Traffic Simulator

Michael Gattavecchia Marco Santarelli Andrea Zagnoli

ALMA MATER STUDIORUM
II Facoltà di Ingegneria - Sede di Cesena
Tecnologie e Sistemi per la Sicurezza

16 giugno 2010

Traffic Simulator

Realizzazione di un simulatore software di sistemi soggetti a traffico, in particolare di sistemi a singolo servitore e caratterizzati da una statistica del tempo di arrivo di tipo *Poissoniano*

Simulazione e presentazione dei relativi risultati di differenti aspetti concernenti tali sistemi (con la possibilità di variarne i parametri al contorno)

- generatori di numeri casuali
- simulatori $M/G/1$
- simulatori $M/G/1//Prio$
- simulatori $M/G/1//SJN$

Gruppo di lavoro

Componenti del gruppo

- Michael Gattavecchia (*n/m 0000362269*)
- Marco Santarelli (*n/m 0000346651*)
- Andrea Zagnoli (*n/m 0000367565*)

Generazione di numeri casuali

Alla base di una qualsivolglia simulazione risiede la centralità del ruolo rivestito dalla riproduzione dell'aleatorietà riscontrabile all'interno dei sistemi reali.

Perció risulta 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.

Generatori LC

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

Particolarmente apprezzati per la loro leggerezza, dovuta alla semplicità dei calcoli necessari per la generazione.
Esiste correlazione tra valori generati successivamente, che determina l'impossibilità di utilizzo di questo tipo di generatori in ambienti in cui tale correlazione tappresenti un fattore di rischio (e.g. crittografia).

`java.util.Random`

Rappresenta lo standard per la generazione di valori pseudo-random all'interno dell'invironment Java.

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

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

Presenta un periodo pari a 2^{48} ed utilizza un seme a 48 bit per l'inizializzazione.

Generatore ran0

Generatore LC realizzato ex-novo.

parametro	valore
m	2147483647
a	16807
c	0

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

Il valore di $c = 0$ ne fa un generatore “moltiplicativo”, di tipo *Park-Miller*

`MersenneTwister` / `SecureRandom`

MersenneTwister

Generatore implementato dal team Apache[1], implementa l'algoritmo Matsumoto-Nishimura (1996-97). Basato sulla ricorsione lineare: $x_{k+n} := x_{k+m} \oplus (x_k^u | x_{k+1}^l)A, (k = 0, 1, ...)$
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.

SecureRandom

Presente all'interno del package `java.security` permette la generazione robusta dal punto di vista crittografico di numeri pseudo-casuali.

[1] http://www.apache.org

Valutazione della distribuzione [0,1]

Metodo per la valutazione visuale della distribuzione dei valori generati.
Coppie di valori vengono considerate come coordinate x e y e poste su di un piano cartesiano, in cui l'uniformità di distribuzione dei valori nello spazio sarà indicatore della qualità del generatore.

