ALMA MATER STUDIORUM
UNIVERSITÀ DEGLI STUDI DI BOLOGNA

II facoltà di Ingegneria - Cesena
Ingegneria Informatica

Traffic Simulator

Progetto di Reti di Telecomunicazioni L-S

Michael Gattavecchia - *0000362269*
Marco Santarelli - *0000366521*
Andrea Zagnoli - *0000367565*

ing. Walter Cerroni

ANNO ACCADEMICO 2009/10

Indice

Introduzione

L'esposizione delle attività svolte è stata organizzata, come suggerito all'interno del documento di rifermento, in cinque macro-sezioni all'interno delle quali sono poi stati sviluppati gli argomenti specifici. In particolare, dopo la trattazione dei generatori di numeri pseudo-casuali analizzati si passerà allo studio delle varie tipologie di simulazioni svolte, concludendo con l'esposizione delle attività opzionali che sono state scelte.

Struttura del report

L'implementazione del simulatore rende necessaria l'esposizione del lavoro svolto secondo un duplice punto di vista: il primo relativo alla valutazione quantitativa e qualitativa degli esiti delle simulazioni svolte, il secondo (non meno importante) relativo all'infrastruttura tecnica implementata a garanzia della validità dei dati presentati. Per questo motivo, all'interno di ciascuna sezione delineata nella struttura del report sopra indicata, si è cercato di fornire una trattazione esaustiva di entrambi gli aspetti. In particolare, in seguito all'esposizione delle simulazioni svolte, sarà posta una sezione di *analisi tecnica* all'interno della quale verranno esposte le particolarità implementative dei componenti di volta in volta coinvolti.

La piattaforma di riferimento per lo sviluppo del simulatore è *Java*, attraverso la quale sono stati sviluppati sia il simulatore in sé, sia la parte relativa alla costruzine dei grafici relativi alle simulazioni svolte.

Gruppo di lavoro

Il gruppo di lavoro è costituito da:

- Michael Gattavecchia - *0000362269*
- Marco Santarelli - *0000366521*
- Andrea Zagnoli - *0000367565*

Capitolo 1

Generazione di numeri pseudo-casuali

Alla base di una qualsivolglia simulazione risiede la centralità del ruolo rivestito dalla riproduzione dell'aleatorietà riscontrabile all'interno dei sistemi reali. Per questo motivo è fondamentale valutare e conoscere le entità alle quali si fa riferimento per la generazione di numeri casuali, il cui comportamento inciderà in maniera rilevante sulla qualità delle simulazioni.

Per la generazione di numeri pseudo-casuali sono stati considerati, in fase di analisi, quattro metodologie alternative, le cui caratteristiche sono riportate all'interno dell'apposita scheda presente nell'interfaccia grafica del simulatore:

- `java.util.Random`
  generatore predefinito dell'environment Java per la generazione di numeri pseudo-casuali.
- `ran0`
  generatore lineare congruenziale di tipo *ran0*, implementato ex-novo.
- `SecureRandom`
  generatore presente all'interno del package `java.security` in grado di generare sequenze pseudo-casuali crittograficamente robuste.
- `MersenneTwister`
  generatore implementato all'interno del progetto Apache che realizza l'algoritmo Matsumoto-Nishimura.

1.1 Generatori lineari congruenziali

I generatori lineari congruenziali rappresentano una tra le più conosciute e sperimentate alternative per la generazione di numeri pseudo-casuali.

Il meccanismo di generazione è definito, ricorsivamente, come:

$$Z_i = (a \cdot Z_{i-1} + c) \mod m$$

dove:
Z_i rappresenta l'i-esima istanza generata
m $(0 < m)$, il modulo
a $(0 < a < m)$, il "moltiplicatore"
c $(0 \leq c < m)$, l'incremento
Z_0 il seme della sequenza (seed)

Il periodo di un generatore lineare congruenziale è al più pari a m[1], a patto di rispettare alcuni vincoli:

- c ed m siano coprimi tra loro
- $(a - 1)$ sia divisibile per tutti i fattori primi di m
- $(a - 1)$ sia multiplo di 4, qualora anche m lo sia

Le modalità secondo le quali questo tipo di generatori sono definiti fanno sì che essi siano particolarmente apprezzati per la loro leggerezza, vista la necessità di memorizzare le sole variabili di stato per un corretto funzionamento e la semplicità dei calcoli necessari per la generazione. Moltiplicazioni tra interi ed il calcolo del modulo sono infatti operazioni che un moderno microprocessore può realizzare in maniera molto performante. Tuttavia esiste correlazione tra i valori generati successivamente, che determina l'impossibilità di utilizzare questo tipo di generatori in ambienti in cui tale correlazione rappresenti un fattore di rischio, come ad esempio in ambito crittografico.

1.1.1 Classe `Random`

La classe `Random`, presente all'interno del package `java.util`, rappresenta lo standard per la generazione di valori pseudo-casuali all'interno dell'environment Java.

Si tratta di un LCG caratterizzato dai parametri mostrati in tab.1.1 e dotato quindi di un periodo pari a 2^{48} $(281,474,976,710,656)$. Un'istanza della classe `Random` permette di "prelevare" valori pseudo-casuali di diverso tipo a seconda delle necessità, mediante l'invocazione parametrizzata del metodo protetto `next()`.

[1] in questo caso il periodo del generatore è detto "pieno".

parametro	valore
m	2^{48}
a	25214903917
c	11

Tabella 1.1: Parametri del LCG implementato in `java.util.Random`

Il seme della sequenza pseudo-casuale, formato da 48 bit, può essere specificato in modo esplicito all'atto della costruzione (`Random(long seed)`), o acquisito tramite l'invocazione interna del metodo `System.currentTimeMillis()` nel caso in cui si ricorra al costruttore standard.

1.1.2 Generatore `ran0`

Il generatore realizzato ex-novo per il simulatore è di tipo lineare congruenziale (*ran0*). Come possibile osservare in tabella 1.2, il parametro c è stato posto uguale a 0: questo caratterizza il generatore come di tipo "Park-Miller" (moltiplicativo).

Analogamente a quanto visto nel caso del LCG standard di Java, il generatore può o meno essere instanziato unitamente ad un seed di partenza. In caso tale parametro venga omesso si ricorrerà nuovamente alla chiamata `System.currentTimeMillis()` per l'ottenimento di un valore arbitrario da impostare come seed iniziale.

parametro	valore
m	2147483647
a	16807
c	0

Tabella 1.2: Parametri del LCG implementato ex-novo (`RandomProvider`)

1.1.3 SecureRandom

La classe `SecureRandom` presente all'interno del package `java.security` estende la classe `Random` discussa in precedenza, e permette la generazione robusta dal punto di vista crittografico di numeri pseudo-casuali attraverso l'agoritmo[2] e l'eventuale provider specificati nella chiamata di instanziamento.

[2] e.g. ``SHA1PRNG'' che segue le specifiche dello standard IEEE P1363

Il seed verrà poi automaticamente generato attraverso l'invocazione automatica del metodo `setSeed()`.

1.1.4 MersenneTwister

La classe `MersenneTwister`, sviluppata dal progetto *Apache* e disponibile all'interno del package `commons.math.random.MersenneTwister`, implementa l'algoritmo ideato da Makoto Matsumoto e Takuji Nishimura nel 1996/97.

Questo generatore è caratterizzato da un periodo parcolarmente lungo (pari a $2^{19937} - 1$), da una equidistribuzione a 623 dimensioni e da una accuratezza massima pari a 32 bit.

L'algoritmo è basato sulla seguente ricorsione lineare:

$$x_{k+n} := x_{k+m} \oplus (x_k^u | x_{k+1}^l)A, (k = 0, 1, ...)$$

per il cui approfondimento si rimanda alla bibliografia [3].

1.1.5 Analisi tecnica: classe `RandomProvider`

All'interno della classe `RandomProvider`, oltre al generatore creato ex-novo, vengono gestite tutte le tipologie di generatori disponibili, elencate all'interno dell'`enum Provider`. In particolare all'interno della chiamata per l'instanziamento è possibile specificare la tipologia di generatore che si intende utilizzare, permettendo una fruizione del servizio disaccoppiata dalla specifica implementazione scelta.

1.2 Analisi del generatore

Il generatore che si è scelto di utilizzare come default per il simulatore è quello fornito dalla classe `Random` di Java. Tale scelta è stata dettata dalla estrema efficienza computazionale di tale generatore e dalla trascurabile rievanza degli effetti della correlazione per l'applicazione di interesse.

1.2.1 Distribuzione uniforme in [0,1]

Per la valutazione dell'uniformità della distribuzione dei valori generati si è deciso di realizzare uno scatter plot. Per la realizzazione di questo grafico è necessaria la generazione di N coppie di numeri pseudo-casuali, i quali devono essere generati in modo consecutivo, senza variare il seme dell'algoritmo di generazione. Le coppie generate sono quindi utilizzate come le coordinate X e Y dei punti da mostrare nel grafico. Nel caso più generale è possibile specificare

un numero variabile di generazioni che devono essere effettuate per ogni coppia tra la coordinata dell'ascissa e dell'ordinata: nel nostro caso si è scelto di usare la generazione immediatamente successiva a quella dell'ascissa. Questo grafico permette di individuare correlazioni evidenti tra i valori o problemi dovuti a parametri critici dell'algoritmo, che si presentano come disposizioni ordinate dei punti del grafico. In figura 1.1 si mostra la distribuzione dei valori generati da un'istanza della classe Random: è possibile apprezzare il buon livello di dispersione e la discreta uniformità dei punti nella copertura dello spazio. La sezione del simulatore dedicata all'analisi di generatori di numeri random permette di creare dinamicamente scatter plot per ciascuna delle 4 tipologie di generatori discussi in precedenza.

