettrica, ovvero:

$$\text{se } energy \leq 10 \text{ then } powerOn = true \tag{4.10}$$

$$\text{se } energy \geq 99 \text{ then } powerOn = false \tag{4.11}$$

Con la rete elettrica collegata si ha:

$$energy = energy + 0.8 \text{ se SC non presente,} \tag{4.12}$$

$$energy = energy + 0.6 \text{ se SC presente} \tag{4.13}$$

in caso di rete elettrica scollegata si ha:

$$energy = energy - 0.2 \text{ se SC non presente,} \tag{4.14}$$

$$energy = energy - 1 \text{ se SC presente} \tag{4.15}$$

4.1.5 Latency

Nel caso della latenza è stato usato un meccanismo dipendente solo dalla presenza dello SC e dal tipo di connessione. In particolare se nel nodo è presente lo SC oppure ha una connessione wireless si ha:

$$latency = ((\alpha_1 \cdot 100) mod 60) + \left(\left(\frac{\alpha_2 \cdot 100}{\frac{\beta}{50}} \right) mod 40 \right) \tag{4.16}$$

come si può notare vengono generati due numeri, uno modulo 60 ed uno modulo 40 in modo da avere un numero massimo di 100. Inoltre è presente un valore β che rappresenta la banda del nodo.

4.1.6 Reliability

Il valore della affidabilità del canale viene generato nello stesso modo della latenza, infatti si ha:

$$reliability = ((\alpha_1 \cdot 100) mod 60) + \left(\left(\frac{\alpha_2 \cdot 100}{\frac{\beta}{50}} \right) mod 40 \right) \tag{4.17}$$

4.1.7 ReqInterval

In quest'ultima risorsa la generazione avviene molto semplicemente generando un numero casuale da 0 a 100.

$$reqInterval = \alpha \cdot 100 \tag{4.18}$$

4.2 SLA Checker

Lo SC essendo il componente centrale di tutto il sistema ha richiesto un lavoro molto accurato per la realizzazione di un buon algoritmo e di un indice di valutazione efficiente. Per algoritmo si intendono le operazioni che vengono svolte dallo SC ciclicamente (ad ogni tick dell'agente), ovvero il controllo sulla validità dei contratti e il controllo della necessità di migrazione. Per indice di valutazione si intende il valore che viene generato per ogni nodo dai valori del suo contesto, tale indice viene utilizzato dallo SC per trovare il nodo su cui eventualmente migrare.

4.2.1 Algoritmo controllo

L'algoritmo realizzato consente il controllo della violazione dei contratti e il controllo della necessità di migrazione. In particolare tali operazioni possono essere effettuate grazie alla conoscenza di tutte le informazioni sui nodi del sistema, ovvero il contesto e i contratti. L'algoritmo è suddiviso in due parti, la prima si occupa del controllo della validità dei contratti.

SLA Contract checking

1. si prende un contratto SLA dalla lista dei contratti;
2. si recuperano i dati del contesto dei due partecipanti al contratto (Publisher, Subscriber);
3. si controlla la validità del contratto tramite la relazione seguente:

$$latency_{pub} > latency_{sla}$$

OR

$$reliability_{pub} > reliability_{sla}$$

OR

$$reqInt_{sub} > reqInt_{sla}$$

4. se tale relazione risulta valida allora il contratto è stato violato.

Dalla relazione riportata si può notare la presenza dei valori di `latency` , `reliability` e `reqInt` che hanno come pedice `pub` , `sub` e `sla` che indicano rispettivamente il Publisher, il Subscriber e il contratto SLA. La relazione riportata non fa altro che controllare se il publisher viola il contratto non garantendo la latenza e l'affidabilità precedente, oppure se il Subscriber viola il contratto effettuando più richieste di quante stabilite dal contratto SLA.

SC Migration checker

La seconda parte di cui si occupa lo SC è la verifica della necessità di migrazione su altri nodi. In particolare tale operazione consiste nel valutare l'indice del contesto di ogni nodo, controllando se il valore supera una certa soglia prefissata, nel qual caso sarebbe necessario migrare su un nuovo nodo. Il calcolo dell'indice di utilizzazione viene effettuato con la seguente formula:

$$index = cpu \cdot 0.23 + ram \cdot 0.23 + memory \cdot 0.23 + (100 - energy) \cdot 0.3 \quad (4.19)$$

Il valore *index* calcolato rappresenta una percentuale che va da 0 a 100 che indica l'utilizzazione del nodo valutato. L'indice è il fulcro dell'algoritmo di *Migration checking*, in quanto la migrazione si basa proprio sul confronto di tale valore. Verranno ora riportati i passi dell'algoritmo:

1. Si recupera il nodo migliore su cui migrare tramite la funzione `getBestNode()` ;
2. si seleziona il contesto attuale del nodo in cui si trova lo SC;
3. si verifica che l'indice del contesto attuale sia maggiore dell'indice di migrazione;
4. se la condizione è verificata si effettua la migrazione sul nodo risultato migliore;

5. si notifica, tramite dsm, l'avvenimento della migrazione a tutti nodi presenti;

6. se la condizione non si verifica non viene effettuata la migrazione.

Best Node

La funzione `getBestNode()` riportata nell'algoritmo precedente viene descritta dettagliatamente di seguito:

```
AID next, iter;
Enumeration<AID> keys = sc.getContextTable().keys();
Context context;
float cpu, ram, memory, energy;
float total=500, iter=0;

while(keys.hasMoreElements()) {
  next = keys.nextElement();
  context = sc.getContextTable().get(next);
  if(context!=null) {
    cpu = context.getCpu();
    ram = context.getRam();
    memory = context.getMemory();
    energy = context.getEnergy();
    iter = cpu + ram + memory + (1-energy);
    if(cpu < Context.CPU_LIMIT && ram < Context.RAM_LIMIT && memory <
        Context.MEMORY_LIMIT && energy > Context.ENERGY_LIMIT && iter <
        total) {
      best = next;
      total = iter;
    }
  }
}
```

Lo scopo dell'algoritmo riportato è di recuperare il contesto che ha i valori "migliori", ovvero che si attiene alla seguente relazione:

$$cpu < cpu_{limit}\ AND\ ram < ram_{limit}\ AND$$

$$memory < memory_{limit}\ AND\ energy > energy_{limit}\ AND\ \theta < min$$

in cui i valori con pedice *limit* indicano il massimo numero per cui si accetta il valore come "migliore". θ invece rappresenta il minimo attuale, calcolato

tramite 4.20

$$\theta = cpu + ram + memory + (1 - energy) \tag{4.20}$$

in ogni ciclo viene calcolato tale valore e se risulta essere minore del minimo attuale si sostituisce min con tale valore.

Chapter 5

Simulazione

Per valutare le prestazioni del sistema realizzato si sono effettuate numerose simulazioni. Tutti i test condotti sono stati basati su una specifica configurazione di nodi, in particolare 1 nodo Publisher e 2 Subscriber. I due nodi consumer richiedono la stipulazione di un contratto al producer, il quale espleta tutte le richieste. Si riporta ora un esempio di esecuzione con relativa descrizione ed analisi dei dati.

5.1 Esecuzione

In principio si avvia il nodo Producer il quale andrà in attesa di richieste dei consumatori. In seguito si avviano i due Consumer (o Subscriber) i quali richiedono ed ottengono un contratto con il Subscriber. Stabiliti i contratti inizia la vera fase di simulazione in cui ogni nodo genera valori casuali per le proprie risorse e lo SC raccoglie tali dati per valutare gli SLA e la necessità di migrazione.

5.1.1 Scenario

I tre nodi dello scenario d'esecuzione hanno le seguenti caratteristiche:

1. Publisher con connessione WIRED e 512Kb di banda;

2. subscriber con connessione WIRELESS e 256Kb di banda;

3. subscriber con connessione WIRELESS e 256Kb di banda.

5.1.2 Generazione contratti SLA

Il contratto SLA, ovvero i livelli di latency, reliability e reqInterval vengono generati casualmente dal publisher nel momento in cui raccolgono una richiesta di contratto. Tale operazione può essere vista nel log di esecuzione del programma simulativo.

1. Il Subscriber richiede un contratto con un qualsiasi publisher disponibile;

2. Il publisher legge la richiesta dal dsm;

   ```
   SLAContract-request received from cm2
   ```

3. il publisher genera i parametri del contratto casualmente e lo immette sul dsm;

4. lo SC rileva la presenza del contratto e lo aggiunge alla sua lista.