Scatter plot

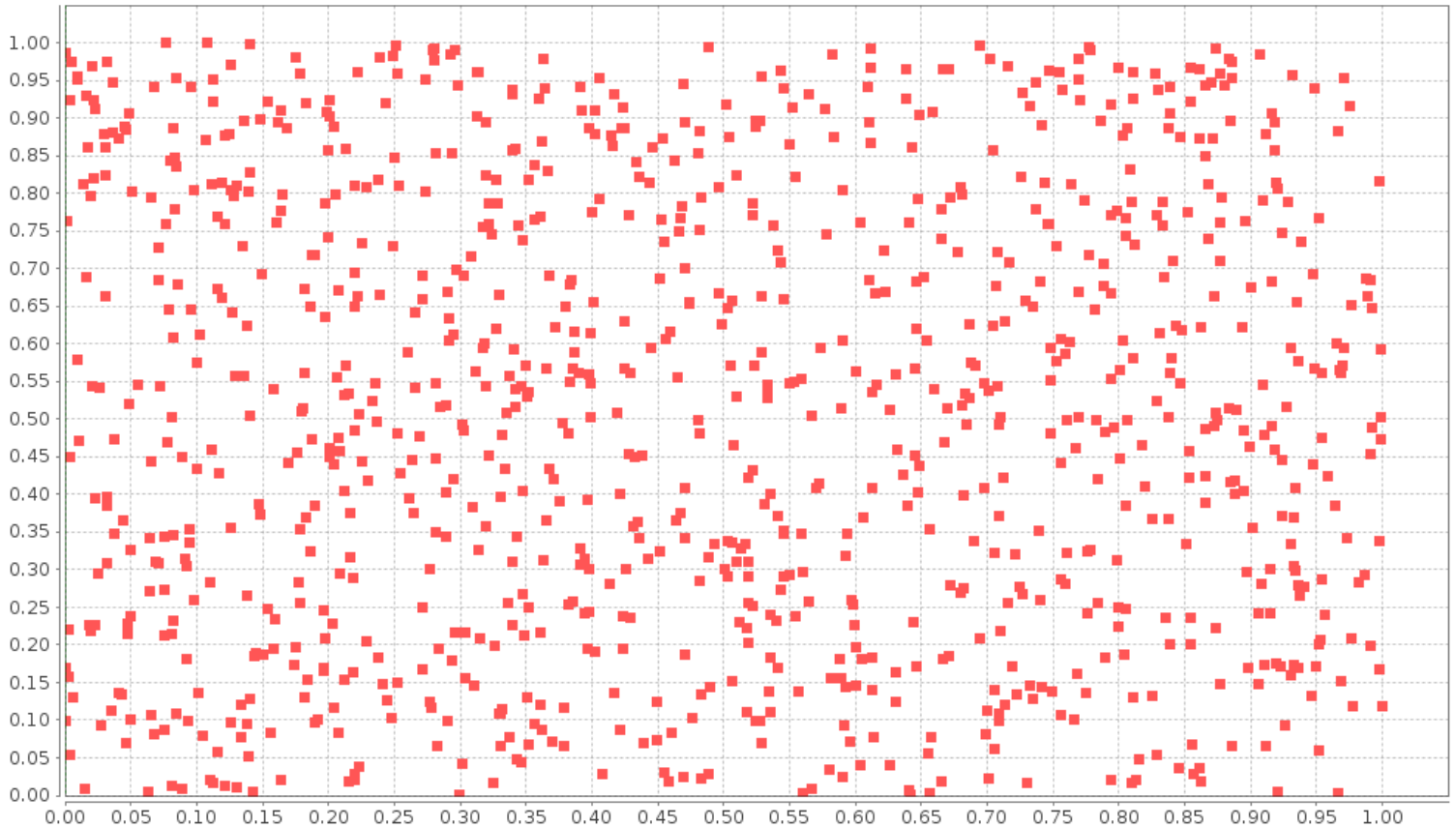

Figura: Distribuzione dei valori generati da `util.Random`

Stima del valore medio

Al fine di valutare il valore medio delle serie di valori prodotti sono state effettuate alcune serie di generazioni di campioni di dimensione crescente.
All'aumentare della numerosità dei campioni si nota la convergenza della media al valore teorico 0.5.
Si notano oscillazioni rilevanti esclusivamente in concomitanza a campioni di dimensioni ridotte.

Media dei campioni

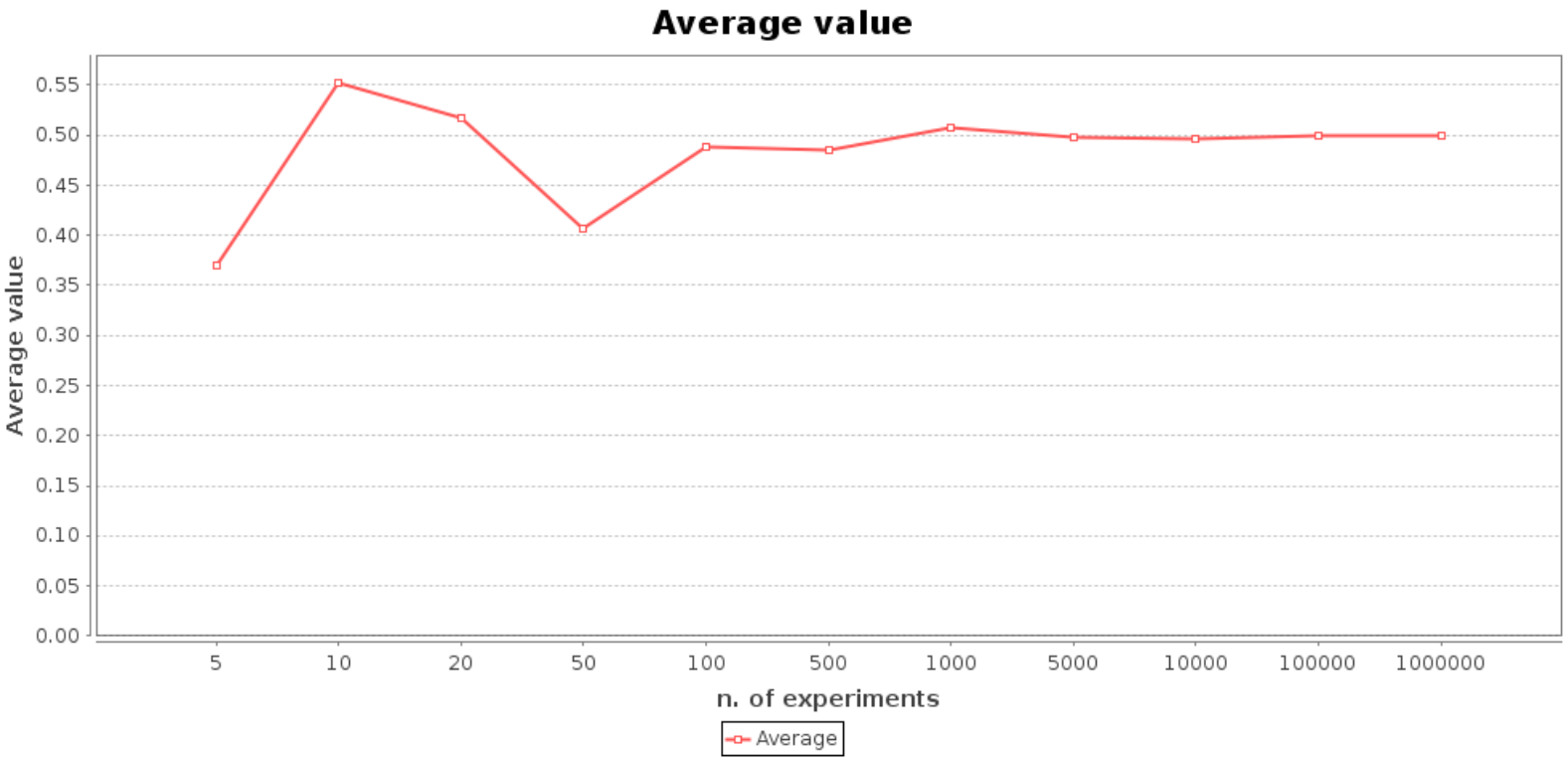

Figura: Media dei campioni in funzione della numerosità

Intervallo di confidenza (1/2)

L'intervallo di confidenza è stato valutato in funzione del livello di confidenza richiesto...