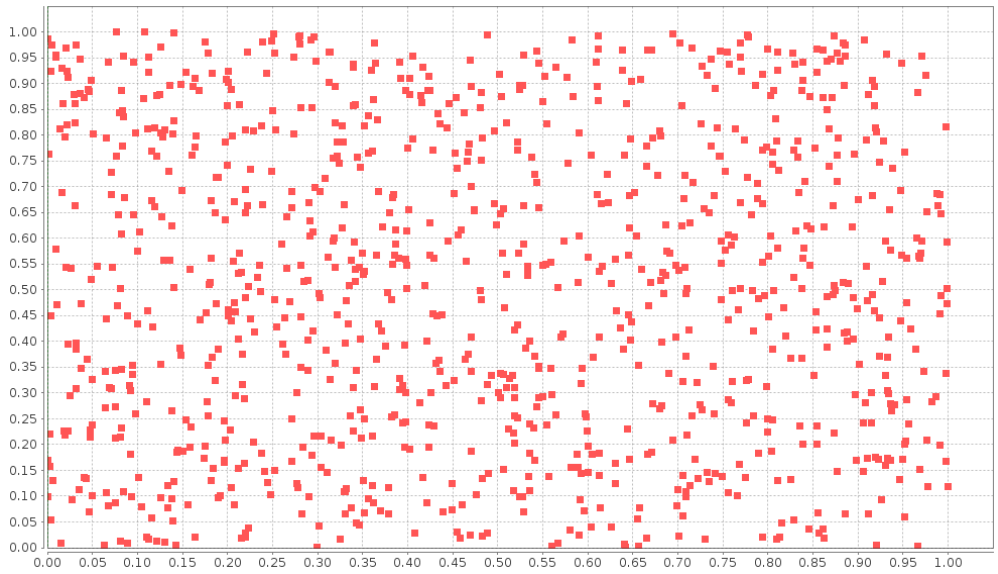

Figura 1.1: Distribuzione dei valori generati da `util.Random`

1.2.2 Stima del valore medio

Al fine di permettere un'agevole valutazione visiva del valor medio delle sequenze di numeri pseudo-casuali prodotte si è ricorsi ad una serie di generazioni con campioni di dimensione crescente.

In figura 1.2 è possibile osservare come, all'aumentare della dimensione dei campioni, si abbia una convergenza al valore teorico di 0.5. In particolare, già a partire da qualche migliaio di esperimenti, si può osservare un'importante grado di aderenza al valore teorico.

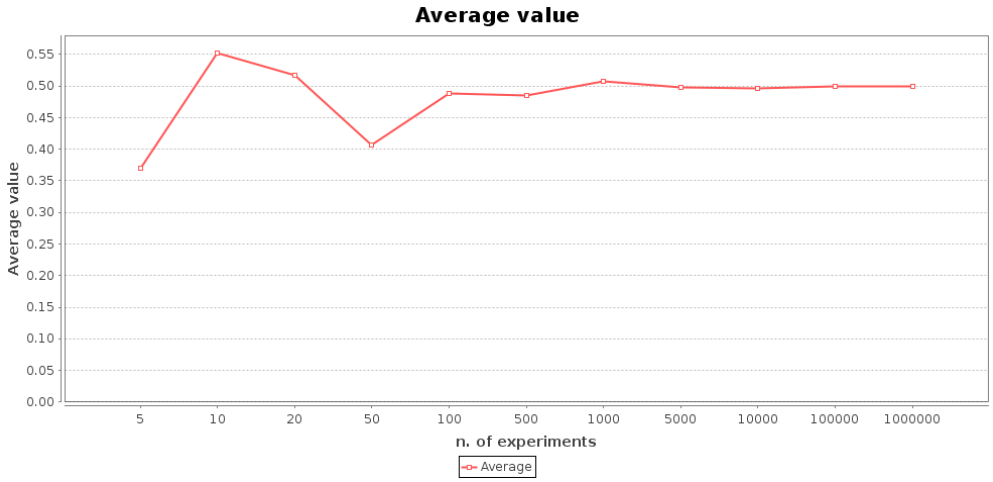

Figura 1.2: Media dei campioni in funzione della numerosità

L'andamento della media campionaria presenta sempre ad ogni ripetizione dell'esperienza la stessa tendenza a convergere al valore teorico atteso, con oscillazioni rilevanti presenti esclusivamente in concomitanza ai campioni di piccole dimensioni, come era lecito attendersi.

Anche per l'analisi del valor medio dei numeri pseudo-casuali generati è possibile selezionare le varie tipologie di generatori messe a disposizione, ottenendo di volta in volta il relativo grafico.

1.2.3 Intervallo di confidenza

Per la valutazione degli intervalli di confidenza sono state effettuati esperimenti sia al variare del livello di confidenza, sia al variare del numero di campioni. Come era lecito aspettarsi, l'intervallo di confidenza mostra un andamento decrescente all'aumentare del numero di campioni presi in considerazione, mentre ha un'andamento crescente all'aumentare del livello di confidenza richiesto. I grafici ci hanno permesso inoltre di confrontare quantitativamente i valori calcolati dalle funzionalità da noi realizzate con i valori teorici, confermandoci la loro corretta implementazione

In fig.1.3 e 1.4 è possibile osservare l'andamento dell'IDC in funzione del numero di esperimenti svolti e del livello di confidenza, rispettivamente.

1.2.4 Analisi tecnica

L'esecuzione effettiva delle due possibili tipologie di simulazione per il calcolo dell'intervallo di confidenza (in funzione del numero di run o del livello di

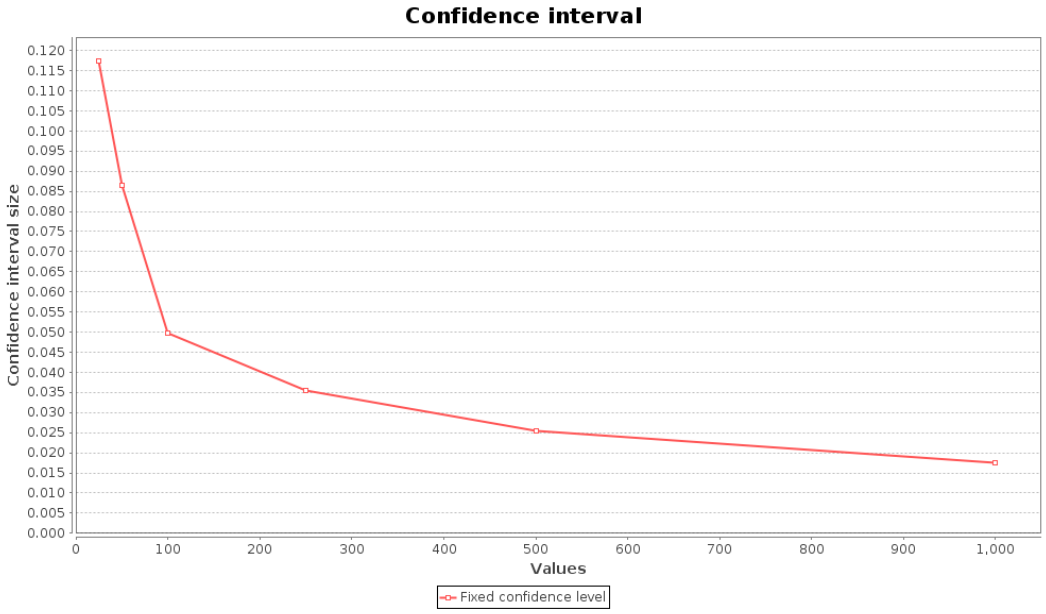

Figura 1.3: Intervallo di confidenza in funzione del numero di campioni

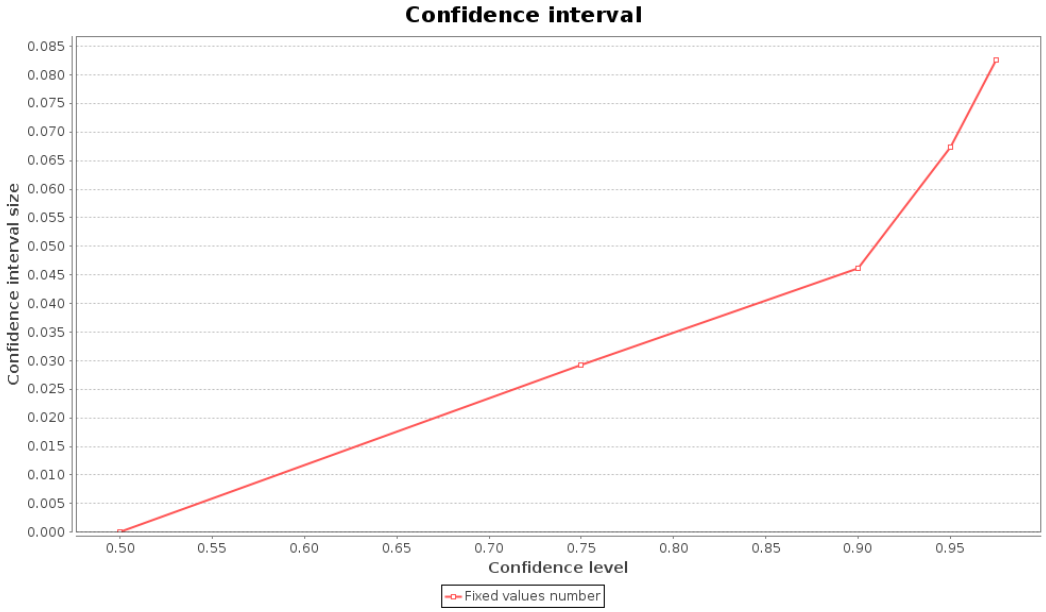

Figura 1.4: Intervallo di confidenza in funzione del livello di confidenza

confidenza) è opera, rispettivamente, dei metodi:

- `testConfidenceIntervalWithVariableConfidence(...)`
- `testConfidenceIntervalWithVariableRuns(...)`

della classe `SimulationRunners`. All'interno di quest'ultima sono stati infatti raccolti, al fine di ottenere una struttura topologicamente ben organizzata, i metodi corrispondenti alle varie tipologie di esperimenti messi a disposizione dal simulatore implementato.

All'utente viene data la possibilità di scegliere un generatore di numeri pseudo-casuali tra i quattro esposti al capitolo 1, e di specificare la tipologia di simulazione ed il relativo parametro di riferimento.

Una volta terminata la parte computazionale, attraverso un apposito metodo della classe `gui.GraphUtils`, viene generato e visualizzato un grafico riportante gli esiti della simulazione.

La classe `GraphUtils` verrà in seguito utilizzata per la creazione della totalità dei grafici generati a seguito delle simulazioni, vista la volontà di mantenere una strutturazione ben organizzata di classi e package all'interno del progetto.

1.2.5 Realizzazione dei grafici

Per la realizzazione dei grafici relativi alle simulazioni si è ricorsi all'utilizzo del tool gratuito *JFreeChart*[3], che permette la realizzazione di grafici all'interno dell'invironment Java, oltre a permettere l'esportazione degli stessi in diversi formati di immagine (png). JFreeChart mette a disposizione numerose tipologie di grafici e diagrammi, consentendo una gamma di personalizzazioni pressoché illimitata. Si è deciso di appoggiarsi a questa libreria poiché, implementando il simulatore in Java, si ha l'indubbio vantaggio di poter mostrare i grafici direttamente da codice, senza bisogno di utilizzare tool esterni e senza dover estrarre ed esportare i dati dal simulatore. Inoltre, il framework che la libreria mette a disposizione consente di navigare, ridimensionare, zoomare ed editare i chart tramite interfaccia grafica, consentendoci di ottimizzare la presentazione dei dati in maniera più rapida.

[3] http://www.jfree.org/jfreechart/

Capitolo 2

Generazione di traffico con diversa variabilità

2.1 Distribuzioni di traffico considerate

Le distribuzioni secondo le quali il traffico si presenta all'ingresso del sistema influenzano in modo marcato le dinamiche che a regime vengono a determinarsi all'interno dello stesso. Per questo motivo è di grande importanza osservare la risposta specifica del sistema ad input differenziati, al fine di valutarne le caratteristiche e di meglio comprendere le specifiche peculiarità delle varie distribuzioni in analisi.

Le tipologie di distribuzione prese in considerazione sono, come da specifica:

- Deterministica
- Poissoniana (Exponential)
- Switched Poisson Process (SPP)
- Pareto

Il simulatore permette, in funzione del tipo di distribuzione selezionata e dei parametri imposti, di valutare valore medio, varianza ed indice di dispersione relativi al traffico generato.

Al termine della computazione, i risultati numerici vengono riportati sulla console appositamente realizzata (fig. 2.1).

2.1.1 Analisi tecnica

L'esecuzione della simulazione avviene, analogamente a quanto visto nel capitolo precedente, facendo ricorso alla classe `SimulationRunners`. L'organizzazione gerarchica delle distribuzioni, ottenuta mediante varie specializzazioni

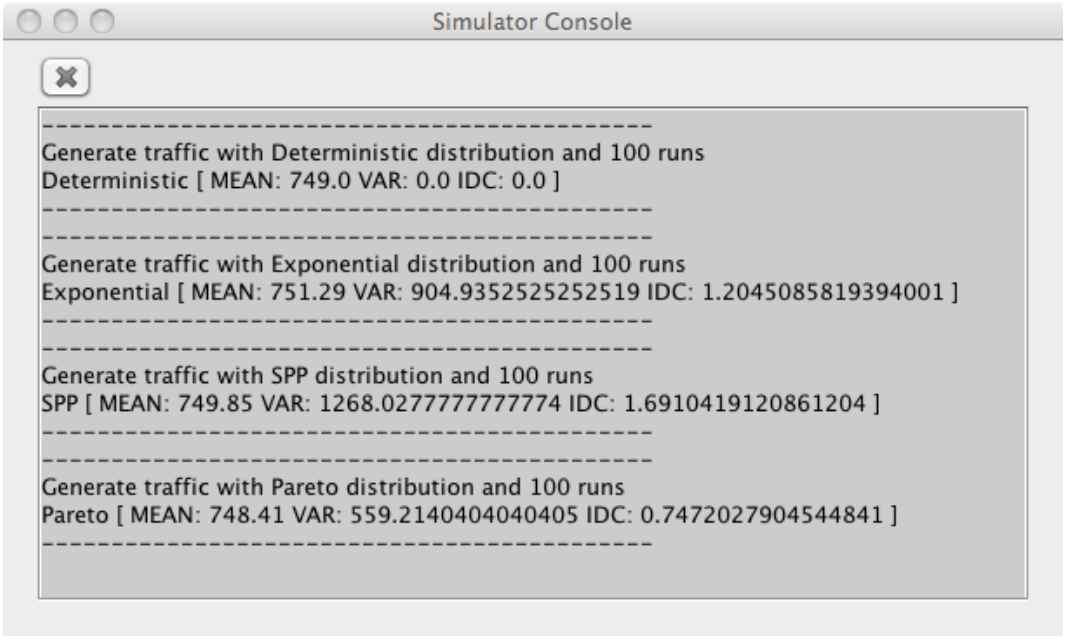

Figura 2.1: Esito della simulazione di varie tipologie di traffico (console)

della classe `Distribution` rende possibile una gestione trasparente rispetto alla scelta effettuata dall'utente.

Per la gestione della console di sistema, all'interno della quale vengono presentati i risultati delle simulazioni, si è ricorsi alla libreria LOG4J[1] In particolare si è realizzata una console su finestra indipendente la cui visualizzazione può essere (dis)abilitata dall'utente, senza che i dati in essa contenuti vadano persi.

La generazione di valori da parte delle classi che implementano la classe astratta `Distribution` (si veda lo schema in fig 2.2) avviene attraverso il metodo `nextDouble()`, in particolare ogni estensione della classe implementa una spefica funzione in base al tipo di distribuzione rappresentato.

Infine, è interessante notare come attraverso il ricorso alla classe `ExecutorService` l'onere computazionale sia stato scisso dalla gestione dell'interfaccia grafica, permettendo così un'esperienza d'utilizzo esente da blocchi dovuti alla computazione.