   ```
   Added Contract
   ```

5.1.3 Migrazione

Inizialmente lo SC risiede sul primo nodo, ovvero il publisher (ved. 5.1).

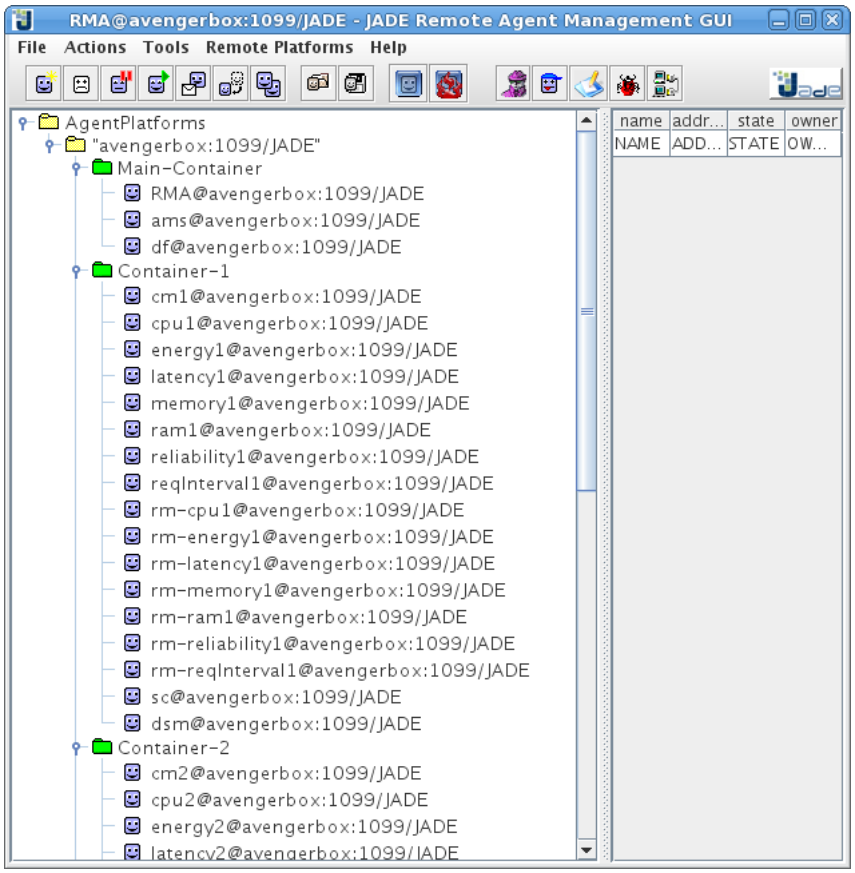

Figure 5.1: Migrazione non avvenuta

In questa figura si può notare la presenza del dsm e dello SC sul primo nodo, in quanto la simulazione è stata appena avviata. Dal log si può rilevare che lo SC ha già iniziato a controllare la necessità di migrazione e che al momento si trova sul Container-1 ovvero il nodo 1. Il valore index del nodo attuale non ha ancora superato il valore di soglia (52), di conseguenza lo SC rimane sul nodo attuale.

```
Sono su Container-1
Context[cm3]--> cpu: 21.126825, ram: 12.573414, memory: 35.398045, energy:
    64.588715, index: 26.515991
Context[cm1]--> cpu: 78.77115, ram: 63.975082, memory: 15.567526, energy:
    100.0, index: 36.412167
Context[cm2]--> cpu: 50.302402, ram: 11.7872715, memory: 1.3100842, energy:
    91.46843, index: 17.141415
```

```
Associated with: cm1
Actual context--> cpu: 78.77115, ram: 63.975082, memory: 15.567526, energy:
    100.0, index: 36.412167
```

Nel momento in cui il carico su tale nodo raggiunge valori troppo alti avverrà la migrazione. Infatti poco tempo dopo si ha una situazione del genere:

```
Sono su Container-1
Context[cm3]--> cpu: 10.354793, ram: 26.346369, memory: 2.7747908, energy:
    75.6174, index: 16.394249
Context[cm1]--> cpu: 77.09479, ram: 64.88432, memory: 87.20903, energy:
    100.0, index: 52.713272
Context[cm2]--> cpu: 5.725228, ram: 59.014675, memory: 7.089407, energy:
    92.148476, index: 18.876198
Associated with: cm1
Actual context--> cpu: 77.09479, ram: 64.88432, memory: 87.20903, energy:
    100.0, index: <52.713272>
Migration to cm3
Moving DSM to 3

Sono su Container-3
Context[cm1]--> cpu: 7.0192904, ram: 18.302485, memory: 86.83973, energy:
    100.0, index: 25.797146
Context[cm2]--> cpu: 25.42843, ram: 30.698774, memory: 4.063858, energy:
    91.34849, index: 16.4394
Context[cm3]--> cpu: 3.4131508, ram: 16.183126, memory: 7.618082, energy:
    74.61742, index: 13.874079
Associated with: cm3
Actual context--> cpu: 3.4131508, ram: 16.183126, memory: 7.618082, energy:
    74.61742, index: <13.874079>
```

l'indice sul nodo attuale ha raggiunto un valore di 57.71 a causa di un elevato uso delle risorse disponibili, in particolare della della memoria. Dalla figura 5.2 si può notare che il dsm e lo SC si sono spostati sul Container-3. Su quest'ultimo nodo l'indice è adesso di 21.1.

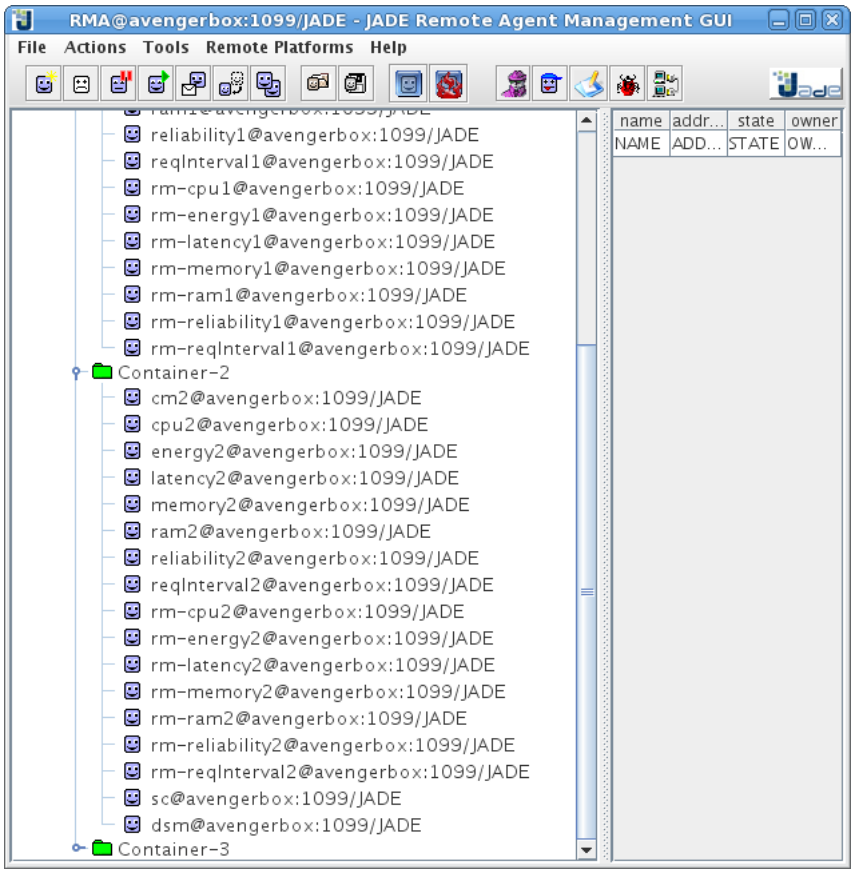

Figure 5.2: Migrazione su secondo nodo

Il nodo 3 è di tipo WIRELESS e quindi ha una energia limitata, il che implica un tempo di residenza per l'SC e il dsm molto più ridotto, in quanto l'indice raggiungerà rapidamente valori alti.

```
Sono su Container-3
Context[cm1]--> cpu: 0.81155604, ram: 33.999382, memory: 51.11227, energy:
    100.0, index: 19.762337
Context[cm2]--> cpu: 36.432972, ram: 58.424694, memory: 1.620206, energy:
    84.948586, index: 26.705338
Context[cm3]--> cpu: 67.67117, ram: 95.22946, memory: 26.253895, energy:
    49.21744, index: 58.74031
Associated with: cm3
Actual context--> cpu: 67.67117, ram: 95.22946, memory: 26.253895, energy:
    49.21744, index: <58.74031>
Migration to cm1
```

```
Moving DSM to 1

Sono su Container-1
Context[cm1]--> cpu: 50.261562, ram: 62.669903, memory: 48.456142, energy:
    100.0, index: 37.11915
Context[cm2]--> cpu: 1.1113803, ram: 6.8676476, memory: 18.00424, energy:
    83.9486, index: 10.791573
Context[cm3]--> cpu: 83.216156, ram: 71.41142, memory: 32.01974, energy:
    44.21744, index: 59.663654
Associated with: cm1
Actual context--> cpu: 50.261562, ram: 62.669903, memory: 48.456142, energy:
     100.0, index: <37.11915>
```