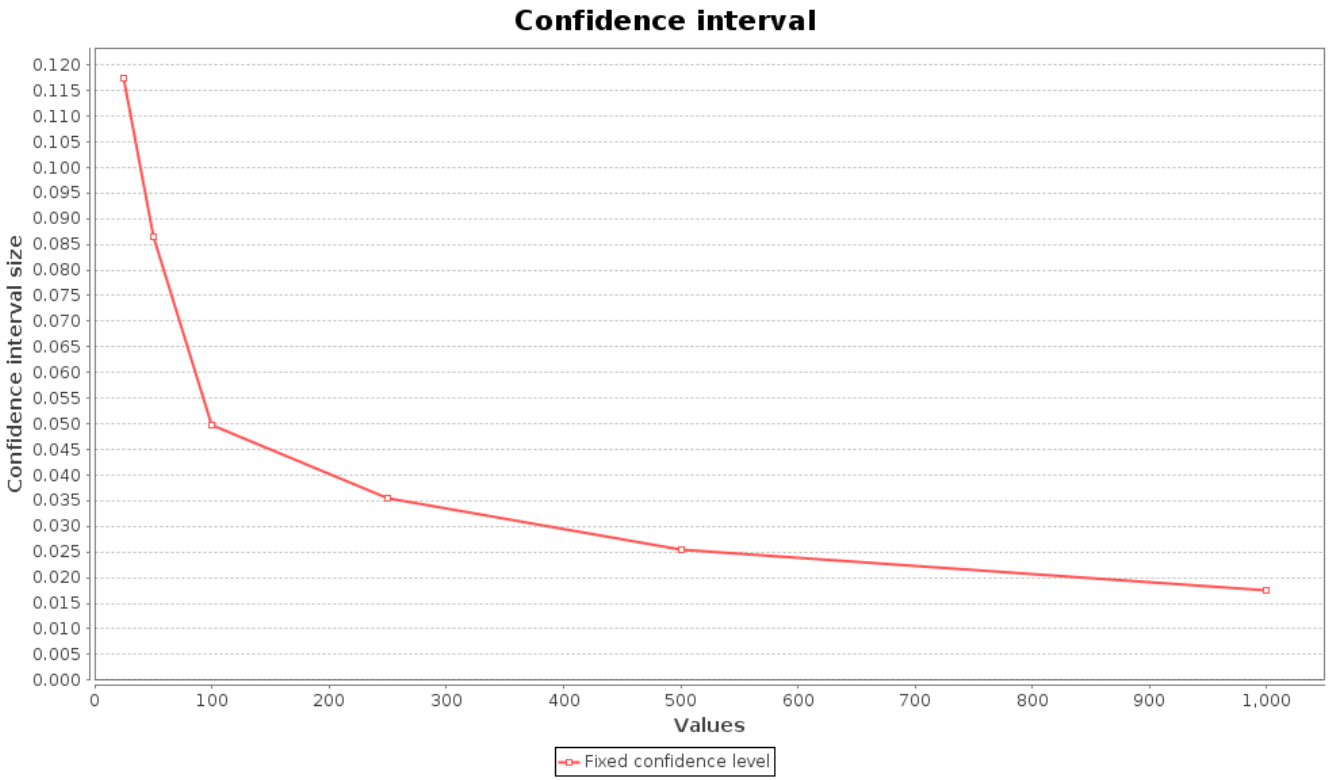

Figura: Intervallo di confidenza in funzione del numero di campioni

Intervallo di confidenza (2/2)

...ed al variare della dimensione dei campioni

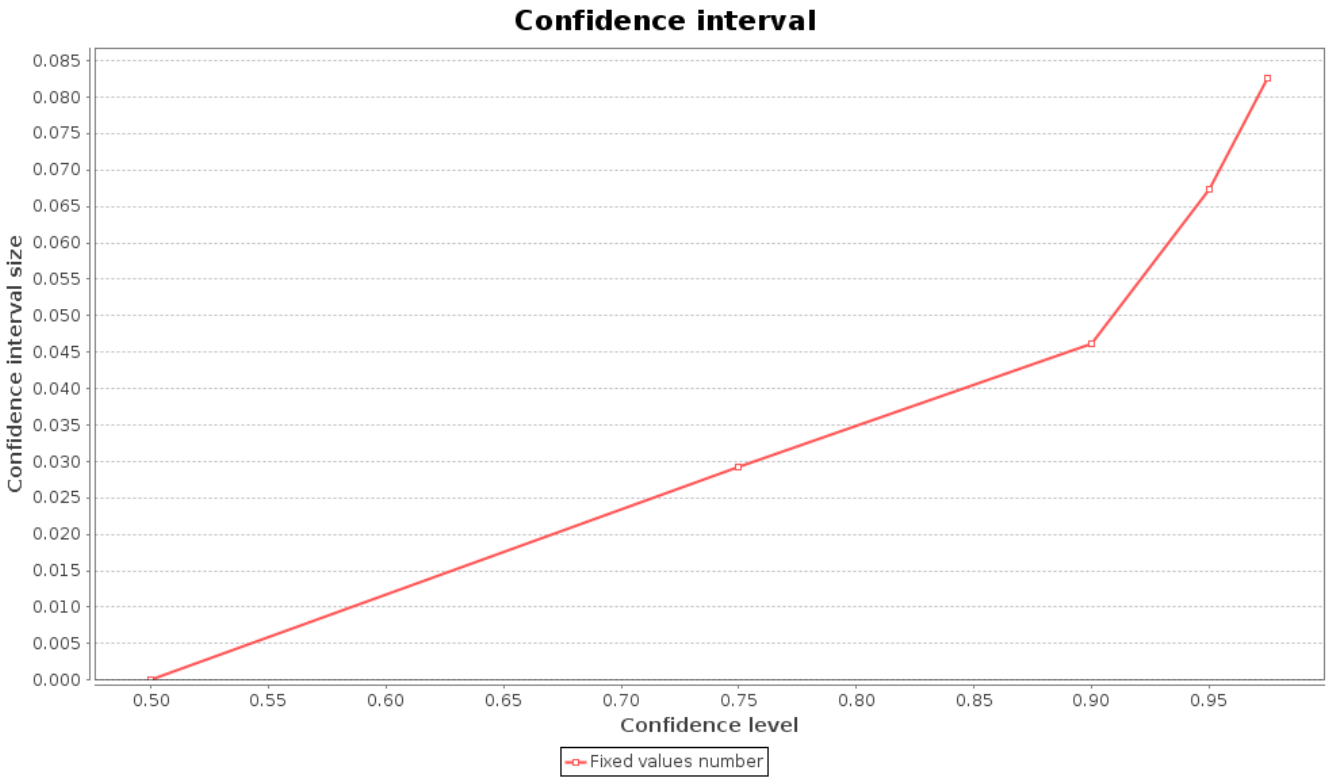

Figura: Intervallo di confidenza in funzione del livello di confidenza

Analisi tecnica

Le simulazioni relative all'intervallo di confidenza sono state realizzate attraverso i metodi:

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

della classe `SimulationRunners`, che contiene i metodi utilizzati per svolgere le varie tipologie di simulazioni disponibili.