[1]LOG4J - http://logging.apache.org/log4j/1.2/

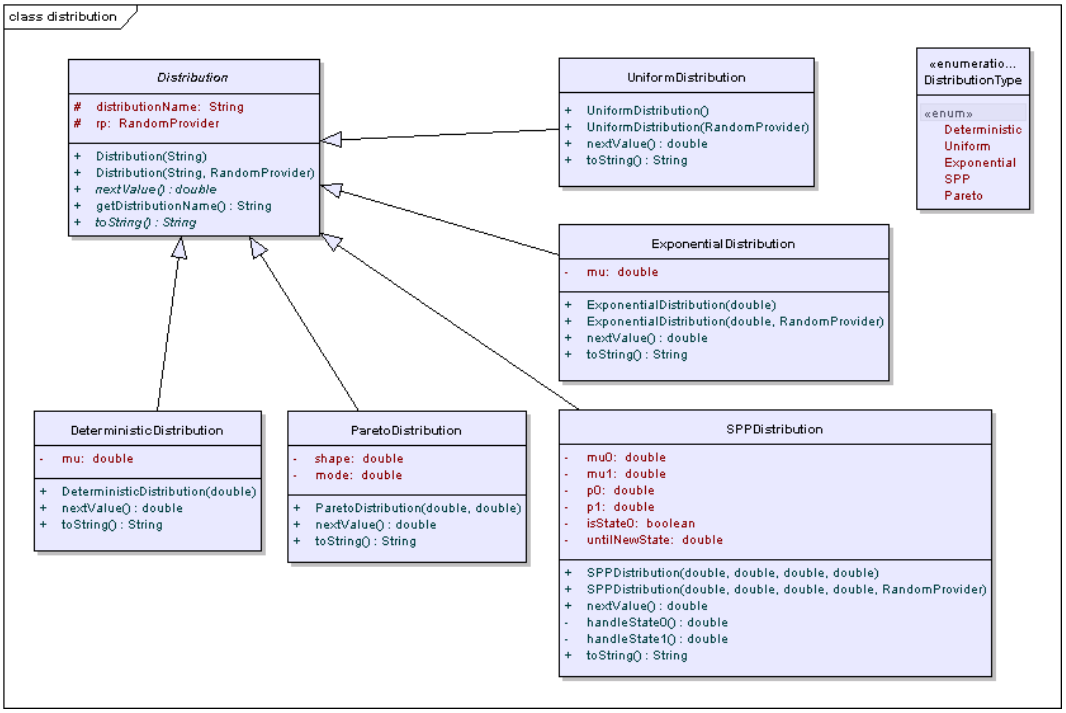

Figura 2.2: Struttura del package `simulator.distribution`

Capitolo 3

Simulazione di una coda $M/G/1$

3.1 Sistema a coda $M/G/1$

Il sistema considerato in questo capitolo è di tipo a singolo servitore, con arrivi di tipo poissoniano, tempi di servizio variabilmente distribuiti e media μ definiti dall'utente in fase di inizializzazione.

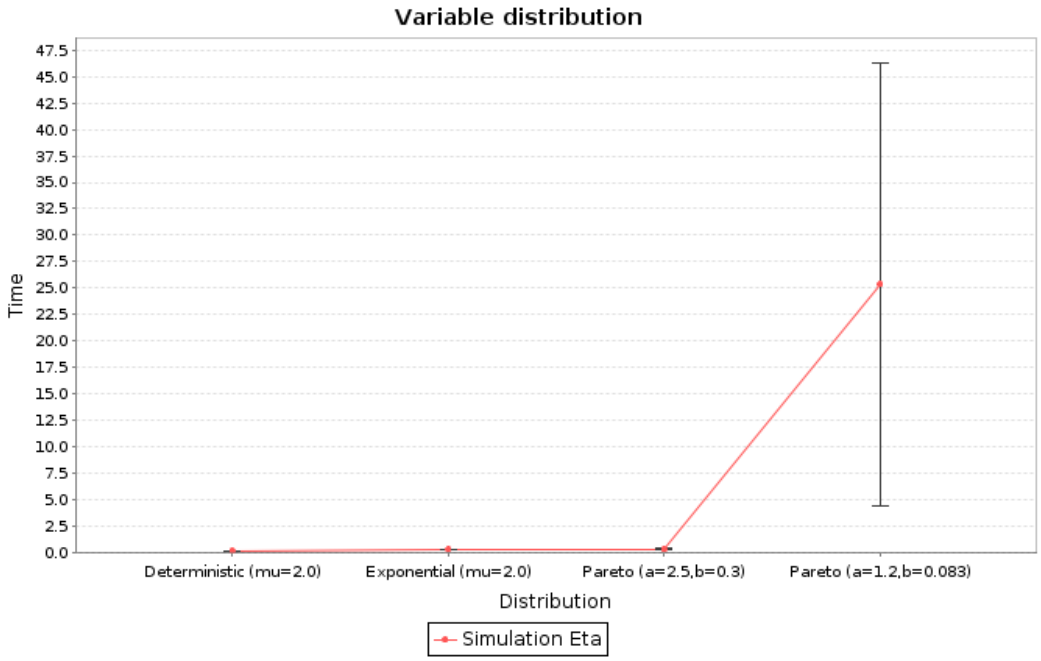

Figura 3.1: Tempi medi di attesa al variare del tipo di distribuzione $\rho = 0.4$, $\mu = 2.0$, $N = 100$

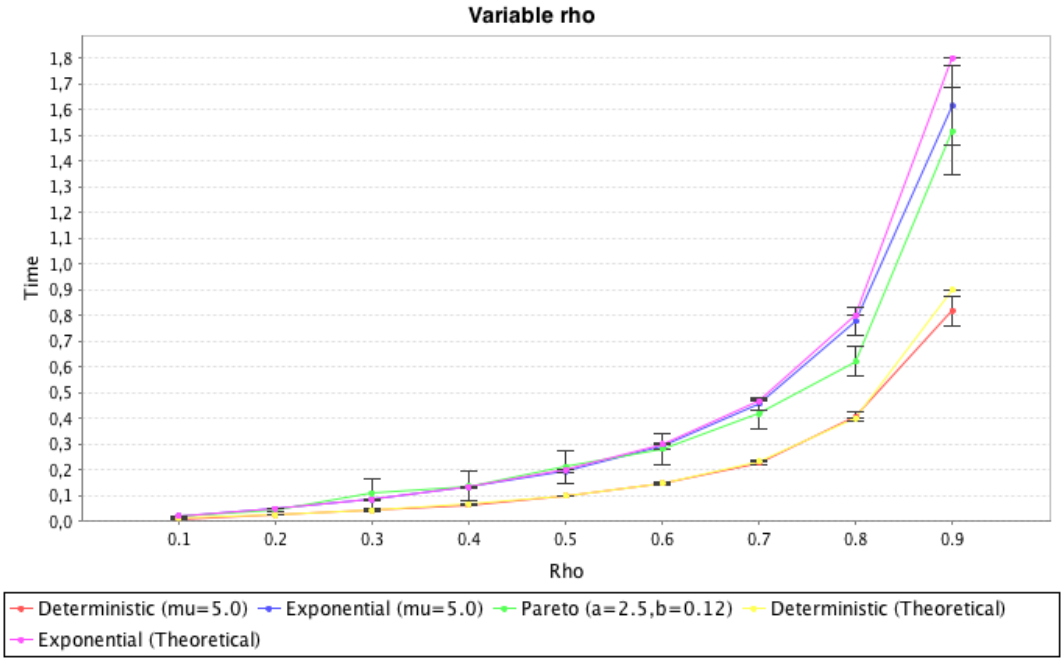

Figura 3.2: Tempi medi di attesa al variare di ρ per i vari tipi di distribuzione

La simulazione di una coda di tipo $M/G/1$ può essere effettuata, come da specifica richiesta, in funzione delle varie tipologie di distribuzione o in funzione della variazione supervisionata di ρ nell'intervallo $(0, 1)$.

La simulazione in funzione della tipologia di distribuzione permette di apprezzare, analogamente a quanto visto in fase di analisi teorica, come la distribuzione di Pareto configurata con parametro $1 < \alpha \leq 2$ presenti un'estrema variabilità, soprattutto se postaa confronto con traffico di tipo Poissoniano o deterministico (figura 3.1).

La simulazione in funzione della variazione di ρ[1] evidenzia la tendenza a divergere del tempo medio di attesa nel sistema η al crescere dell'occupazione dell'unico servitore presente. In figura 3.3 è possibile notare come la caratterizzazione qualitativa e quantitativa dei dati ricalchi in maniera molto fedele l'andamento della curva teorica relativa ai sistemi $M/G/1$.

All'interno della scheda dedicata alla simulazione della coda $M/G/1$ è possibile notare l'opzione relativa alla stima della probabilità di stato della coda, tale opzione è parte delle simulazioni a scelta implementate e verrà meglio esposta al capitolo 6.

[1]la simulazione è stata realizzata facendo variare ρ nell'intervallo $[0.1, 0.9]$ con passo di incremento pari a 0.1

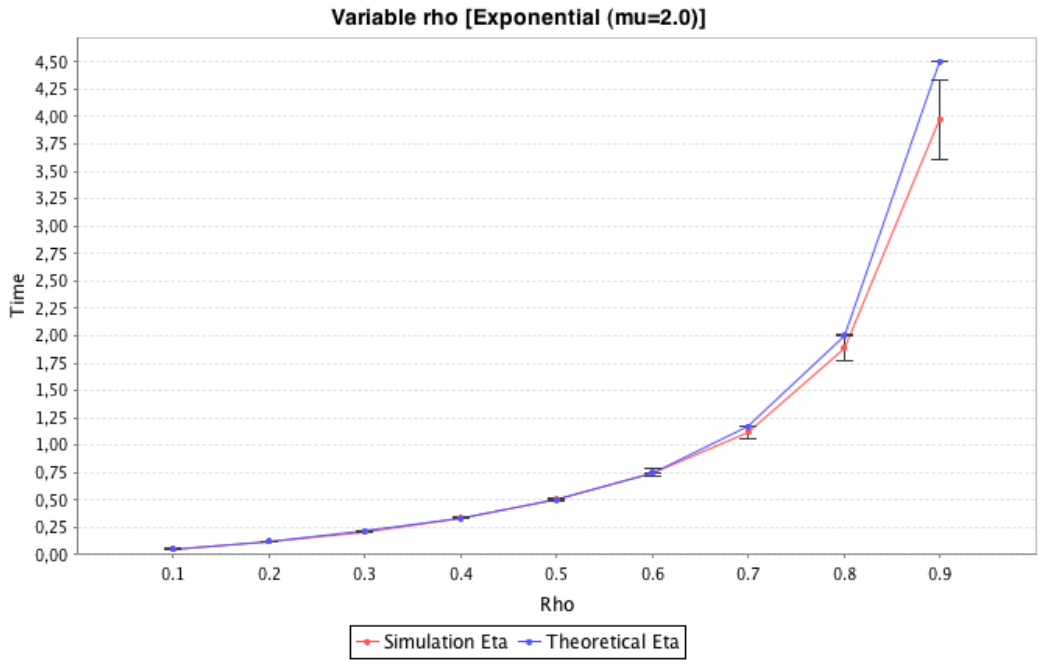

Figura 3.3: Tempi medi di attesa al variare di ρ con distribuzione Poissoniana dei tempi di servizio

3.1.1 Analisi tecnica

La struttura del simulatore è stata definita in aderenza alle specifiche fornite all'interno delle slide a supporto del corso [2]. Al fine aggiuntivo di conferire al sistema una migliore strutturazione dal punto di vista dell'ingegneria del software si è deciso di implementare una classe astratta `Simulator`, svincolata dalla specifica politica di gestione delle code utilizzata.