L'energia del nodo 3 è scesa da 49 a 44 il che ha costretto la migrazione sul nodo meno carico, ovvero l'uno (WIRED). In seguito vi è stata anche una violazione del contratto ed è stata segnalata ad entrambi i nodi partecipanti al contratto, i quali dovranno adattarsi per evitare di ripetere la violazione.

```
Violated with latency 23.106468 > 82.419365, reliability 41.978447 >
    99.44324, reqInterval 79.21008 > 76.23332
```

Dal log si può notare che è stato il subscriber ad eccedere in rischieste, infatti il reqInterval è inteso come frequenza.

5.2 Analisi dati

In questa sezione verranno esposti e commentati i grafici relativi alla simulazione, in quanto molto più esplicativi dei log riportati sopra. Tutti i grafici riportati avranno come ascisse i secondi che non rappresentato un valore reale ma simulato, infatti servono per dare un senso più reale alla simulazione.

5.2.1 Indice

L'indice è il valore più importante del test eseguito, in quanto la migrazione si basa su di esso. Come passo fondamentale vengono ora riportati i grafici relativi agli indici.

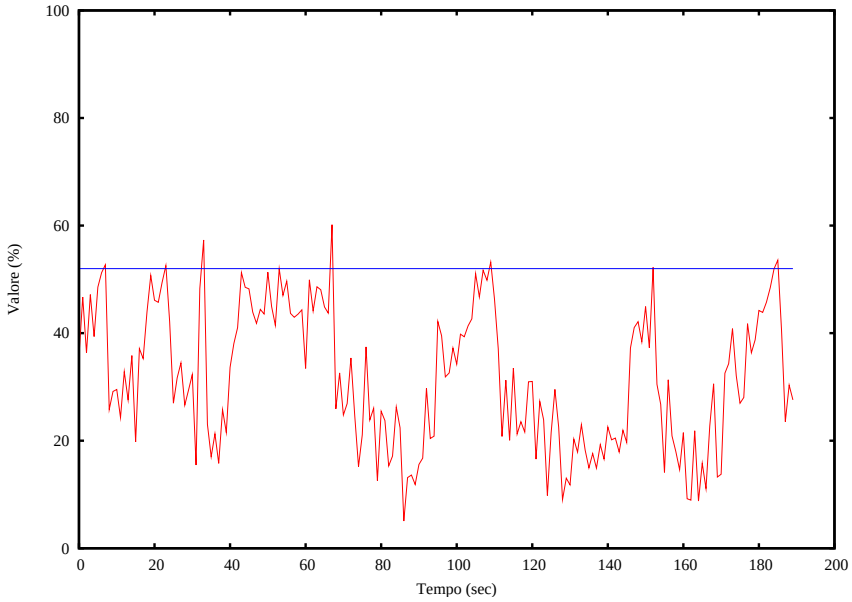

Figure 5.3: Indice primo nodo

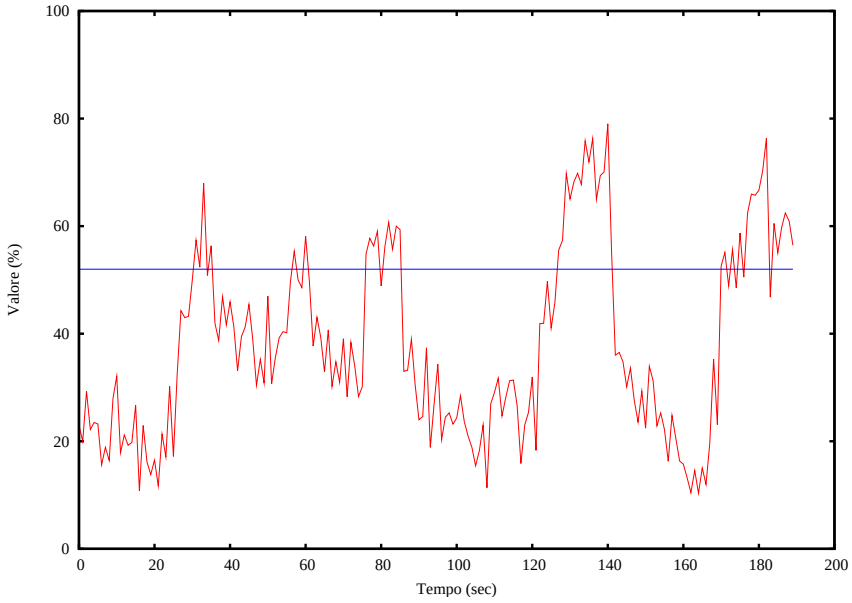

Figure 5.4: Indice secondo nodo

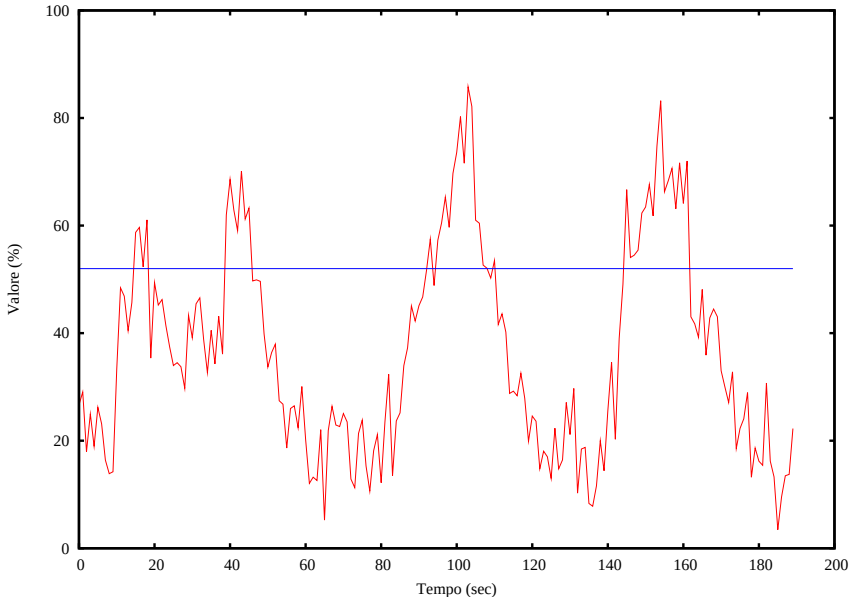

Figure 5.5: Indice terzo nodo

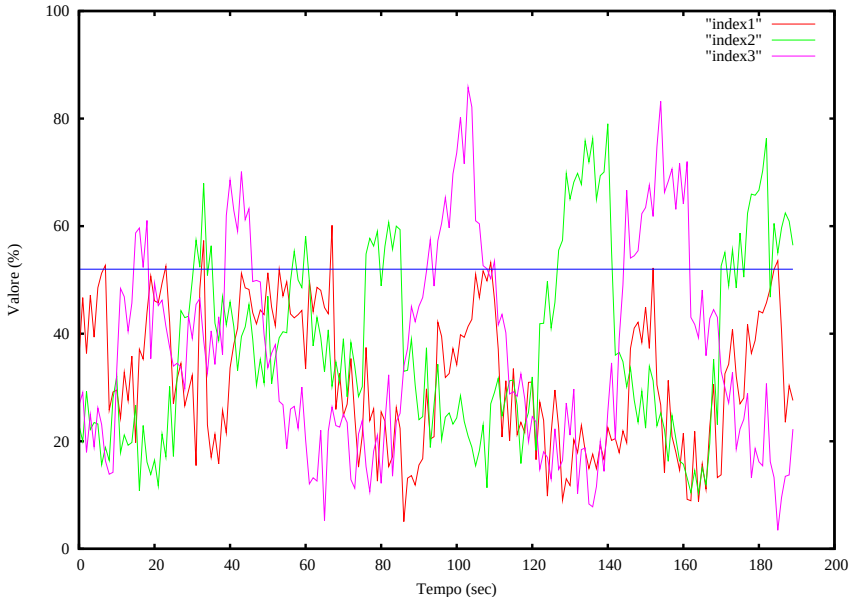

Figure 5.6: Indice tutti i nodi

Nelle figure 5.3, 5.4, 5.5 e 5.6 si può notare la presenza di una retta di colore blu, questa rappresenta il limite dell'indice, ovvero 52. Per comprendere meglio i grafici basta sapere che quando i punti superano la retta blu si ha una migrazione dal nodo corrispondente ad un altro. Ad esempio in figura 5.3 avviene la prima migrazione al secondo 8 circa.