Generazione di traffico

Generazione di eventi secondo differenti distribuzioni

- Deterministico
- Esponenziale
- SPP
- Pareto

Parametri di traffico scelti in modo da ottenere per ogni distribuzione lo stesso λ medio.
Confronto dei vari modelli di traffico, numero medio di eventi occorsi in un dato periodo, varianza e IDC.

Confronto modelli di traffico

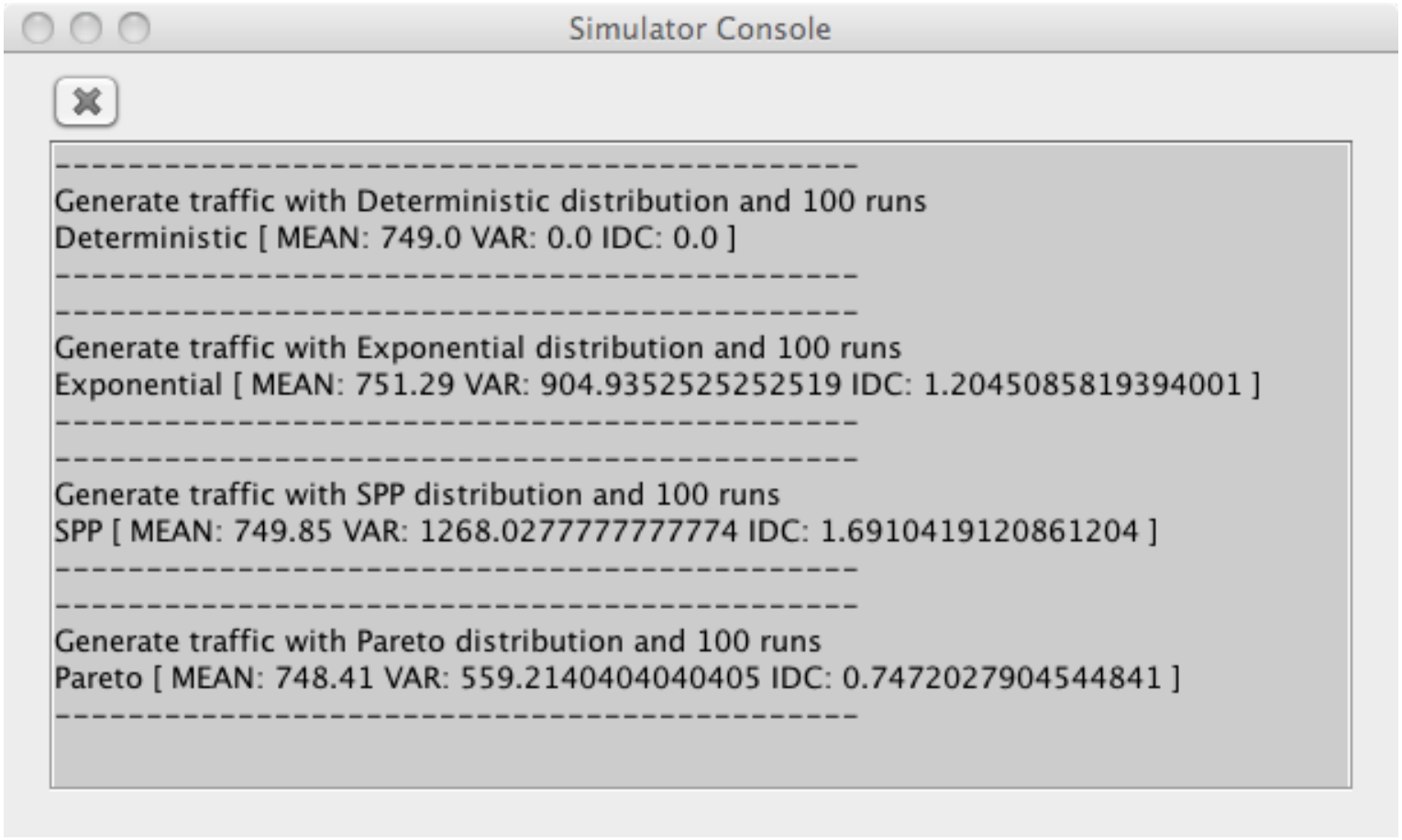

Figura: Confronto dei valori medi di vari tipi di distribuzione

Simulatore $M/G/1$

Sistema a singolo servitore e a singola coda di attesa, con arrivi di tipo poissoniano e tempo di servizio di tipo variabile

- Simulazione al variare di ρ in $[0, 1]$
- Simulazione con tempo di servizio Esponenziale, Deterministico, SPP e di Pareto