Tale polica viene infatti gestita specificamente all'interno di due specializzazioni della suddetta classe, tali `FCFSSimulator` e `SJNSimulator`, che nell'ordine realizzano un simulatore in cui la politica di servizio è *"first-come/first-served"* e *"shortest-job next"*.

La gestione della simulazione viene, in fase preliminare, gestita all'interno del pannello dell'interfaccia grafica dedicato. Dopo l'analisi della correttezza dell'input però, al fine di garantire il massimo disaccoppiamento tra elborazione dati e interazione con l'utente, viene generato un nuovo thread che tramite l'utilizzo di metodi dedicati della classe `SimulationRunners` permette lo svolgimento effettivo della simulazione. Al termine di quest'ultima verranno quindi generati eventuali grafici e/o output in console.

Un'ulteriore accorgimento messo in atto per evitare l'andarsi a delineare

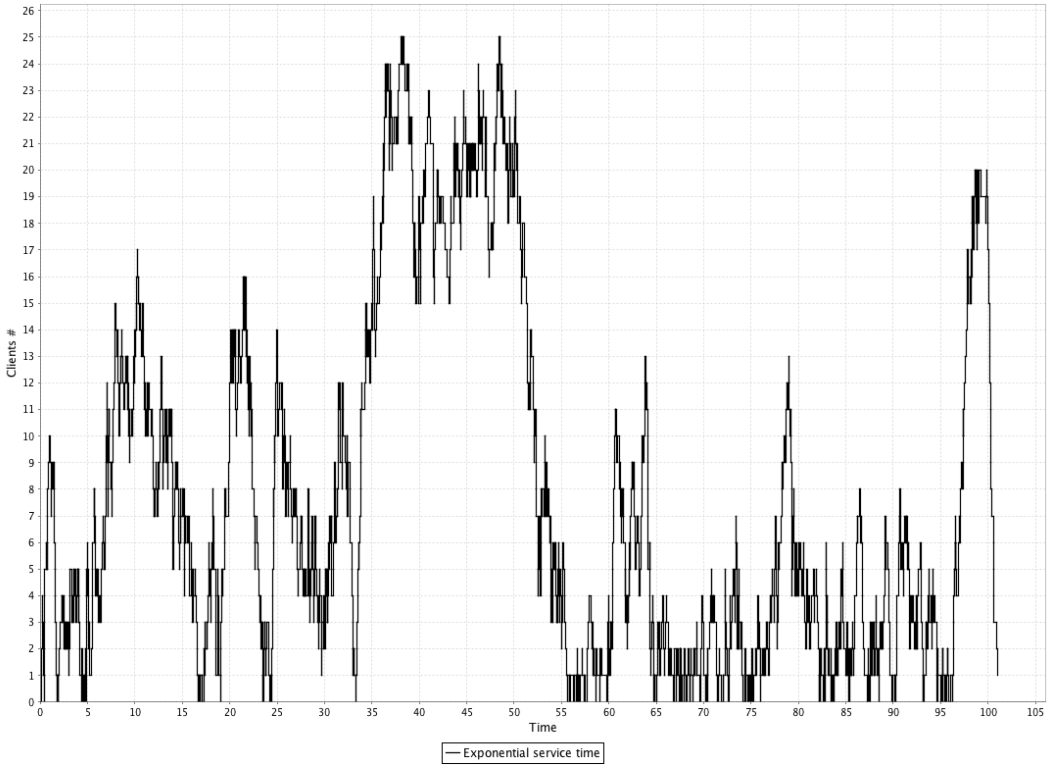

Figura 3.4: Numero di utenti nel sistema $k(t)$ in funzione del tempo, durante una simulazione

di situazione di forte concorrenza in fase di esecuzione[2] è il ricorso alla classe `ExecutorService`. In particolare, l'esecuzione delle simulazioni viene in tutte le sue forme demandata ad un istanza della classe `ExecutorService` instanziata attraverso l'invocazione del costruttore `newSingleThreadExecutor()` della classe `Executor`, che di fatto permette l'utilizzo controllato di un singolo thread per le operazioni a maggior carico computazionale.

In figura 3.5 è possibile osservare la strutturazione definita per entità dedicate alla simulazione.

[2]vista anche la grande richiesta computazionale in fase di simulazione, determinata da una sostanziale prevalenza di cicli cpu-burst.

Figura 3.5: Struttura del package `simulator`

Capitolo 4

Simulazione di una coda $M/G/1//Prio$

4.1 Sistema a coda $M/G/1//Prio$

Per la simulazione di un sistema a coda con classi di priorità assegnata agli utenti è possibile, attreverso l'opportuna scheda, selezionare le varie configurazioni per la caratterizzazione delle classi.

Tali configurazioni comprendono, in aderenza alle specifiche fornite, due o tre classi alle quali sono associate differenti priorità. Per la specificità delle stesse si rimanda alla tabella 4.1.

ID	n. classi	caratteristiche
2	2	$\rho_1 = x\rho$ $\rho_2 = (1-x)\rho$
3a	3	$\rho_1 = \frac{x}{2}\rho$ $\rho_2 = \frac{x}{2}\rho$ $\rho_3 = (1-x)\rho$
3b	3	$\rho_1 = \frac{x}{10}\rho$ $\rho_2 = \frac{9x}{10}\rho$ $\rho_3 = (1-x)\rho$
3c	3	$\rho_1 = x\rho$ $\rho_2 = \frac{(1-x)}{2}\rho$ $\rho_3 = \frac{(1-x)}{2}\rho$

Tabella 4.1: Caratteristiche delle configurazioni di classi disponibili.

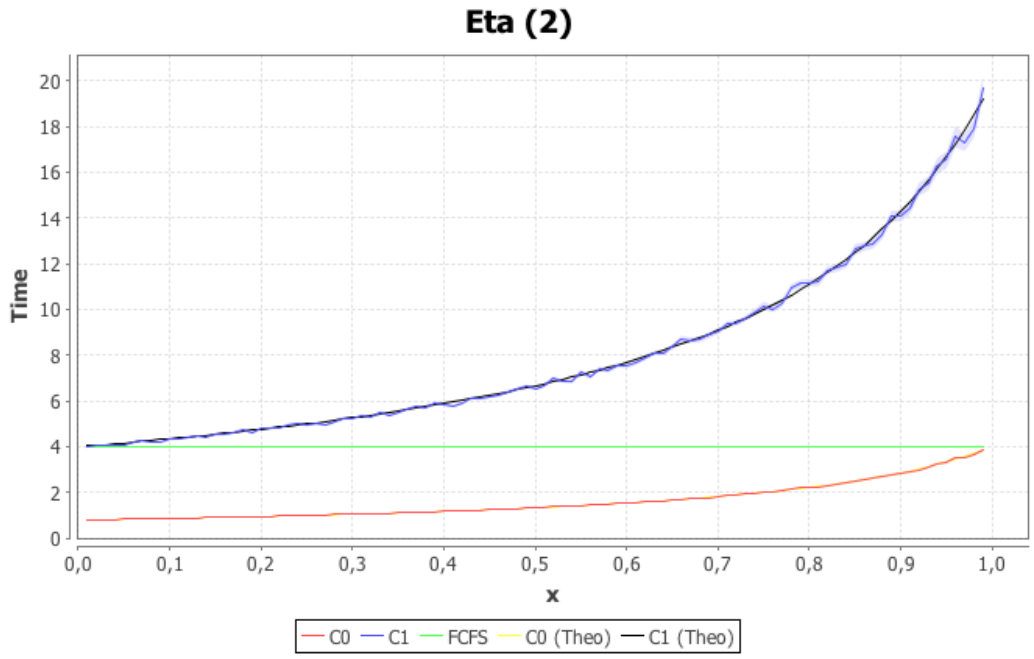

Figura 4.1: Tempi medi di attesa medio nel sistema con coda a due classi di priorità

Analogamente a quando disposto nel caso della simulazione di sitema di tipo $M/G/1$, al termine dell'elaborazione verranno visualizzati i risultati grafici relativi alla simulazione svolta.