5.2.2 Cpu

La cpu è molto sensibile alla migrazione, infatti quando lo SC si trova su un nodo si può notare un incremento del 40% sull'utilizzazione della cpu locale. Tale comportamento può essere facilmente estrapolato dai grafici riportati.

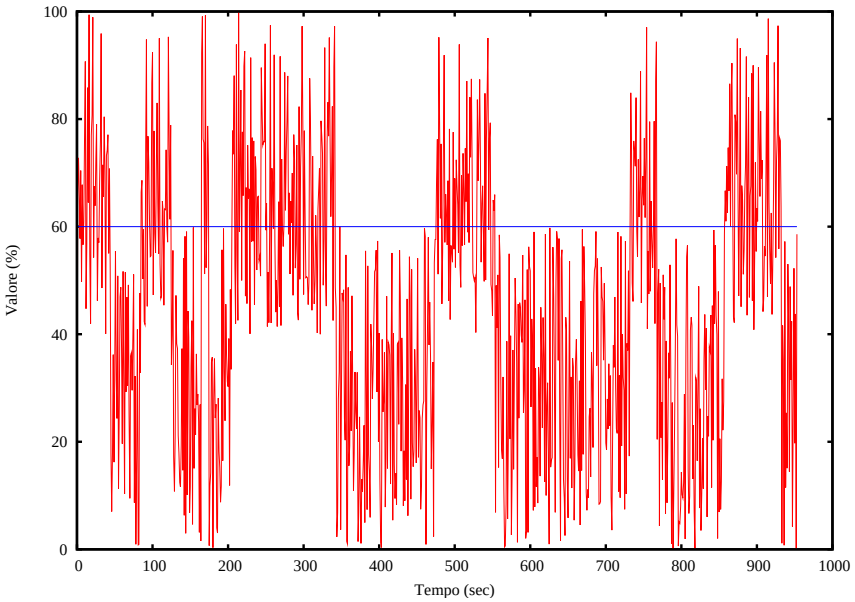

Figure 5.7: Cpu primo nodo

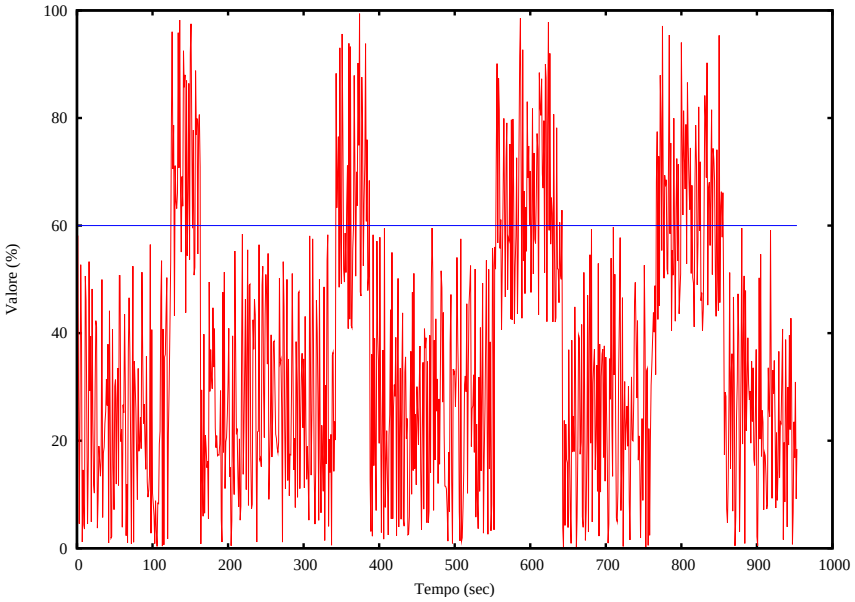

Figure 5.8: Cpu secondo nodo

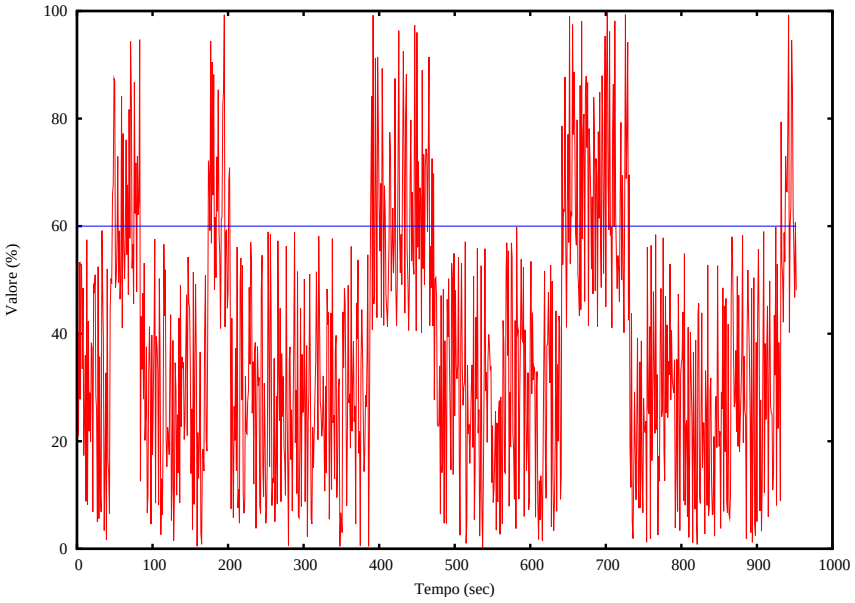

Figure 5.9: Cpu terzo nodo

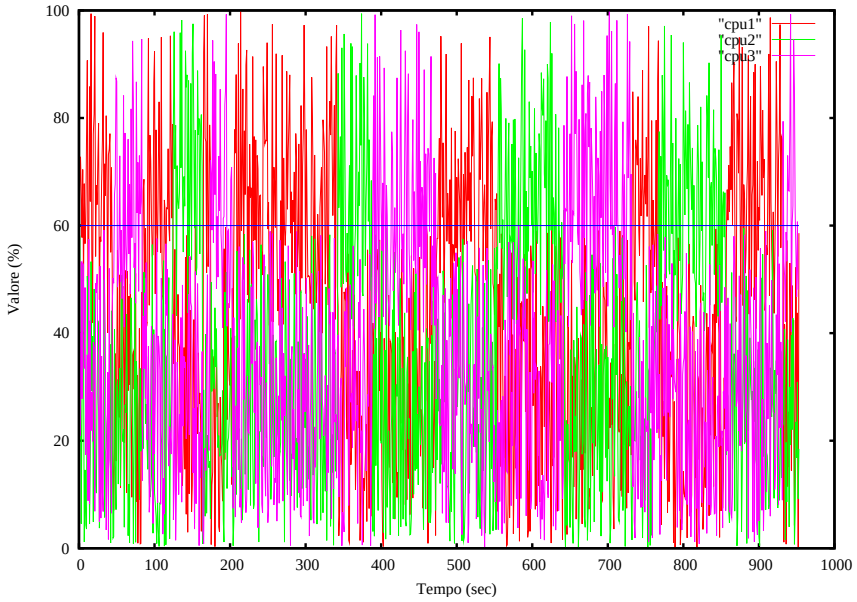

Figure 5.10: Cpu tutti i nodi

In tali grafici è presente una retta sul valore 60% per distinguere il i nodi che hanno lo SC dagli altri, in particolare quando i valori si trovano oltre 60% è presente lo SC sul nodo corrispondente. In presenza di una migrazione avviene un brusco cambiamento di carico, infatti si scende sotto il 60%. Nella figura 5.10 sono graficati i tre nodi, questo crea confusione ma rende più semplicemente l'idea di come migri lo SC da un nodo ad un altro. Nello specifico si può notare che nella parte superiore al 60% si trova sempre un nodo nel singono istante di tempo, questo è causato dalla presenza di un solo SC che si muove tra i nodi.