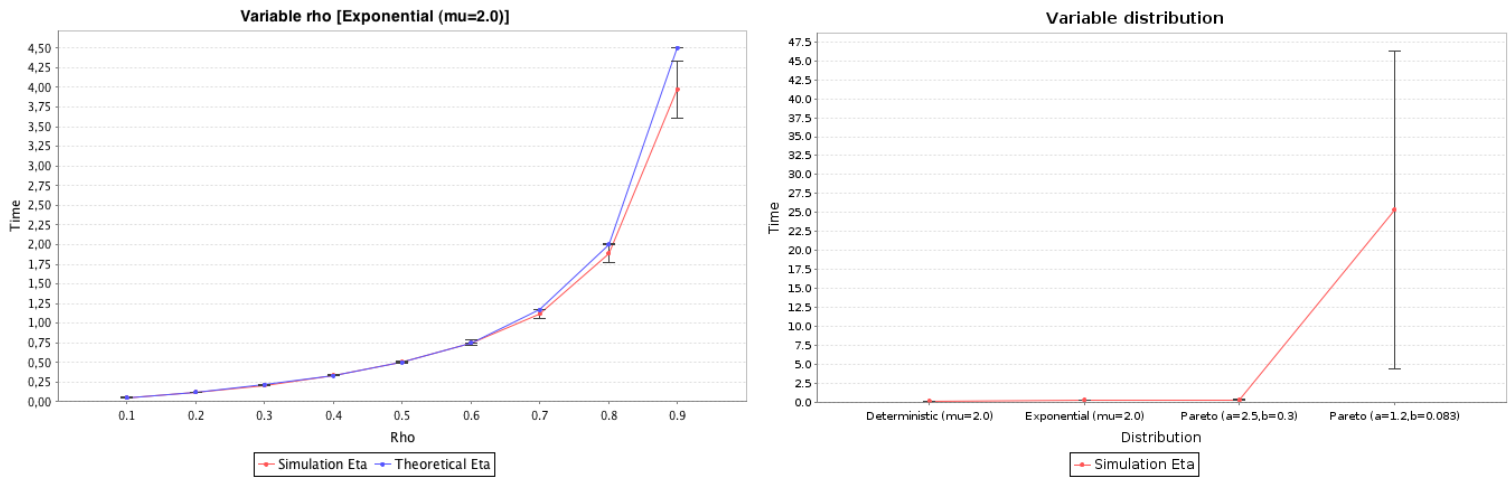

Figura: Simulazioni $M/G/1$ al variare di ρ e del tipo di distribuzione.

Dettagli tecnici (1/2)

- Validazione dei parametri in ingresso, relativamente al tipo di simulazione scelta
- Metodi `compareMG1Simulations()` and `simulateMG1WithVariableRho()`
- Classe astratta, svincolata dalla particolare politica di gestione delle code, con implementazioni fattorizzate dei metodi comuni
- `FCFSSimulator`, implementa anche la possibilità di specificare classi di priorità

Dettagli tecnici (2/2)

- Simulatore di tipo singolo servitore
- Singola coda di attesa ordinata in base all'occurrence time
- Eventi di tipo arrival and departure, organizzati secondo una gerarchia di classi
- Politica di ordinamento specificata in classi wrapper appositamente create, che ereditano da `ComparableEvent`
- `OccurrenceTimeComparedEvent`, stabilisce, come politica, un ordinamento semplicemente basato sul tempo di occorrenza dell'evento first-come first-served.

Sistemi $M/G/1/Prio$ (1/3)

Sistemi a singolo servitore con differenti code di attesa, ciascuna delle quali contraddistinta da uno specifico grado di priorità.

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: Caratteristiche delle configurazioni di classi disponibili.

Sistemi $M/G/1/Prio$ (2/3)