Nello specifico, la variabile x viene fatta variare all'interno dell'intervallo (0,1), precedentemente suddiviso in 100 quanti.

Gli andamenti degli $\overline{\eta}$ relativi alle varie classi mostrati all'interno dei grafici sono contraddistinti da una particolare notazione grafica che, attraverso l'utilizzo della medesima colorazione assegnata alla rispettiva classe, evidenzia gli intervalli di confidenza senza rendere difficoltosa l'interpretazione dei grafici, come invece accadrebbe con l'usuale notazione degli stessi.

Le diverse caratterizzazioni delle classi di priorità, definite come da specifica richiesta, sono consultabili in tabella 4.1.

Nota

Nei grafici mostrati in figura 4.1 e 4.2 non è possibile apprezzare a pieno gli intervalli di confidenza a causa della specificità dei test eseguiti. In particolare le simulazioni relative alle suddette figure sono state compiute con quantità di dati parcolarmente elevate, che hanno determinato un'apprezzabile precisione

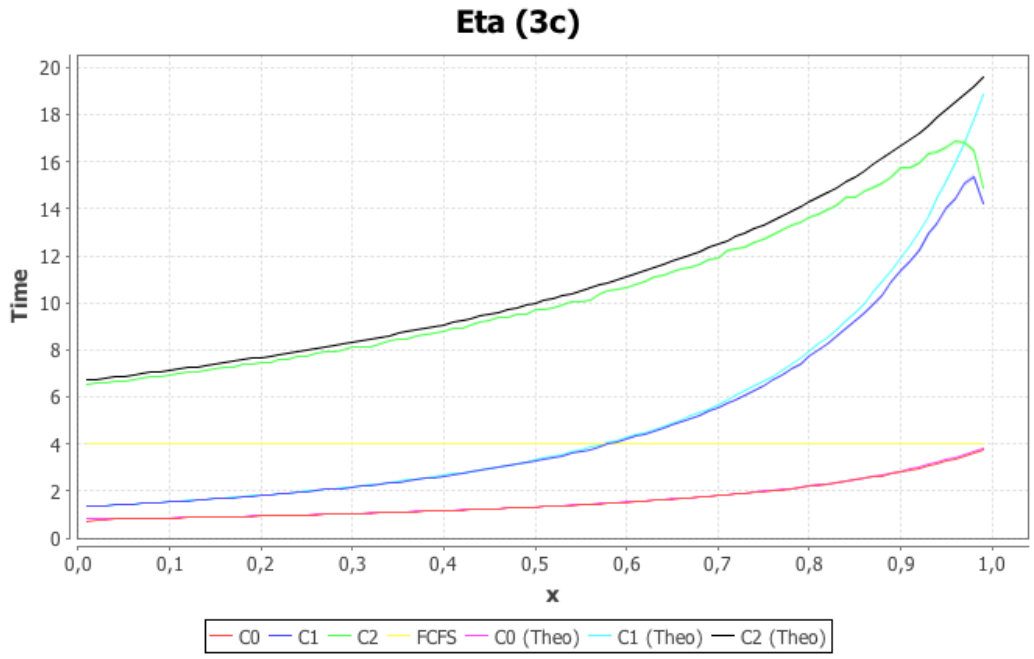

Figura 4.2: Tempi medi di attesa nel sistema con coda a tre classi (tipo c)

dei risultati oltre ad una notevole durata in termini di tempo di simulazione. Dualmente, la figura 4.3 è il risultato di una simulazione a due classi di priorità effettuato specificando un numero molto inferiore di *runs* e *arrivi*, in maniera tale da mostrare l'incertezza dovuta alla discordanza tra i risultati: in particolare nell'immagine è possibile notare, che contorna ciascuna delle due linee, un'ombratura particolare che meglio identifica l'intervallo di confidenza puntuale associato al valor medio di η al variare di x.

4.1.1 Analisi tecnica

Anche in questo caso per l'implementazione della simulazione sono state seguite le linee guida presenti nella documentazione del corso. Il corpo della simulazione, definito all'interno della classe `SimulatorRunners` (si veda fig 3.5), è suddiviso in una prima parte di parsing dell'input fornito dall'utente (metodo `simulateMG1PrioWithVariableRhos(...)` durante il quale si definiscono i valori di $\rho_{1,2,(3)}$ e quindi dalla simulazione vera e propria, implementata all'interno del metodo `simulateMG1Prio(...)`.

Tale metodo si appoggia sulla concretizzazione della classe astratta `Simulator`, `FCFSSimulator`, in cui si implementa un simulatore di tipo singolo servitore, con più code di attesa a decrescente grado di priorità, ciascuna ordinata in

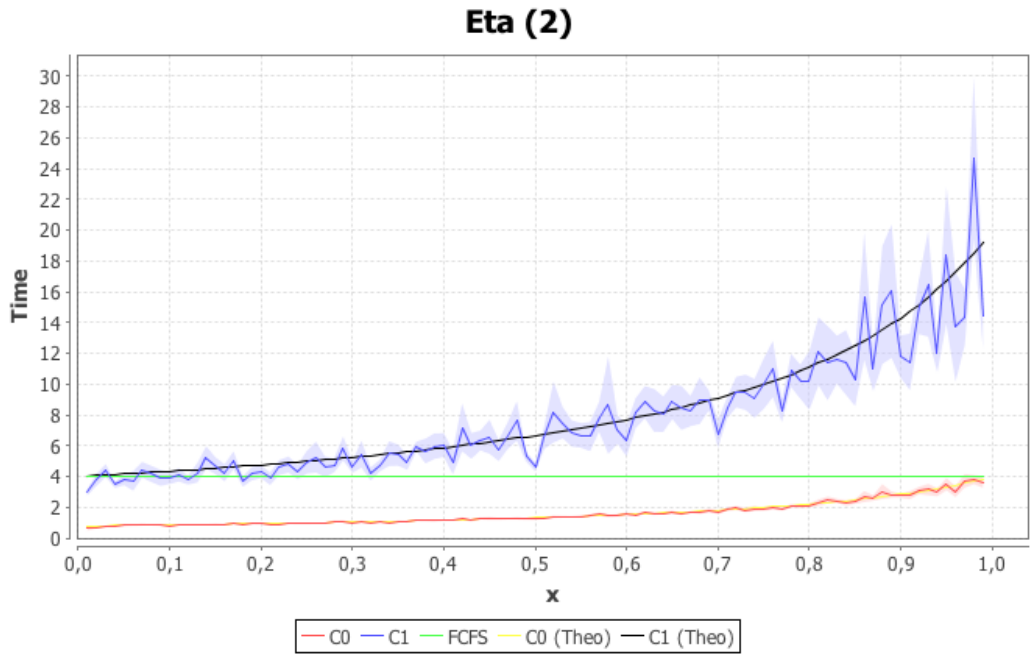

Figura 4.3: Tempi medi di attesa nel sistema con coda a due classi (basso #runs)

base al semplice tempo di arrivo, `OccurenceTime`.
Si vuole quì inoltre notare che i singoli eventi che accadono all'interno del simulatore (arrivi e partenze) vengono mappati in analoghe classi (`Event`, `Arrival` e `Departure` rispettivamente) organizzate in una gerarchia che vede sia gli arrivi che le partenze come specializzazioni del generico (e astratto) evento. Inoltre, come scelta progettuale, si è deciso di non includere nei singoli eventi anche la relativa politica di ordinamento, che viene invece specificata di volta in volta in classi appositamente create, derivanti da `ComparableEvent`, le quali fungono da *wrapper* dello specifico evento e ne stabiliscono i criteri di ordinamento; nel caso in questione una particolare sottoclasse di `ComparableEvent`, `OccurrenceTimeComparedEvent`, stabilisce, come politica, un ordinamento semplicemente basato sul tempo di occorrenza dell'evento: *first-come-first-served*.

Capitolo 5

Simulazioni opzionali

5.1 Stima della probabilità di stato del sistema $M/G/1$

Il calcolo ed il grafico relativi alla probabilità di stato k all'interno di un sistema a coda $M/G/1$ sono accedibili attraverso la scheda relativa al sistema $M/G/1$, le cui caratteristiche sono state precedentemente esposte (cap. 3).

All'utente viene lasciata la libertà di definire il grado di occupazione del servitore del sistema ρ, il tempo medio di servizio μ, il numero di run da effettare N, il numero di arrivi per ciascuna run e, ovviamente, la tipologia di distribuzione.

Come dati esemplificativi di questa gamma di simulazioni sono stati scelti quelli rappresentati in fig. 5.1, che descrive la probabilità di stato di un sistema con tempo di servizio deterministico, e quelli di fig. 5.2, raffigurante le probabilità di stato di un sistema con tempo di servizio distribuito secondo Pareto. È possibile notare come, a causa della maggior variabilità dei tempi di servizio, il sistema con traffico di Pareto raggiunga stati di congestione caratterizzati da un alto numero di utenti nel sistema. Le probabilità di stato infatti, sono non nulle anche con k pari a qualche centinaio di utenti.