5.2.3 Ram

La ram ha una sensibilità come la cpu, infatti i valore vengono generati con le stesse formule. Dai grafici che seguono si possono notare i picchi di carico in corrispondenza delle migrazioni.

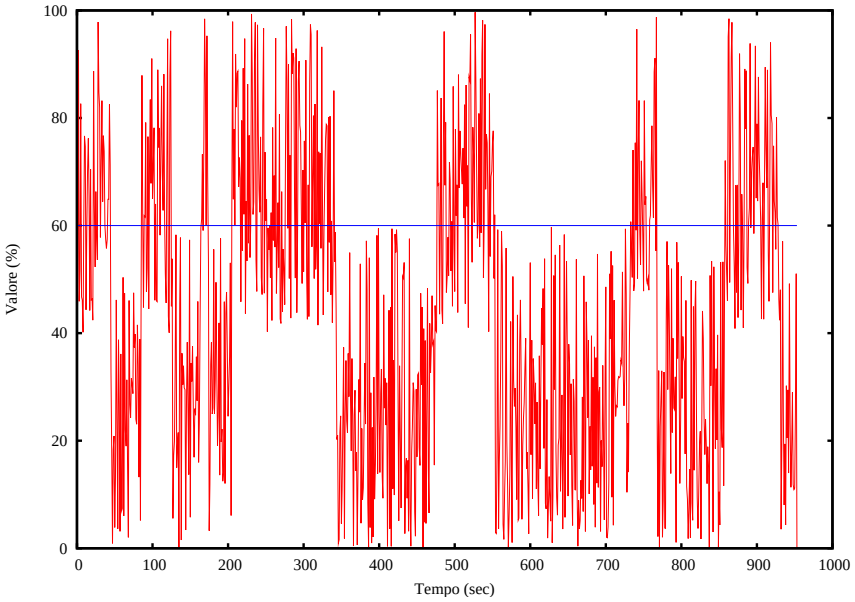

Figure 5.11: Ram primo nodo

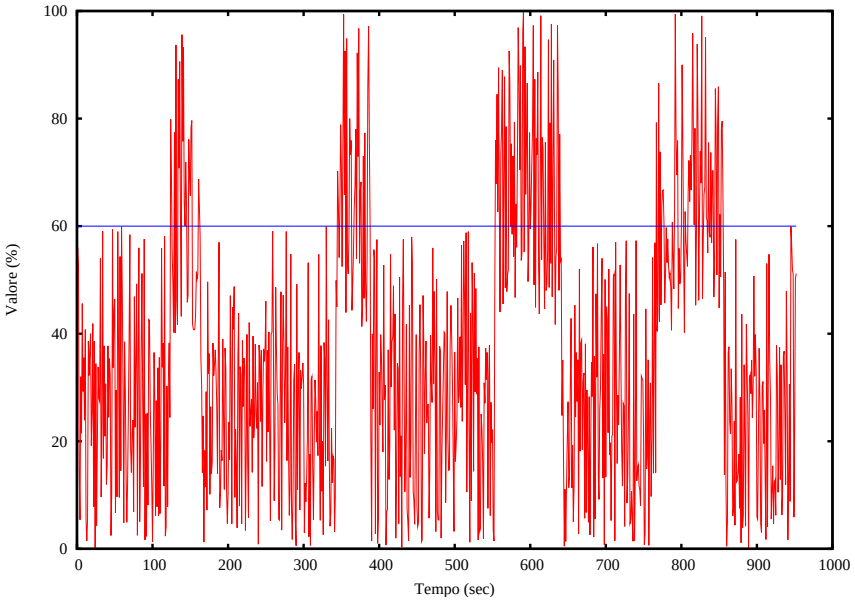

Figure 5.12: Ram secondo nodo

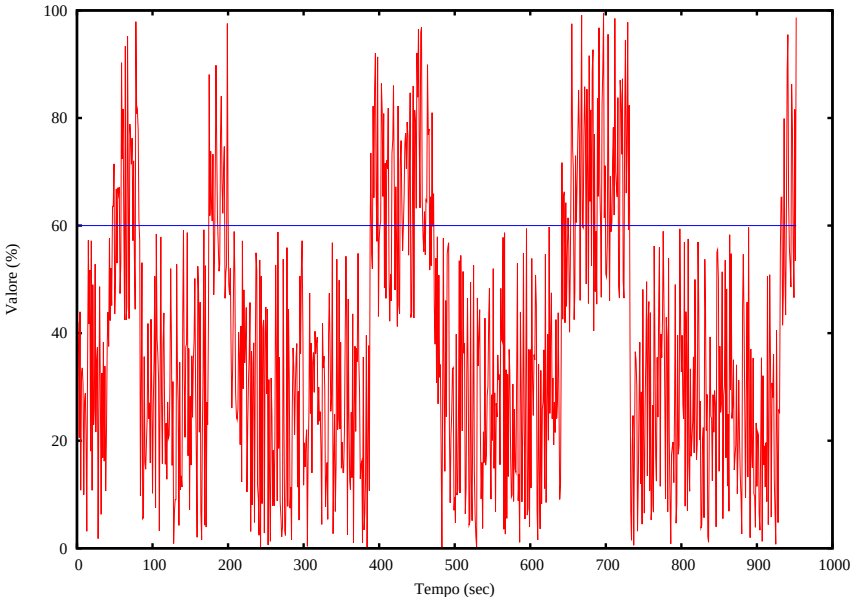

Figure 5.13: Ram terzo nodo

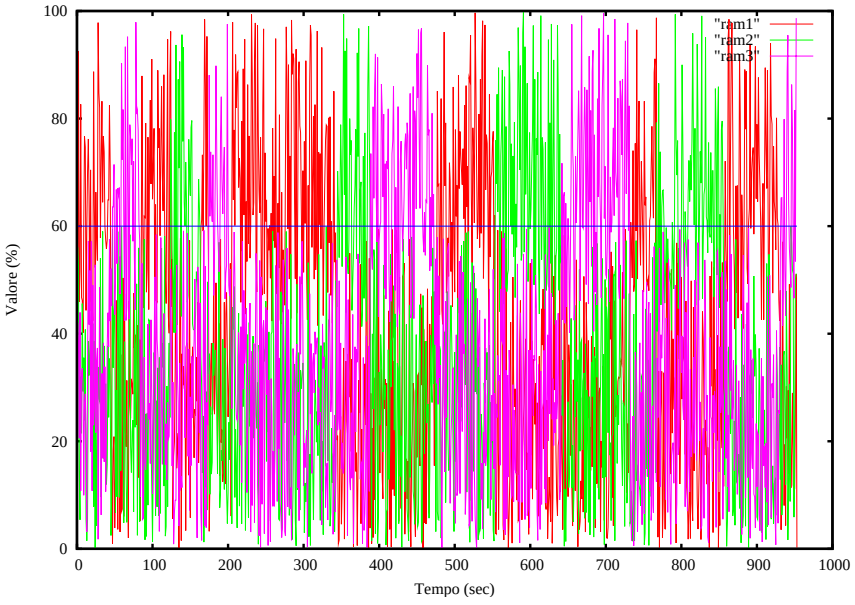

Figure 5.14: Ram tutti i nodi

5.2.4 Memory

La memoria è un parametro molto più dipendente dalle migrazioni, infatti in corrispondenza di esse si ha un incremento di utilizzazione costante ma non istantaneo. Dai grafici 5.15, 5.16, 5.17 e 5.18 si può notare che quando i valori di memoria raggiungono la soglia dello 0 o del 100 si ha un picco di traffico verticale, questo non è dovuto alle migrazioni ma potrebbe essere associato, ad esempio, a generazioni di file temporanei da parte del nodo. In corrispondenza di incrementi di memoria costanti vuol dire che è presente lo SC, invece quando si hanno cali costanti si ha il contrario, ovvero lo SC è migrato su un altro nodo.