Il parametro x viene fatto variare da 0 a 1 con passi di 0.01

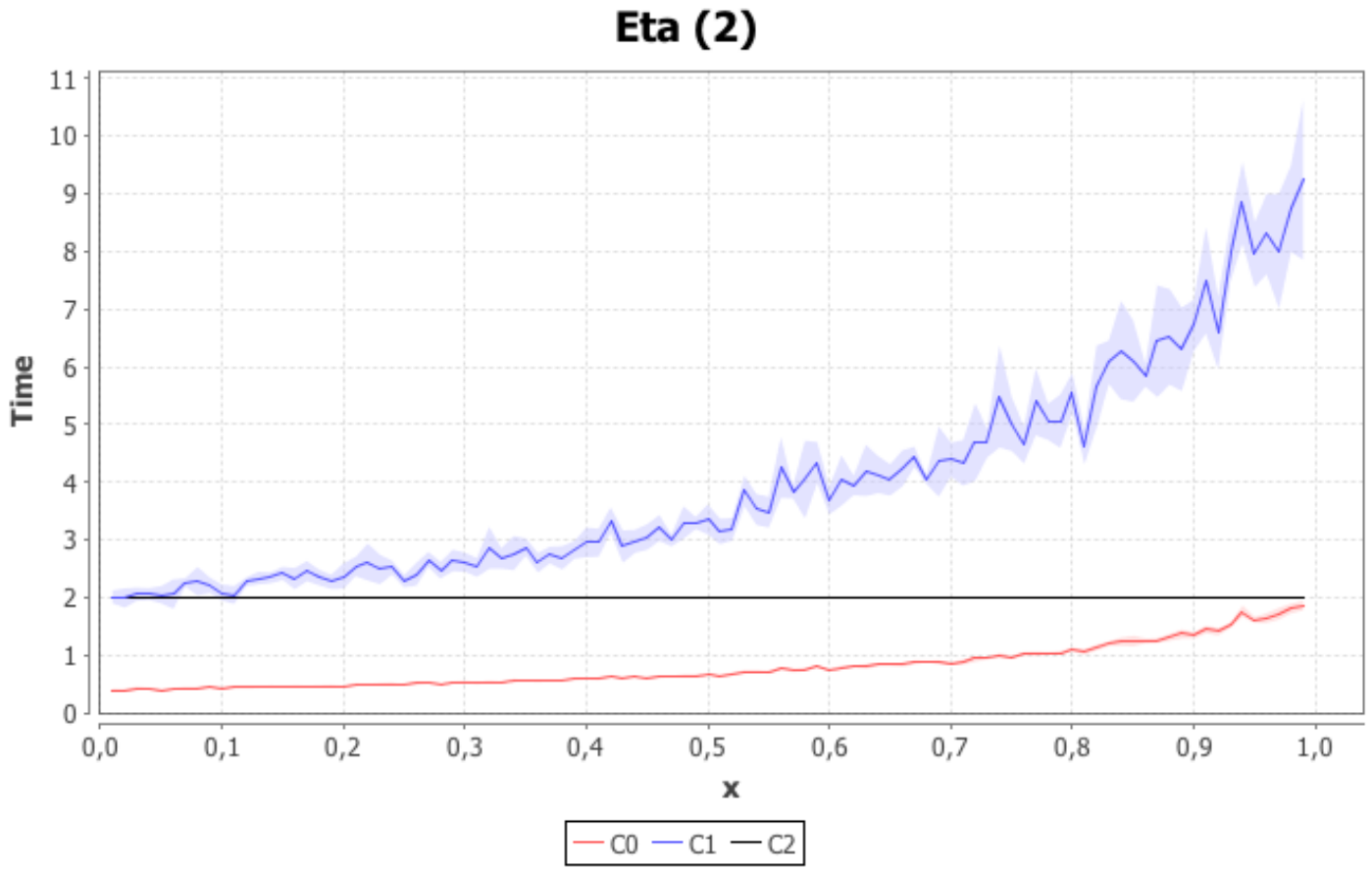

Figura: Tempi medi di attesa nel sistema con coda a due classi

Sistemi $M/G/1/Prio$ (3/3)

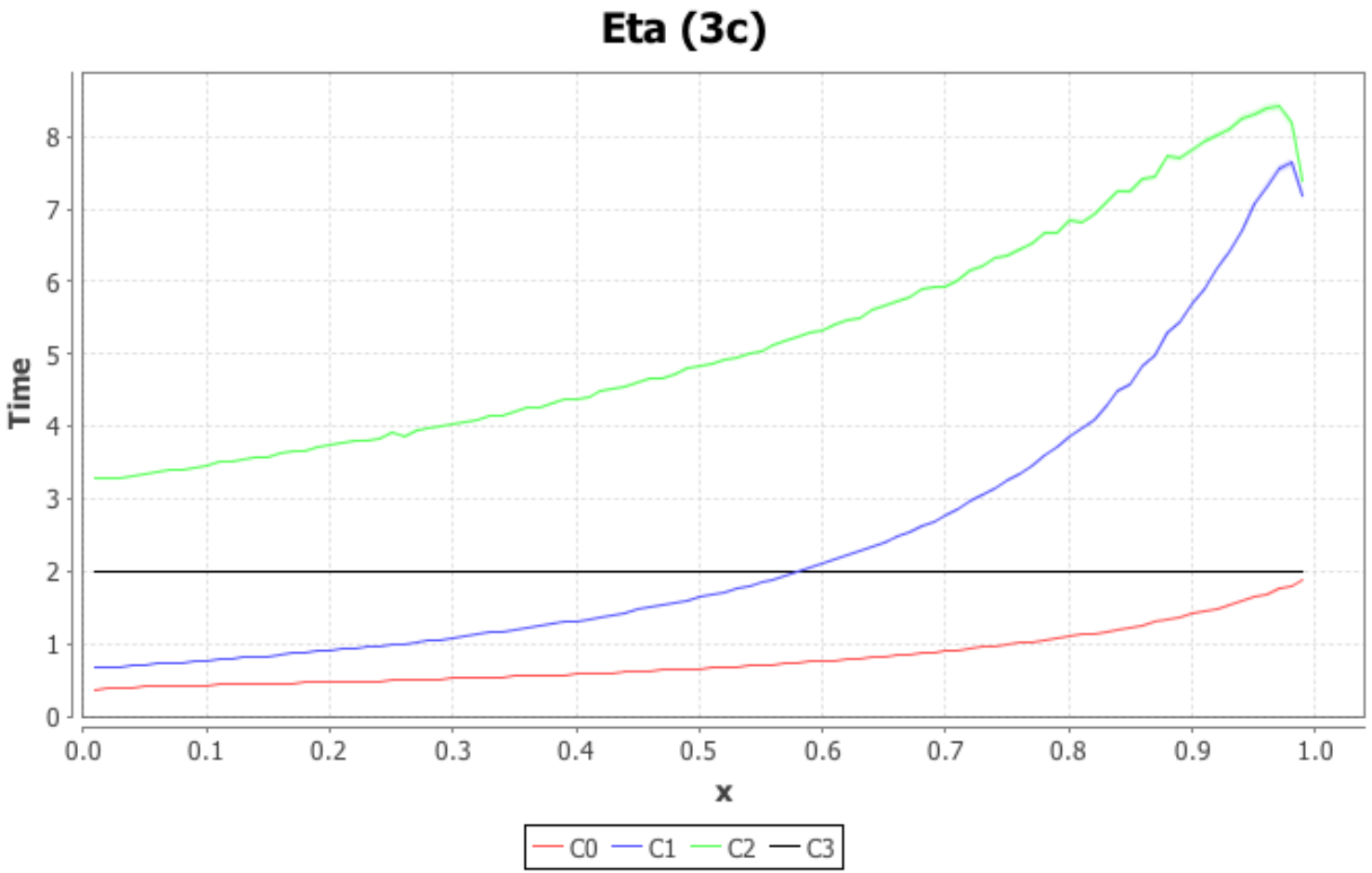

Figura: Tempi medi di attesa nel sistema con coda a tre classi (tipo c)

Precisione

Nota

Nel secondo grafico mostrato non si apprezzano a pieno gli intervalli di confidenza. Ciò è dovuto alla quantità di dati parcolarmente elevata utilizzata, che ha determinato una notevole precisione dei risultati (oltre ad una notevole durata in termini di tempo di simulazione).

Svolgimento della simulazione

Lo svolgimento della simulazione è definito all'interno della classe `SimulationRunners` ed è suddiviso in due fasi (metodi):

- `simulateMG1PrioWithVariableRhos(...)`
  in cui si definiscono le ρ relative alle varie classi.
- `simulateMG1Prio(...)`
  in cui si svolge la simulazione vera e propria.