5.1.1 Analisi tecnica

Analogamente a quanto visto in tutti casi precedenti ed in aderenza al modello definito per il simulatore, dopo un prima parte di handling da parte del pannello relativo all'interno dell'interfaccia grafica (classe `MG1Panel`), si ricorre alla classe `SimulationRunners`, ed in particolare al suo metodo:

· `simulateMG1EvaluatingProbability(...)`

per l'esecuzione effettiva, mostrando il grafico di $P(k)$ in conclusione.

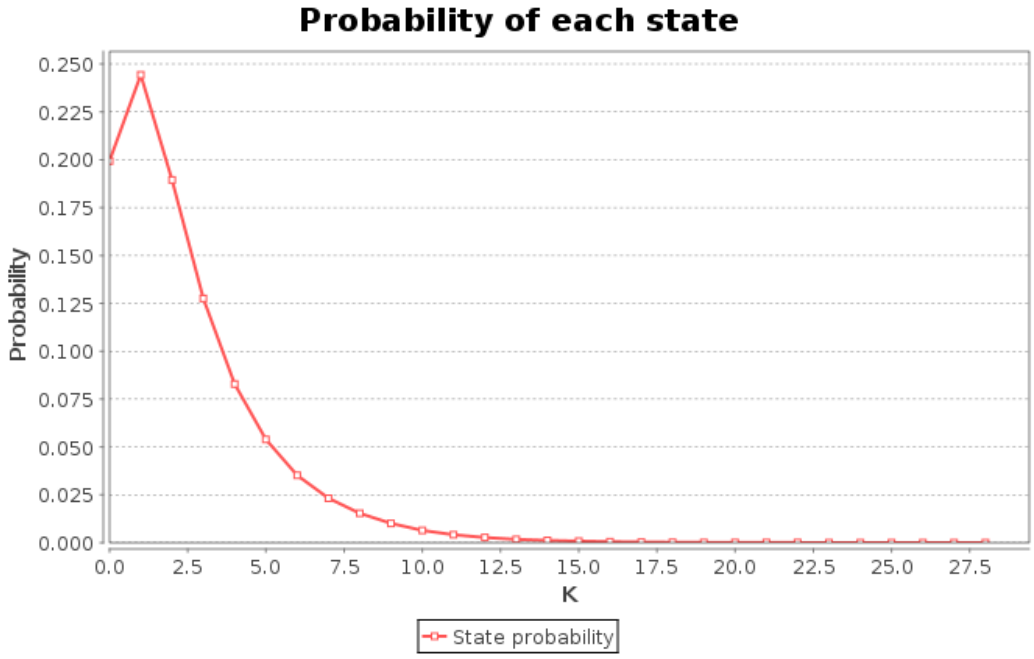

Figura 5.1: Probabilit'a di stato k, $\rho = 0.8$, $\mu = 1$, $N = 100$, $arrivi = 1000$ (Deterministic service time)

Tale metodo, per funzionare correttamente, si appoggia sul già precedentemente descritto `FCFSSimulator`, utilizzato quì con una sola classe di priorità (quindi una singola coda di attesa), il quale fornisce un metodo ad-hoc, `getStatesProbability`, che restituisce, per ogni stato $K(t)$ raggiunto durante la simulazione, la sua probabilità empirica, calcolata semplicemente tenendo traccia del tempo totale trascorso in quel particolare stato, diviso per il tempo totale della simulazione.

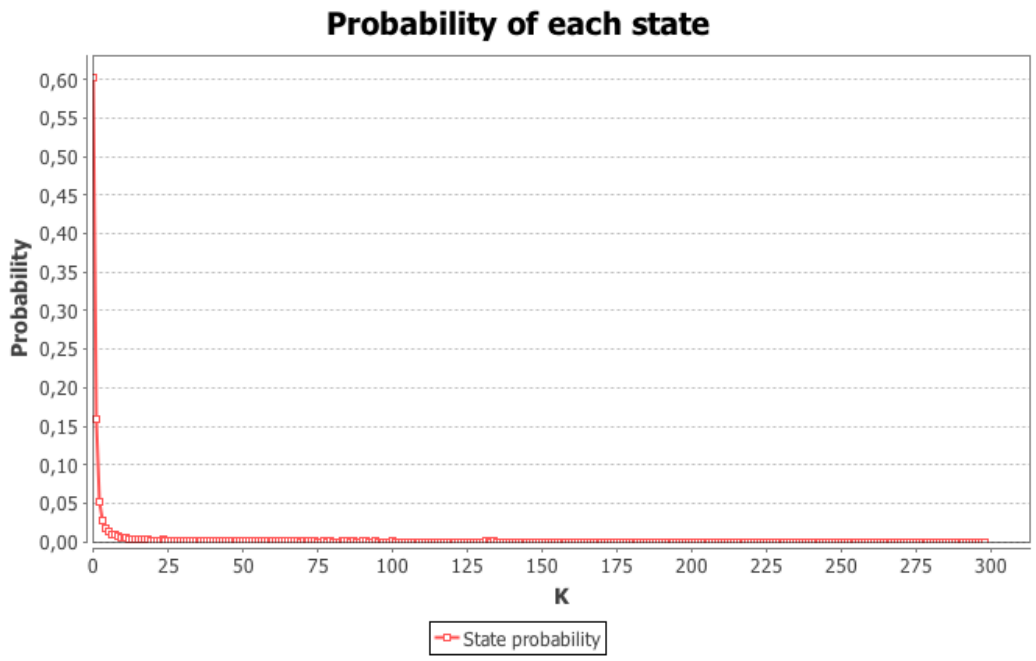

Figura 5.2: Probabilit'a di stato k, $\rho = 0.5$, $\mu = 2$, $N = 100$, $arrivi = 1000$ (Pareto service time)

5.2 Sistema $M/G/1$ con politica *Shortest Job Next*

Come seconda simulazione opzionale si è scelto di implementare un sistema a coda con politica di tipo *shortest job first*.

In questo caso all'utente viene chiesto di determinare, oltre agli usuali parametri di simulazione, il numero di step da considerare ed un moltiplicatore di θ al fine di eseguire la discretizzazione logaritmica dei tempi di servizio degli utenti del sistema in funzione della costruzione del grafico visualizzato al termine dell'elaborazione.

Il moltiplicatore di θ, in particolare, andrà a determinare l'estensione del grafico lungo l'asse delle ascisse influendo dunque sulla porzione di curva visualizzata.

In figura 5.3 è possibile osservare l'esito di una simulazione con parametri: $\rho = 0.8$, $\mu = 0.5$, $N = 1000$, arrivi= $100'000$, $0 \rightarrow \theta$ steps= 30 ed intervallo di considerazione pari a 10θ: si noti come attraverso la discretizzazione logaritmica gli intervalli siano più concentrati e di dimensioni più ridotte intorno alla media (2) e di dimensioni maggiori man mano che ci si allontana dalla stessa.

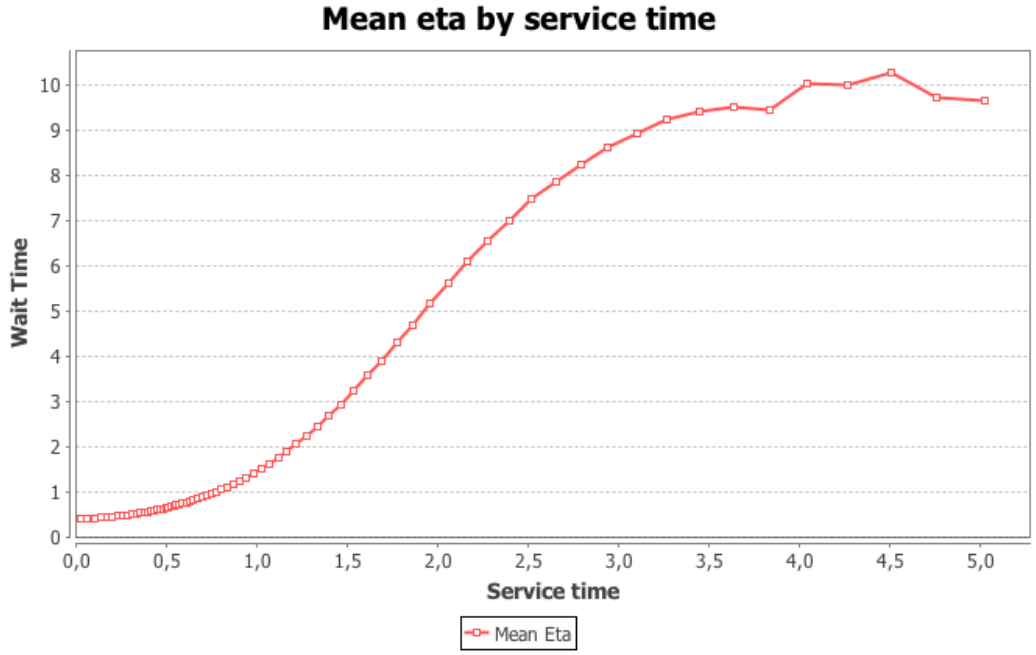

Figura 5.3: $\overline{\eta}$ in funzione del tempo di servizio θ

5.2.1 Note sulla discretizzazione logaritmica

Al fine di ottenere simulazioni più accurate, l'intervallo continuo θ viene discretizzato, anzichè mediante una classica funzione lineare, ricorrendo ad una più complessa funzione logaritmica in modo tale da avere una maggiore concentrazione di piccoli intervalli nell'intorno della media di θ stesso, dove è più probabile ottenere valori, e pochi intervalli sempre più grandi mano a mano che ci si allontana da $\overline{\theta}$. La funzione utilizzata risulta essere la seguente:

$$f(x) = \begin{cases} -\log(-x+\overline{\theta}+b)+a+\overline{\theta} & \text{for} \quad x < \theta \\ \log(+x-\overline{\theta}+b)-a+\overline{\theta} & \text{for} \quad x \leq \theta \\ a = \overline{\theta} \cdot \frac{10^{-\overline{\theta}}}{1-10^{-\overline{\theta}}} \\ b = log(a) \end{cases}$$

parametrizzata rispetto a $\overline{\theta}$, a e b; dove

- $\overline{\theta}$ è la media attorno alla quale i valori di θ si dispongono
- i parametri a e b dipendono da $\overline{\theta}$ e servono per forzare la funzione $f(x)$ a passare dai punti $(0,0)$ e $(\overline{\theta},\overline{\theta})$

Utilizzando una scala lineare lungo l'asse delle y, con passo costante e arbitrariamente definibile, e calcolandosi i relativi valori di x mediante $f^{-1}(y)$ si

ottengono i suddetti intervalli concentrati attorno al valore di $\overline{\theta}$; il valore scelto come rappresentativo di ciascun intervallo è semplicemente il valore mediano $x_c = \frac{x_i + x_{i+1}}{2}$

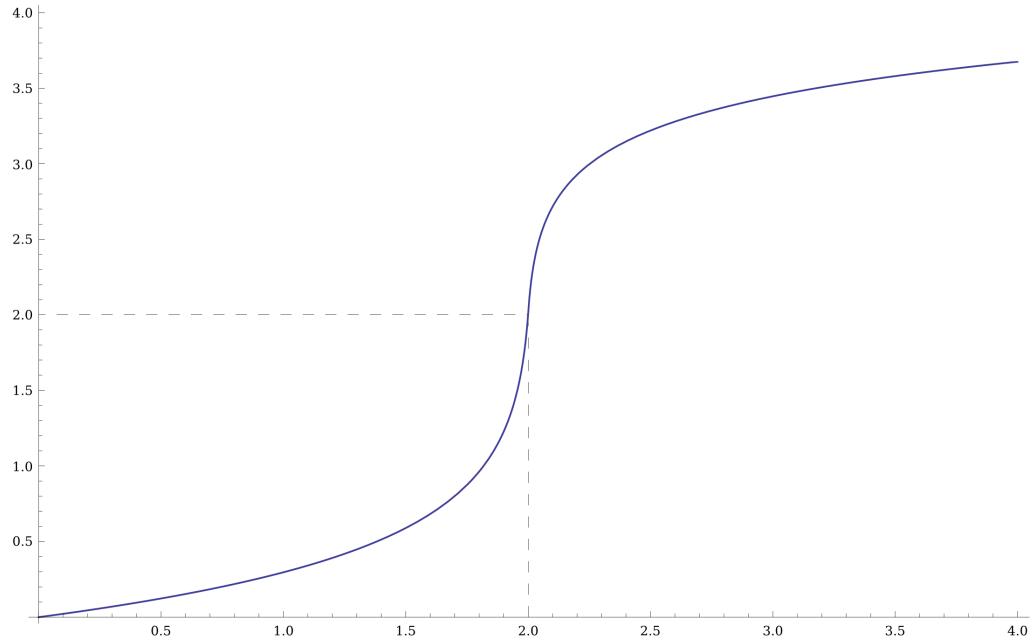

Figura 5.4: esempio di una funzione logaritmica usata per la discretizzazione nel caso di $\overline{\theta} = 2$

5.2.2 Analisi tecnica

Nei casi precedenti, l'utilizzo (indiretto) della classe astratta `Simulator` era sempre avvenuto attraverso la sua specializzazione `FCFSSimulator`, sulla quale è stata implementata una gestiene di tipo FIFO.

Per la diversa tipologia di politica si è quindi provveduto all'implementazione della classe `SJNSimulator`, che unitamente al metodo `simulateMG1SJN(...)` della classe `SimulationRunners` permette la definizione della nuova dinamica all'interno del sistema.

La classe `SJNSimulator` concretizza la classe astratta `Simulator` e, analogamente alla classe `FCFSSimulator`, utilizza un'apposita specializzazione di `ComparableEvent`, `ServiceTimeComparedEvent`, la quale adotta come criterio di ordinamento il `ServiceTime` di ciascun evento: eventi piú corti verranno

serviti prima. Una differenza importante rispetto al FCFSSimulator risiede nel calcolo dell'$\overline{\eta}$ per ciascuna *classe*: infatti in questo caso, la coda di attesa è unica, non esistono priorità differenti, ma viene richiesto di tenere traccia dei differenti tempi di attesa in base ad una discretizzazione dell'intervallo del tempo di servizio, θ. A questo fine, ad ogni tempo di servizio, tramite la classe `LogaritmicQuantizator`, viene assegnata una classe di appartenenza, con la quale aggiornare il relativo valore di attesa in coda.

Nota finale

Per ulteriori dettagli tecnici sull'implementazione del simulatore si rimanda alla consultazione della documentazione specifica (javadoc) contestualmente prodotta.

Capitolo 6

Struttura del codice sorgente

6.1 Struttura dei package

Il codice sorgente è stato strutturato in tre package principali al fine di ottenere una buona separazione delle varie componenti presenti all'interno del progetto.

Si riporta di seguito la struttura completa dei package:

- `gui`
    - `panels`
- `simulator`
    - `distribution`
    - `events`
    - `misc`
    - `random`
- `launchers`

All'interno del package `gui` sono state implementate le classe dedicate all'interfaccia grafica. Il package `gui.panels` contiene i pannelli dell'interfaccia dedicati a ciascuna tipologia di simulazioni.

Il package `simulator` contiene la gerarchia di classi relative all'implementazione dei vari meccanismi di simulazione, fattorizzando le operazioni comuni all'interno della classe padre `Simulator`. È stato necessario implementare numerose tipologie di simulatore poiché in ciascuna delle politiche di scheduling considerate gli eventi da elaborare vengono gestiti in maniera specifica. I package `distribution`, `events`, `misc` e `random`, contenuti all'interno di `simulator`,

contengono le classi utilizzate dal simulatore per modellare eventi, distribuzioni di numeri casuali e le implementazioni dei metodi utili ai fini dei calcoli statistici.

Il package `launcher` contiene gli entry-point dell'applicazione: in particolare la classe `SimulatorGUI` provvede all'avvio del simulatore, mentre gli ulteriori metodi presenti al suo interno avviano alcune interessanti simulazioni accessorie non accedibili dall'interfaccia grafica principale.

Capitolo 7

Conclusioni

7.1 Simulatore di teletraffico

L'utilizzo di un simulatore di teletraffico ha dato la possibilità di andare a valutare il comportamento di sistemi di servizio noti a fronte di richieste inoltrate da popolazioni generate artificialmente ma caratterizzate da comportamenti riconducibili a casi reali. L'aderenza tra i modelli di traffico generati ed i corrispettivi teorici è garantita dalle specifiche dei generatori di numeri pseudo-casuali ai quali si è fatto ricorso.

In termini generali è comunque bene tenere conto del fatto che la simulazione di teletraffico rappresenti in sé un problema di grande complessità. Al di là delle considerazioni sulla qualità dei numeri pseudo-casuali ai quali si fa riferimento è infatti importante tenere conto della realtà dei sistemi che si intendono simulare, nei quali le dinamiche di funzionamento sono determinate da grandi quantità di variabili, spesso difficili anche da identificare, oltre che da esprimere attraverso un modello.

Fatta questa precisazione, la creazione e l'utilizzo di un simulatore possono sicuramente rappresentare una risorsa di indiscutibile utilità per la sintesi ed il dimensionamento di sistemi di teletraffico per applicazioni reali, rendendo possibile un'immediata rappresentazione e valutazione di caratteristiche e dinamiche altrimenti difficili da contemplare.

Bibliografia

[1] F. Callegati, G. Corazza *"Elementi di teoria del traffico per le reti di telecomunicazioni"* Società Editrice Esculapio, Bologna 2006

[2] W. Cerroni *Lucidi del secondo modulo del corso di Progetto di reti di telecomunicazioni* DEIS, Bologna 2010

[3] M. Matsumoto, T. Nishimura *"Mersenne Twister: A 632-dimensionally equidistribuited uniform pseudorandom number generator"* Keio University 1997

[4] David Gilbert *"The JFreeChart Class Library - Developer Guide (v.1.0.9)"* Object Refinery Limited 2000-2008

Elenco delle figure

Elenco delle tabelle