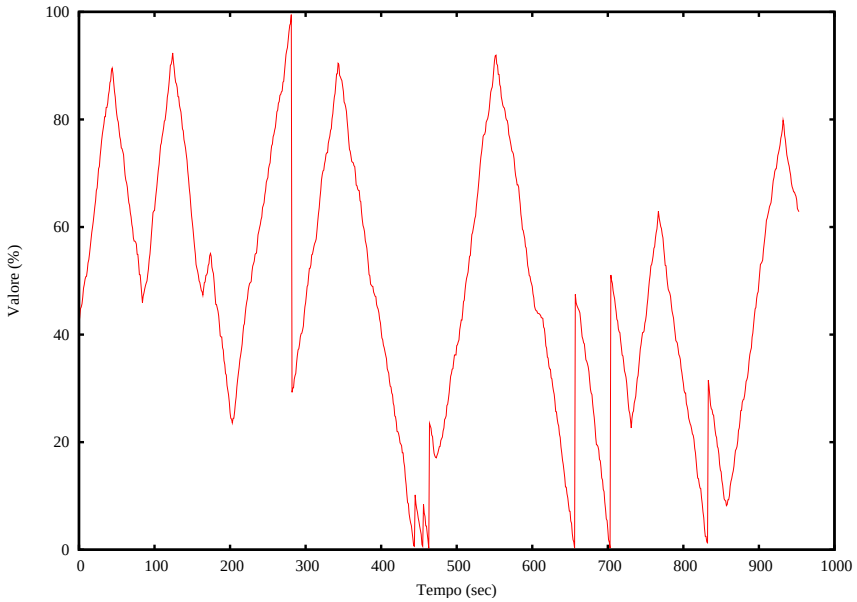

Figure 5.15: Memory primo nodo

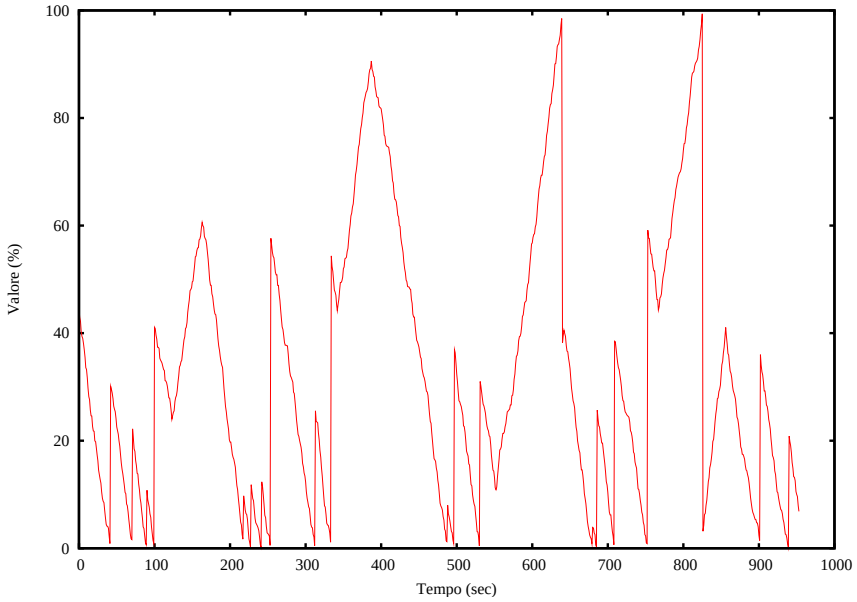

Figure 5.16: Memory secondo nodo

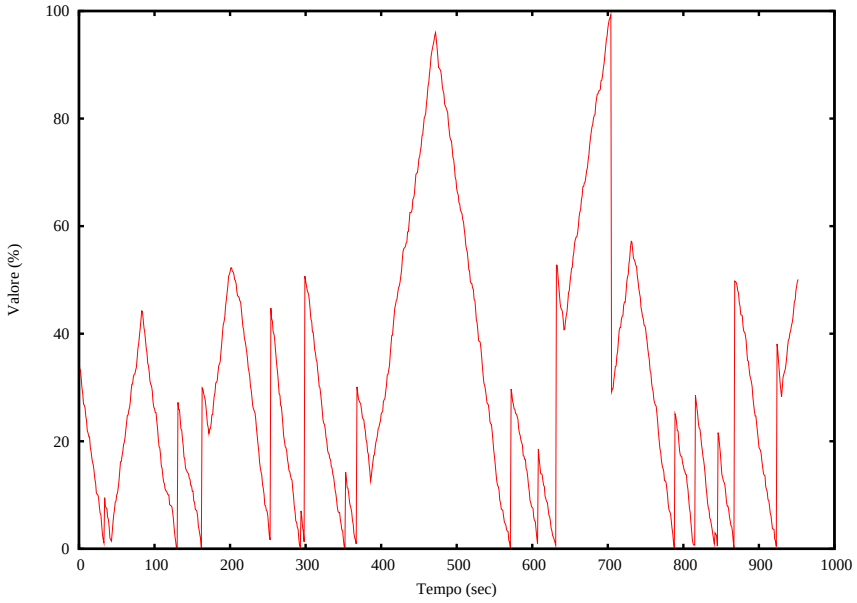

Figure 5.17: Memory terzo nodo

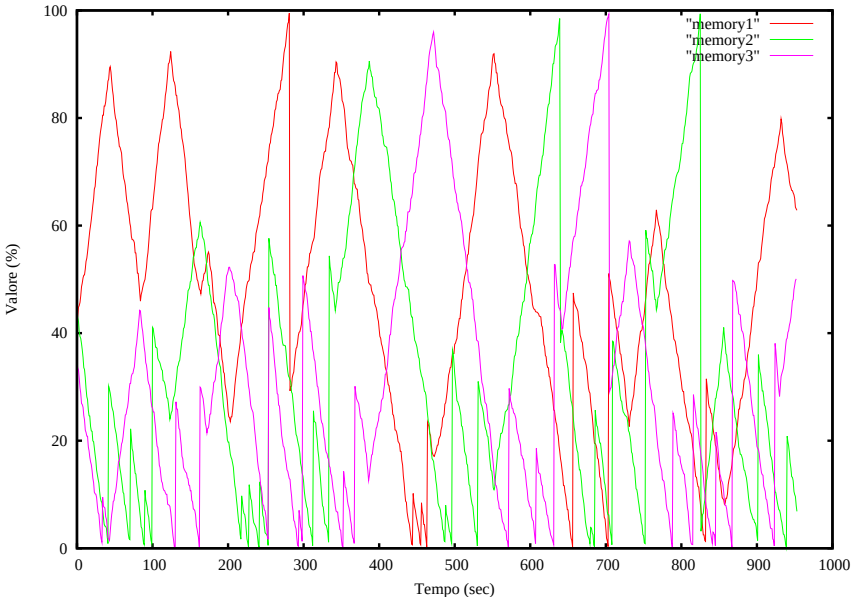

Figure 5.18: Memory tutti i nodi

5.2.5 Energy

L'energia è il parametro che varia più regolarmente in quanto dipende dal tipo di connessione e dalla presenza delle rete elettrica. Tale comportamento può essere notato dai grafici 5.19, 5.20, 5.21 e 5.22. In particolare in presenza dello SC su un nodo si ha un utilizzo di corrente maggiore, nello specifico se il nodo è connesso alla rete elettrica si ha un caricamento più lento, essendo parte dell'energia sprecata a causa dello SC. Anche in caso di scollegamento dalla rete elettrica si ha uno spreco di energia maggiore causato dallo SC, Dalle figure si può anche notare che in corrispondenza del 10% si ha un incremento di carica, essendo il nodo ricollegato alla rete elettrica, invece al raggiungimento del 100% viene scollegato, considerando che il caricamento è completo. Un ulteriore particolare si può notare in figura 5.19, in quanto il nodo è di tipo fisso e quindi con corrente sempre disponibile, infatti il valore di energia è sempre 100%.