Dettagli tecnici

- Valutazione empirica delle probabilità di stato k (numero di utenti presenti nel sistema)
- Metodo `simulateMG1EvaluatingProbability()`
- Si utilizza `FCFSSimulator` con una sola classe di priorità
- Utilizzo del metodo ad-hoc del simulatore, `getStatesProbability`
- Tempo totale trascorso nel particolare stato / tempo totale simulazione

Probabilità di stato

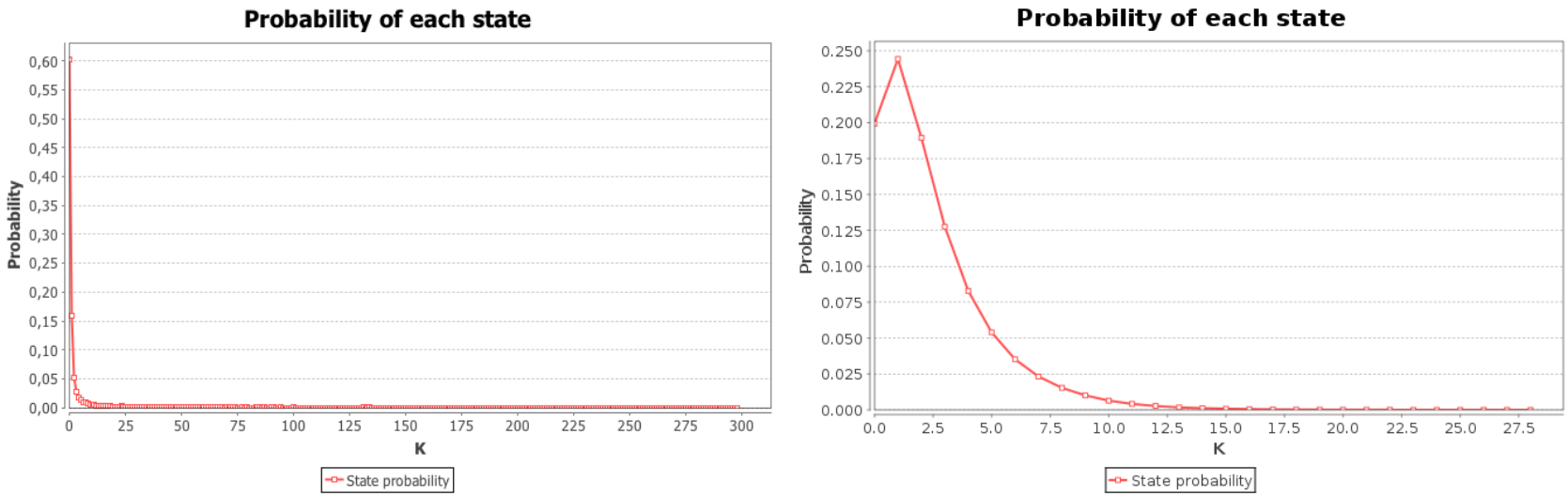

Figura: Probabilità di stato per sistemi con tempi di servizio di Pareto e Deterministici

Simulatore $M/G/1//SJN$

- Sistema a singolo servitore con politica di gestione della coda SJN
- Valutazione dell'η medio per ogni classe di discretizzazione dei valori di θ
- Discretizzazione logaritmica, parametrica
- Concentrazione di piccoli intervalli di discretizzazione attorno alla media
- Simulazione tramite `SJNSimulator`, che concretizza `Simulator` specializzando la politica di ordinamento della coda mediante la classe ServiceTimeComparedEvent, discendente di ComparableEvent, che ordina in base al tempo di servizio

η medio per SJN

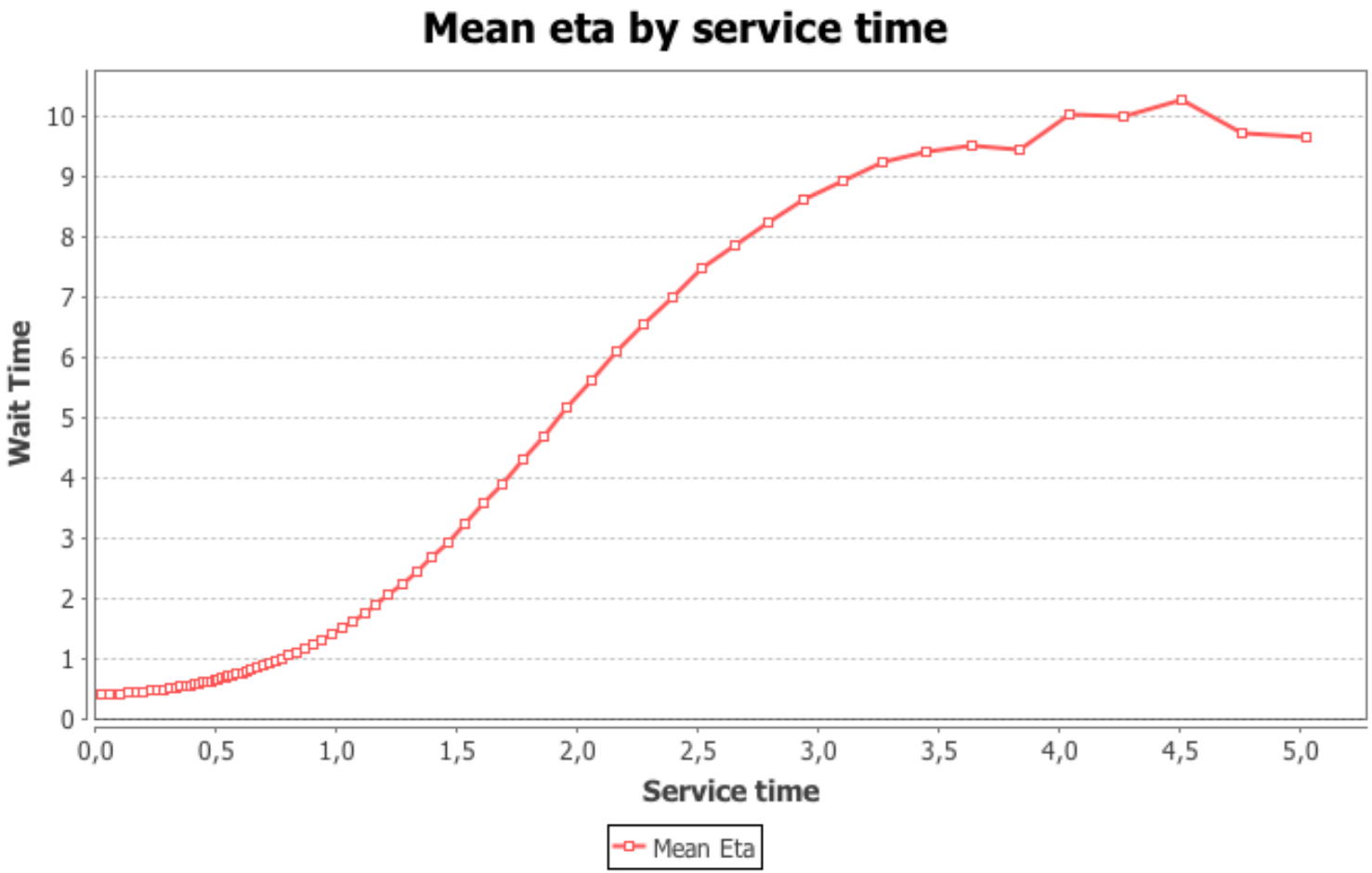

Figura: $\overline{\eta}$ in funzione del tempo di servizio in un sistema con coda SJN

Quantizzatore logaritmico

- Discretizzazione logaritmica al fine di ottenere una maggiore precisione dei risultati in concomitanza dell'intorno del valor medio di θ
- Intervalli più piccoli nei pressi della media, e più laschi e di dimensioni sempre maggiori allontanandosi da esso

Funzione di discretizzazione

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

Struttura package

- `gui` contiene il frame principale del simulatore unitamente alla finestra di console sulla quale vengono visualizzati i messaggi di log
  - `panels` contiene i differenti pannelli che compongono la finestra principale
- `simulator` contiene la logica delle differenti simulazioni unitamente al simulatore stesso (con le sue specializzazioni) e classi di supporto per tracciare il progresso delle varie simulazioni
  - `distribution` contiene la classe astratta `Distribution` e le sue concretizzazioni
  - `events` contiene gli eventi gestiti dal simulatore unitamente ai wrapper funzionali alla loro politica di ordinamento
  - `misc` contiene classi di utilità per la realizzazione del simulatore, quali ad esempio il quantizzatore logaritmico e funzionalità statistiche di supporto
  - `random` contiene le classi utili all'implementazione o incapsulamento dei vari generatori di numeri casuali utilizzati all'interno del simulatore
- `launchers` contiene gli entry point dell'applicazione, unitamente ad alcune simulazioni notevoli non accedibili mediante interfaccia grafica

Executor

Al fine di scaricare l'onere computazionale derivante dallo svolgimento di ogni singola simulazione esternamente alla *gui* si ricorre all'utilizzo di un *worker-thread* al quale vengono demandati tutti i task esterni all'interfaccia grafica.

Particolare tipo di esecutore, `SingleThreadExecutor`, al quale vengono sottomesse tutte le simulazioni

Eventuali paramter-checking e feedback visuali vengono lasciati, come di dovere, all'`Event Dispatch Thread`

Grafici delle simulazioni

Utilizzo di *JFreeChart* [2]

- permette la realizzazione di grafici all'interno dell'environment Java
- permette l'esportazione dei grafici prodotti in diversi formati di immagine
- consente di navigare, ridimensionare, zoomare ed editare i chart tramite interfaccia grafica

Vantaggio di poter generare i grafici direttamente da codice, senza bisogno di utilizzare tool esterni e senza dover estrarre ed esportare i dati dal simulatore

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

Logging delle simulazioni

Utilizzo di una console grafica, di supporto al frame principale, che permette di visualizzare i trace log stampati durante le simulazioni

- aumenta il livello di comprensione delle simulazioni visualizzando i valori precisi sia dei risultati che di alcune fasi intermedie
- bottone ad-hoc nella finestra principale che consente di far apparire/scomparire la console

Utilizzo di *log4j* [3] per le utility di logging

- permette la realizzazione agile di logging all'interno dell'environment Java
- facilita sia il filtraggio che la specifica della *verbosità* dei log
- permette di redirezionare le entry di log su differenti canali di output in maniera molto agile

[3] http://logging.apache.org/log4j/

Conclusioni

Un simulatore di teletraffico aiuta, in generale, l'analisi empirica di sistemi soggetti a traffico i quali, per loro natura, sono relativamente complessi

- aiuta a validare modelli che studiano le grandezze di interesse di questi sistemi
- permettono di farsi un'idea, in breve tempo, del comportamento del sistema variandone agilmente le condizioni al contorno e di utilizzo
- rappresenta una risorsa di indiscutibile utilità per la sintesi ed il dimensionamento di tali sistemi per applicazioni reali, rendedo possibile un'immediata rappresentazione e valutazione di caratteristiche e dinamiche altrimenti difficili da catturare

Bibliografia

- F. Callegati, G. Corazza *"Elementi di teoria del traffico per le reti di telecomunicazioni"* Società Editrice Esculapio, Bologna 2006
- W. Cerroni - *Lucidi del secondo modulo del corso di Progetto di reti di telecomunicazioni* - DEIS, Bologna 2010
- M. Matsumoto, T. Nishimura *"Mersenne Twister: A 632-dimensionally equidistribuited uniform pseudorandom number generator"* Keio University 1997
- David Gilbert *"The JFreeChart Class Library - Developer Guide (v.1.0.9)"* Object Refinery Limited 2000-2008

Traffic Simulator

Michael Gattavecchia Marco Santarelli Andrea Zagnoli

ALMA MATER STUDIORUM
II Facoltà di Ingegneria - Sede di Cesena
Tecnologie e Sistemi per la Sicurezza

16 giugno 2010