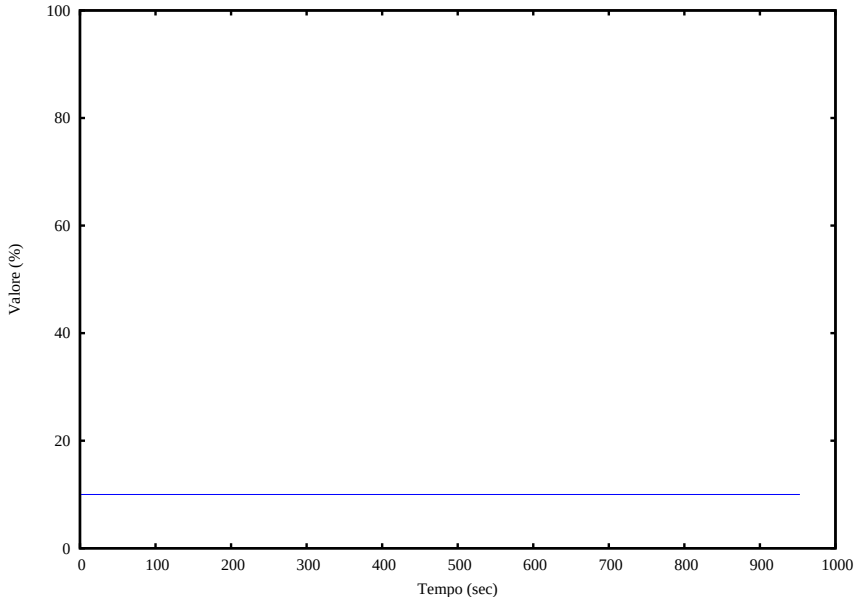

Figure 5.19: Energy primo nodo

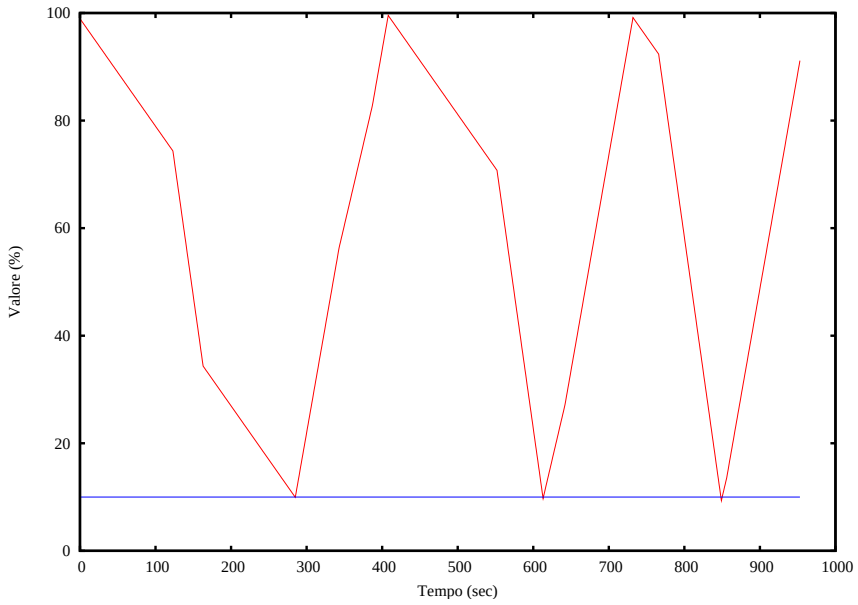

Figure 5.20: Energy secondo nodo

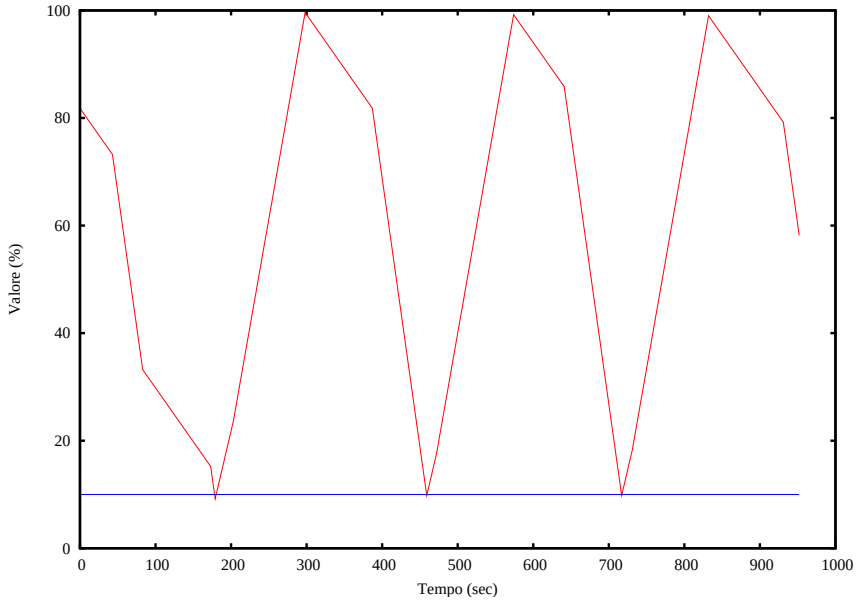

Figure 5.21: Energy terzo nodo

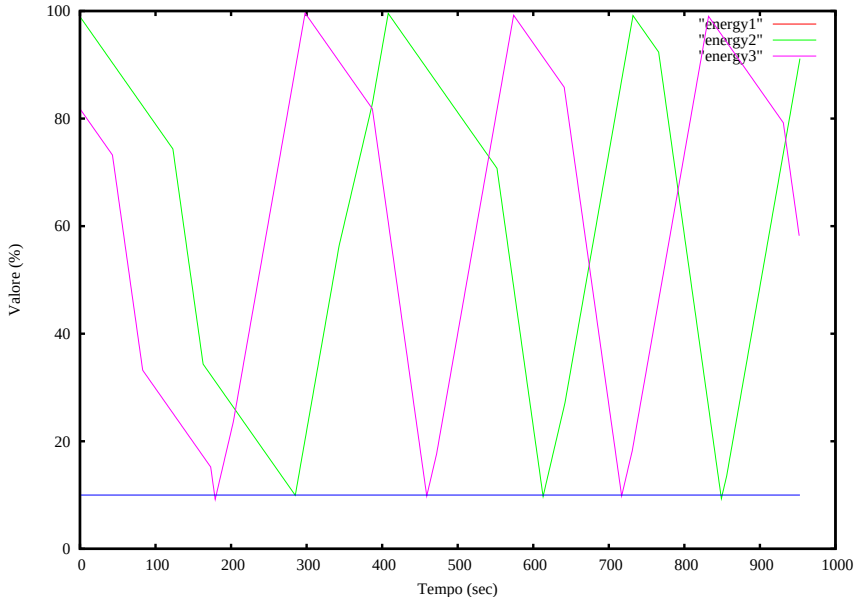

Figure 5.22: Energy tutti i nodi

5.2.6 Latency

Quest'ultimo valore generato dipendenze dalla presenza dell'SC, dal tipo di linea e dalla banda, come già riportato nel modello matematico. Nei seguenti grafici si possono vedere gli andamenti:

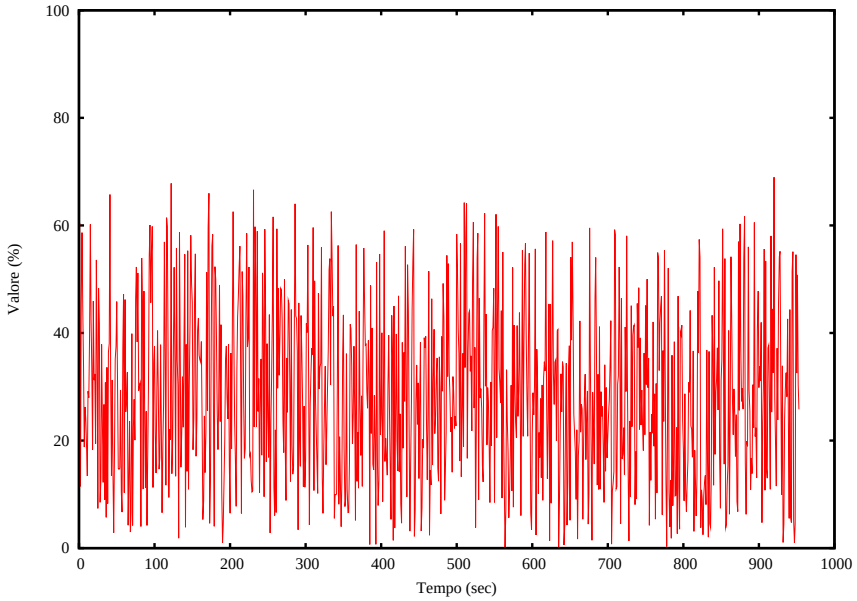

Figure 5.23: Latency primo nodo

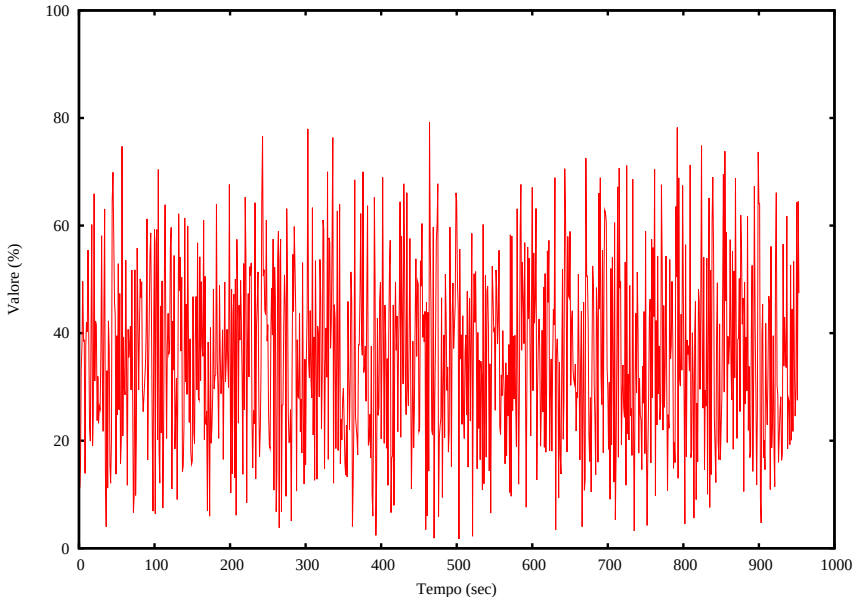

Figure 5.24: Latency secondo nodo

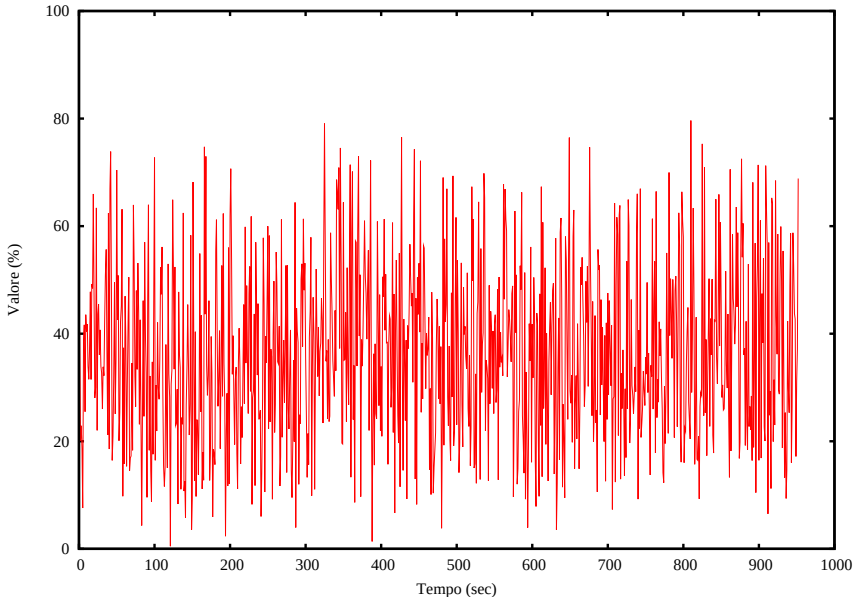

Figure 5.25: Latency terzo nodo

I valori in figura 5.23 è molto più basso rispetto ai valori in figura 5.24 e 5.25, a causa della tipologia di connessione diversa.

5.2.7 Reliability

Questo valore è simile alla latenza ed anch'esso dipende fortemente dal tipo di connessione e dalla banda ma non dall'SC, il quale non influisce sulle caratteristiche della qualità della connessione. Nelle figure si può notare la differenza di valori tra il caso 5.26 e i restanti.

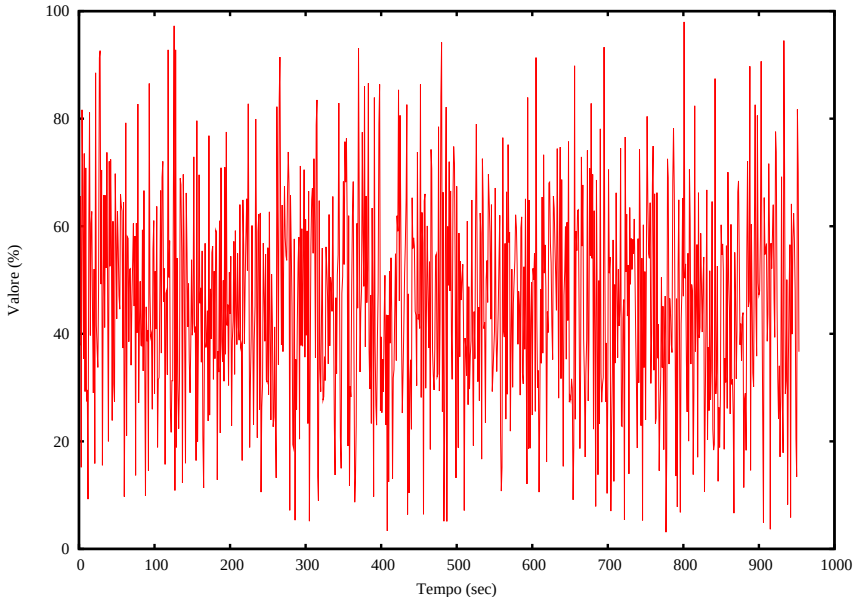

Figure 5.26: Reliability primo nodo

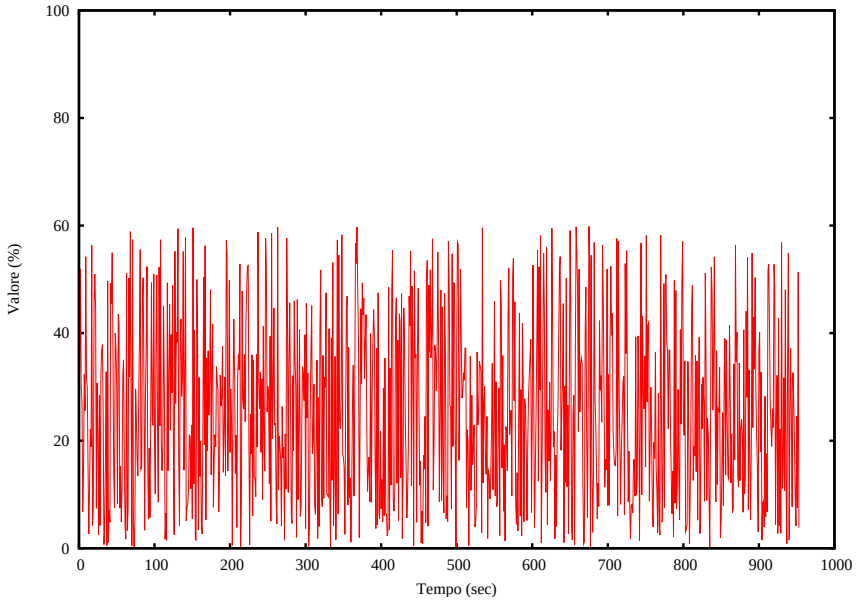

Figure 5.27: Reliability secondo nodo

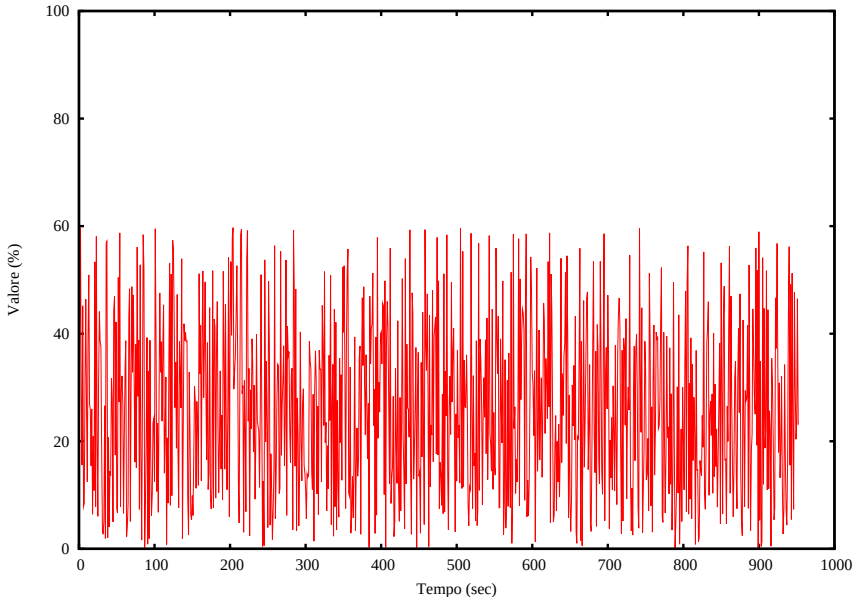

Figure 5.28: Reliability terzo nodo

5.2.8 ReqInterval

Quest'ultimo tipo di valore è strettamente casuale e può assumere valori tra lo 0% e il 100%.

List of Tables

List of Figures

Index

Bibliography

Jing-Helal-Elmagarmid, 'Client Server Computing in Mobile Environments', (1999).